第一届
全国大学生集成电路创新创业大赛
NCICC（邀请赛）

参赛题目　深度学习——卷积神经网络(CNN)优化

参赛高校　北京航空航天大学

团队名称　奋进队

参赛队员　程浩 李金昊 贤仪寒

指导教师　欧阳鹏

完成时间　2017.4.6

参赛须知与承诺

1.参赛者必须保证参赛作品为首次参赛的原创作品，参赛作品不存在权利争议或侵犯第三方知识产权的行为，违反者自行承担相应责任。

2.参赛作品有下列情形，主办单位将取消该队伍的参赛资格或追回原授予的荣誉及奖励：

经人检举或告发为他人代劳或违反本竞赛相关规定，有具体事实者并经查证属实者；

经人检举或告发，涉及著作权、专利权等侵害，且侵权行为一旦经法院判决属实。

3.参赛作品的专利权、著作权等知识产权均归属该参赛队伍所有，但主办单位享有以原作者名义发表该项作品的权利，参赛队伍需全程参与由主办单位所举办的学术及推广教育活动。

4.参加队伍应尊重大赛评审委员会的决议，除非能具体证明其他作品违反本办法相关规定，不得有其他异议。

5.大赛阶段缴交所有文件将不退回，请参赛队伍自行备份。

承诺人：

（请全体团队成员签名）

欧阳鹏　程浩 李金昊 贤仪寒

2017 年 4 月 6 日

摘要

本文主要阐述了在硬件 Zynq 平台上实现卷积层算法的思路、方法及具体实现。卷积层算法在硬件平台上的实现，有很重要的现实意义。卷积层算法作为深度卷及神经网络（CNN）的核心，如果在硬件平台上高效地实现，意味着我们可以将深度学习算法应用在嵌入式系统中，增强其智能化。现今的神经网络由于算法复杂、占用资源空间大，大多数用在计算机平台上，如果能对其进行优化使其能高效、快速地在硬件平台上实现的话，将会带来巨大的现实效益。

我们的设计思路大致分为几个部分。首先我们先使用 C 语言代码进行算法的仿真，并在软件层面对 C 代码进行最大程度的优化。其次，我们在 vivado HLS 环境下进行硬件仿真，并在硬件层面进行优化，最后转化为硬件代码在 Zynq 平台上实现了卷积层算法。硬件层面优化的目的在于尽可能充分利用板子的乘法器等资源，尽可能提高执行的效率。

我们的创新点主要有：考虑 CNN 输入图像中存在大量的 0 值而变得稀疏的特点，构建异步计算架构，使得每个计算资源都可以负载均衡，解决传统同步计算下吞吐受限问题；构建卷积核间数据复用机制，减少内存访问；构建卷积核内并行计算机制，大大提高计算速度；计算架构支持卷积运算和反卷积运算，适用于深度生成网络和深度辨别网络；平衡接口带宽，存储带宽，以及计算吞吐，构建全流水架构，实现 CNN 流计算。

我们在硬件仿真过程中，测得系统时钟频率为 2.114GHz，共花了 791M 个时钟周期，达到 1O2GMAC。我们基本达到了要求。

关键字： 卷积层算法 CNN vivado 深度学习 Zynq

Abstract

This paper mainly introduces the idea, method and implementation of convolution layer algorithm on hardware Zynq platform. Convolution layer algorithm in the hardware platform to achieve, there is a very important practical significance. Convolution layer algorithm, as the core of deep volume and neural network (CNN), if it is implemented on the hardware platform, means that we can apply the depth learning algorithm to the embedded system to enhance its intelligence. Today's neural network because of the complexity of the algorithm, the use of large resource space, most used in the computer platform, if it can be optimized to make it efficient and fast on the hardware platform to achieve, it will bring great real benefits.

Our design ideas are broadly divided into several parts. First of all, we first use the C language code for algorithm simulation, and in the software layer of the C code to maximize the optimization. Secondly, we in the vivado HLS environment for hardware simulation, and in the hardware level to optimize, and finally converted into hardware code Zynq platform to achieve the convolution layer algorithm. The purpose of the hardware level optimization is to make full use of the board multiplier and other resources, as far as possible to improve the efficiency of the implementation.

Our innovative points are: consider the CNN input image in a large number of 0 values and become sparse characteristics, the construction of asynchronous computing architecture, making each computing resource can load balance, to solve the traditional synchronization calculation under the throughput problem; Conforming the data between the inter-core data reuse mechanism, reduce the memory access; build the convolution kernel parallel computing mechanism, greatly improve the computing speed; computing architecture to support convolution and deconvolution operations, suitable for deep generation of network and deep identification network; Interface bandwidth, storage bandwidth, and computing throughput, to build a full-flow architecture, to achieve CNN flow calculation.

In the hardware simulation process, the system clock frequency we measured is 2.114GHz, and spent a total of 791M clock cycles.The GMAC reached to 102.They

basically meet the requirements of contests.

Key words: algorithm of convolution layer, CNN,vivado,deep learning,Zynq

目录

1 总体设计

1.1 问题描述

题目中有以下三点要求：

（1）将卷积算法在硬件平台上实现并优化，主要考虑总线带宽，内部缓存，Pipeline 设计，计算单元等因素，给出不同缓存下，总线带宽计算公式和典型案例下的数据；

（2）给出 100G MAC 加速单元的架构、框图和整体的 Pipeline 设计；

（3）输入图像和输出图像的 channel 数目一般比较大，可达到 1024，图像的分辨率一般为 150x150。

1.2 总体设计思想

我们的总体设计思想是这样的，首先在 C++平台上实现卷积核算法和反卷积算法，然后通过 vivado HLS 仿真进行总线带宽、pipeline、内部缓存、计算单元等方面硬件的优化，仿真波形和实际相符后，生成相应的 IP 核，并在 zynq 平台上实现。概括出来就是：

（1）C++实现；

（2）Vivado HLS 仿真、优化；

（3）硬件 zynq 平台实现。

下面我们将详细介绍我们的各个模块。

2 C++软件平台实现

2.1 卷积核公式

卷积核的公式如下：

$$o(n,i,j)=\sum_{m=0}^{xn-1}\left(\sum_{v=0}^{kh-1}\left(\sum_{u=0}^{kw-1}w(n,m,u,v)\,x(m,u+i*sw,v+j*sh)\right)\right)$$

其中：kw 为单个卷积核的水平宽度，kh 为单个卷积核的垂直高度，sw 为水平方向的滑动窗口 step，sh 为垂直方向的滑动窗口 step。表示输入图像的

channel 数，$o(n,i,j)$ 表示第 n 个输出图像内(i,j)的输出值，$x(m,i,j)$ 表示第 m 个输入图像内(i,j)的输入值，$w(n,m,u,v)$ 表示 weight 的值，n 对应输出图像的 channel，m 对应输入图像的 channel，u 和 v 表示对应单独卷积核的位置[1]。

2.2 C++软件设计

2.2.1 定义

首先我们定义了一个图像的结构体 Feature：

```
typedef struct Feature
{
    unsigned char layer_in[CHANNEL] [LENGTH_FEATURE_IN][LENGTH_FEATURE_IN];
    unsigned char output[CHANNEL][LENGTH_FEATURE_OUT][LENGTH_FEATURE_OUT];
}Feature;
```

图 1 图像的结构体定义

结构体内有两个三维数组，layer_in 代表输入图像，output 代表输出图像，其中 CHANNEL 代表图像的通道数，LENGTH_FEATURE_IN 及 LENGTH_FEATURE_OUT 代表图像像素宽度。由题目知，我们的通道数采用 1024，输入图像像素宽度为 150，我们采用的卷积核像素宽度为 3，故输出图像的像素宽度为 148。

同时我们定义了卷积核的结构体 LeNet5：

```
typedef struct LeNet5
{
    unsigned char   kernel  [SUM_KERNEL][CHANNEL][LENGTH_KERNEL][LENGTH_KERNEL];
}LeNet5;
```

图 2 内核的结构体定义

卷积核是一个三维数组，SUM_KERNEL 代表卷积核的数量，LENGTH_KERNEL 代表卷积核的宽度，我们采用 SUM_KERNEL=1024，LENGTH_KERNEL=3。

2.2.2 算法实现

定义了相关的结构体之后，我们便根据公式给出了算法的代码实现：

```
int CONVOLUTION(Feature *feature, LeNet5 *lenet, int NUMBER_KERNEL, int LAYER_CHANNEL)
{
    //循环：生成m个n层特征图
    for (int Y_FEATURE = 0; LENGTH_FEATURE_OUT > Y_FEATURE; Y_FEATURE++)        //一层内卷积点y轴的循环
    {
        for (int X_FEATURE = 0; LENGTH_FEATURE_OUT > X_FEATURE; X_FEATURE++)    //一层内卷积点x轴的循环
        {
            CONVOLUTION_label1:for (int Y_KERNAL = 0; LENGTH_KERNEL > Y_KERNAL; Y_KERNAL++) //在某点卷积时Y轴相加
            {
                for (int X_KERNAL = 0; LENGTH_KERNEL > X_KERNAL; X_KERNAL++)
                {
                    if (!feature->layer_in[LAYER_CHANNEL][Y_FEATURE + Y_KERNAL][X_FEATURE + X_KERNAL])
                    {
                        continue;
                    }
                    int product =
                        feature->layer_in[LAYER_CHANNEL][Y_FEATURE + Y_KERNAL][X_FEATURE + X_KERNAL] *
                        lenet->kernel[NUMBER_KERNEL][LAYER_CHANNEL][Y_KERNAL][X_KERNAL];
                    feature->output[LAYER_CHANNEL][Y_FEATURE][X_FEATURE] += product;
                }
            }
        }
    }
    return 0;
};
```

```
for (int NUMBER_BATCH = 0; NUMBER_BATCH < BAT_SUM; NUMBER_BATCH++)
{
    for (int NUMBER_KERNEL = NUMBER_BATCH * BAT_KERNEL; (NUMBER_BATCH+1)*BAT_KERNEL > NUMBER_KERNEL; NUMBER_KERNEL++)   //多个个卷积核的循环
    {
        for (int LAYER_CHANNEL = 0; CHANNEL > LAYER_CHANNEL; LAYER_CHANNEL++)       //卷积核内多层(对128层的输入)的循环
        {
            CONVOLUTION(feature, lenet, NUMBER_KERNEL, LAYER_CHANNEL);  //卷积操作
        }
    }
}
```

图 3 卷积算法的核心 C++代码

这样通过六层的嵌套循环，卷积核算法便实现了。

在这个算法里面，我们针对工程中输入图像有很多元素为 0 的情况，做了一些优化：

```
if (!feature->layer_in[LAYER_CHANNEL][Y_FEATURE + Y_KERNAL][X_FEATURE + X_KERNAL])
{
    continue;
}
```

图 4 代码优化

如果某个元素为 0，我们直接跳过这层循环，这样可以大大提高执行效率。

2.3 软件拓展——反卷积算法

如果说卷积是用来将图像变小，来突出输入图像的特征的话，反卷积就是将图像变大，用来生成原图像。从本质上，反卷积其实就是转置卷积核，因此我们的这个算法也可以用来计算反卷积。

2.2.3 软件测试

实现了算法之后，我们立刻在 C++平台上运行，对于 1024 个通道的图像，程序顺利执行，经过 3050s 后运行完成。

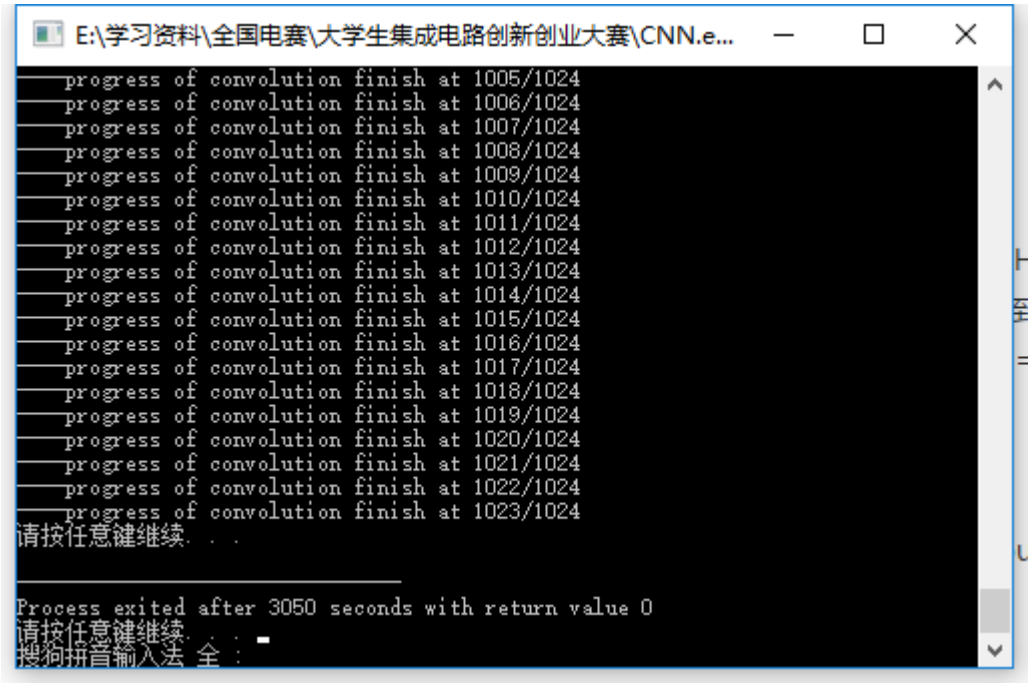

图 5 代码运行情况

下面是输出的图像像素（由于有 1024 个 channel，不可能一一枚举，所以截取了一部分）

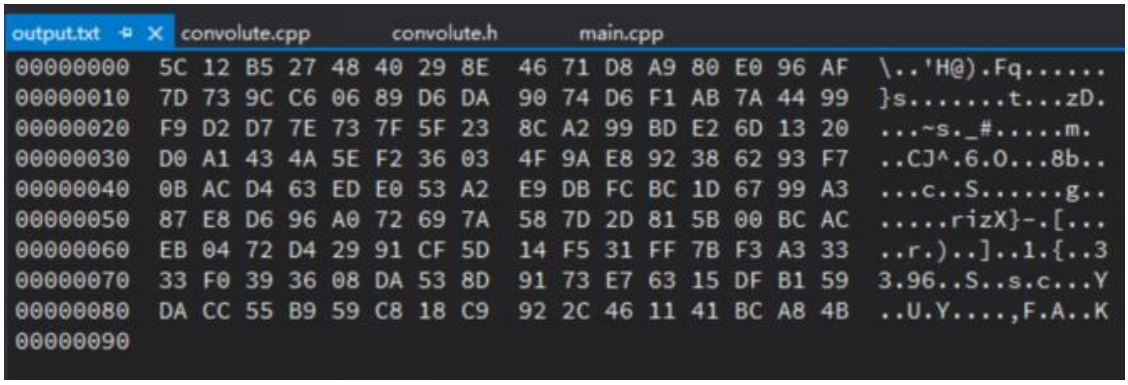

图 6 C++运行输出图像部分数据

3 硬件架构优化

3.1 卷积和反卷积的算法说明

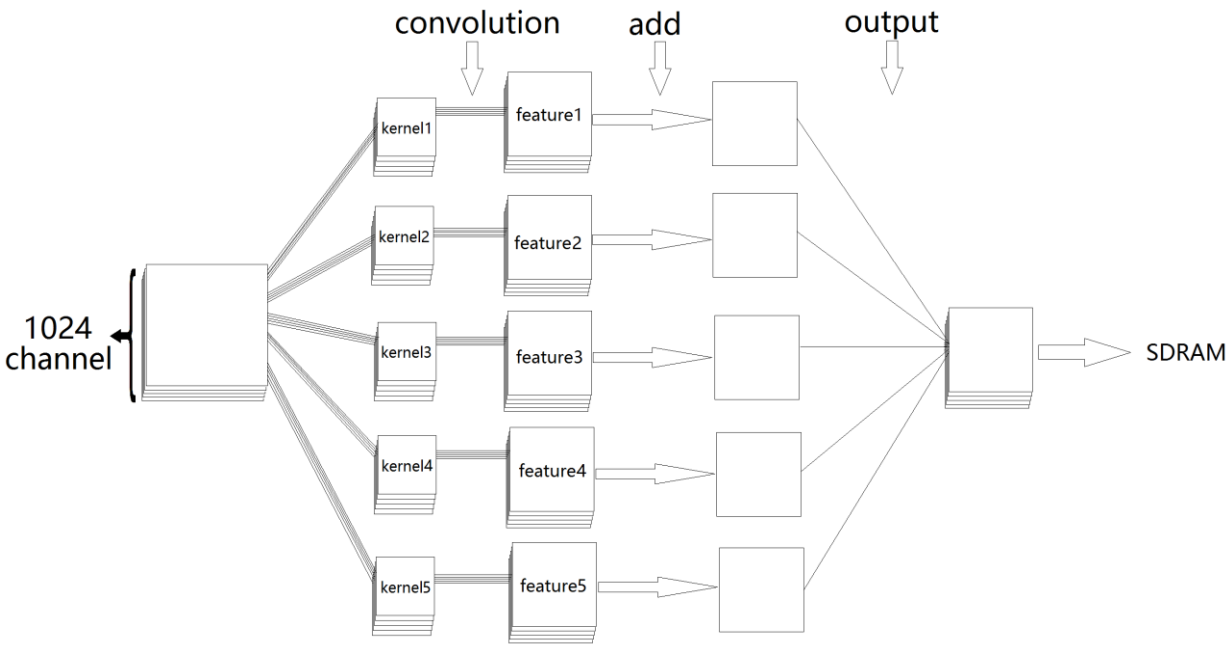

图 7 卷积和反卷积图示

上图所示描述出了卷积层算法的大致流程。

图像卷积，是用卷积核在图像上扫过，并对每个元素进行加权和偏置，得到的结果是特征图。

本题目的要求，是 1024 通道的图片，对应上图的第一列就是输入的 1024×150×150 三维图像。

第二列描述了所有的卷积核。与图像对应我们的卷积核也有 1024 个，每个卷积核是 1024×3×3 的三维卷积块。

第三列是卷积后的特征，卷积时某一个通道的图片将被所有卷积核的对应层卷积。这样卷积过后将得到与卷积核同等数量的特征块。

第四列描述了相加过程。同一个卷积核的卷积结果要相加，即 1024 层图片最终相加得到一张图片。

第五列是最终的输出结果，将每一个卷积核的特征组合在一起，得到 1024 通道，148×148 像素的特征块，是卷积层的输出结果。

反卷积（Deconvolution）和卷积在 CNN 中，虽然原理有所不同，但是卷积的过程其实是一致的，因此上图可以同时表示卷积和反卷积的过程。

3.2 vivado HLS 简介

Vivado HLS 开创了一种全新理念，那就是用户可以通过 C/C++代码即可直接生成硬件执行文件，这使得很多用户不需要掌握特别深入的硬件知识也可进行硬件开发，降低了硬件开发的瓶颈，同时使得开发者把精力集中在优化算法和数据结构上，节省了大量时间。同时 vivado HLS 可以智能地进行内部资源分配，实现硬件优化。基于这些优势，我们采用 vivado HLS 对我们的 C++代码进行硬件仿真和优化[2]。

3.3 优化

针对 vivado HLS 特点和我们的代码结构，我们进行了以下几点优化：

3.3.1 pipeline，unroll 及使用 DSP

pipeline，unroll 所有的循环，pipeline 使得并行度大大提高。

Solution1：对最底层循环 pipeline

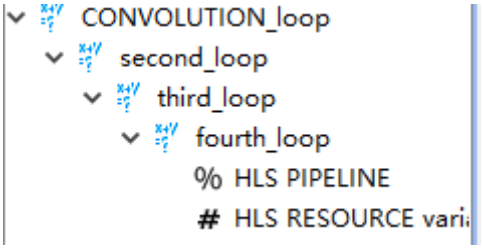

图 8 solution1

solution2：对 second 循环 pipeline

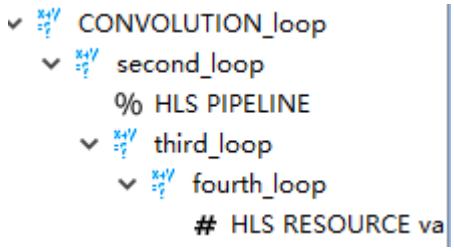

图 9 solution2

solution3：对顶层循环 pipeline

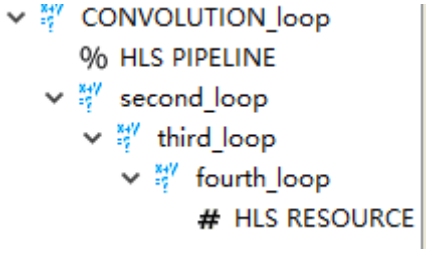

图 10 solution3

可以看出，pipeline 顶层循环并行程度最高，latency 和 interval 最小，使用的 DSP 最多，故随后的 solution 我们只对最顶层循环 pipeline。

3.3.2 DATAFLOW

对顶层函数 CNN 执行 DATAFLOW，对数据流的粗粒度进行优化。DATAFLOW 是提高数据吞吐量的有力方法。这一步和上一步都是粗优化。

3.3.3 数据类型优化

根据处理数据的类型和位宽进行优化，降低资源消耗。我们采用的图像像素的数据类型都是 8 位的，故可将数据类型设置为 uint_8。其实 unsigned char 类型位宽也为 8 位，故我们的数据类型已经进行了优化，可以跳过这一步。

3.3.4 数组分配

对 feature->output 和 lenet->kernel 这两个数组进行数组分割，进一步提高并行度。

```
%pragma HLS ARRAY_PARTITION  variable=feature->output  complete dim=2
%pragma HLS ARRAY_PARTITION  variable=lenet->kernel  complete dim=2
```

3.3.5 INLINE

对顶层函数进行 INLINE，删除所有函数层次结构。用于通过功能边界实现逻辑优化，并通过减少功能调用开销来改善延迟/间隔[3]。

各个 solution 的对比情况如下：

Performance Estimates

- Timing (ns)

Clock		solution1	solution2	solution3	solution4	solution5
ap_clk	Target	10.00	10.00	10.00	10.00	10.00
	Estimated	9.09	9.09	6.38	9.09	6.38

- Latency (clock cycles)

		solution1	solution2	solution3	solution4	solution5
Latency	min	1273	652	326	1273	326
	max	1597	652	326	1597	326
Interval	min	1274	653	327	1274	327
	max	1598	653	327	1598	327

Utilization Estimates

	solution1	solution2	solution3	solution4	solution5
BRAM_18K	0	0	0	0	0
DSP48E	1	1	54	1	54
FF	127	163	1103	128	1104
LUT	181	291	958	181	958

Resource Usage Implementation

	solution1	solution2	solution3	solution4	solution5
RTL	vhdl	vhdl	vhdl	vhdl	vhdl
SLICE	0	0	0	0	0
LUT	93	162	613	88	595
FF	77	115	243	77	243
DSP	1	1	53	1	53
SRL	0	0	0	0	0
BRAM	0	0	0	0	0

Final Timing Implementation

	solution1	solution2	solution3	solution4	solution5
RTL	vhdl	vhdl	vhdl	vhdl	vhdl
CP required	10.000	10.000	10.000	10.000	10.000
CP achieved post-synthesis	3.639	4.889	5.019	3.639	5.019
CP achieved post-implemetation	-	-	-	-	-

图 11 各个 solution 对比

通过上面的对比可以看出，对比 solution1-3，可以发现对外层循环 pipeline 可以大大提高并行度，减少 timing 和 latency。这是因为 pipeline 的循环越往外，展开并行的子过程越多。Solution4 相对于 solution1，我们对顶层函数进行了 DATAFLOW，使得图像数据的输入得到了改善。DATAFLOW 通过输入输出数据流，能够在尽量节省板内资源的基础上，达到相同的效果。可以看出 solution4 相对于 solution1,LUT 的数量减少了。Solution5 是我们最终的方案，我们同时又引入了 INCLINE，进一步改善了延迟。由于 Solution5 各方面性能最好，所以我们采用 solution5 作为最终优化的源码。核心部分如下所示：

```
#include "CNN.h"

void CNN(Feature *feature, LeNet5 *lenet,uint10 NUMBER_KERNEL,uint10 LAYER_CHANNEL)
{
#pragma HLS INLINE
#pragma HLS DATAFLOW
    int Y_FEATURE;
    CNN_label3:for (Y_FEATURE = 0; LENGTH_FEATURE_OUT > Y_FEATURE; Y_FEATURE++)      //一层内卷积点y轴的循环
    {
#pragma HLS PIPELINE
        int X_FEATURE;
        CNN_label4:for (X_FEATURE = 0; LENGTH_FEATURE_OUT > X_FEATURE; X_FEATURE++) //一层内卷积点x轴的循环
        {
            int Y_KERNAL;
            CNN_label5:for (Y_KERNAL = 0; LENGTH_KERNEL > Y_KERNAL; Y_KERNAL++) //在某点卷积时y轴相加
            {
                int X_KERNAL;
                CNN_label6:for (X_KERNAL = 0; LENGTH_KERNEL > X_KERNAL; X_KERNAL++)
                {
                    if (!feature->layer_in[LAYER_CHANNEL][Y_FEATURE + Y_KERNAL][X_FEATURE + X_KERNAL])
                    {
                        continue;
                    }
                    int product;
#pragma RESOURCE HLS variable=product core=DSP_48
                    product=feature->layer_in[LAYER_CHANNEL][Y_FEATURE + Y_KERNAL][X_FEATURE + X_KERNAL]
                        lenet->kernel[NUMBER_KERNEL][LAYER_CHANNEL][Y_KERNAL][X_KERNAL];
                        feature->output[LAYER_CHANNEL][Y_FEATURE][X_FEATURE] += product;

                }
            }
        }
    }
};
```

图 12 最终优化结果核心源码

3.4 最终的硬件架构

首先我们先给出一个计算单元的框图：

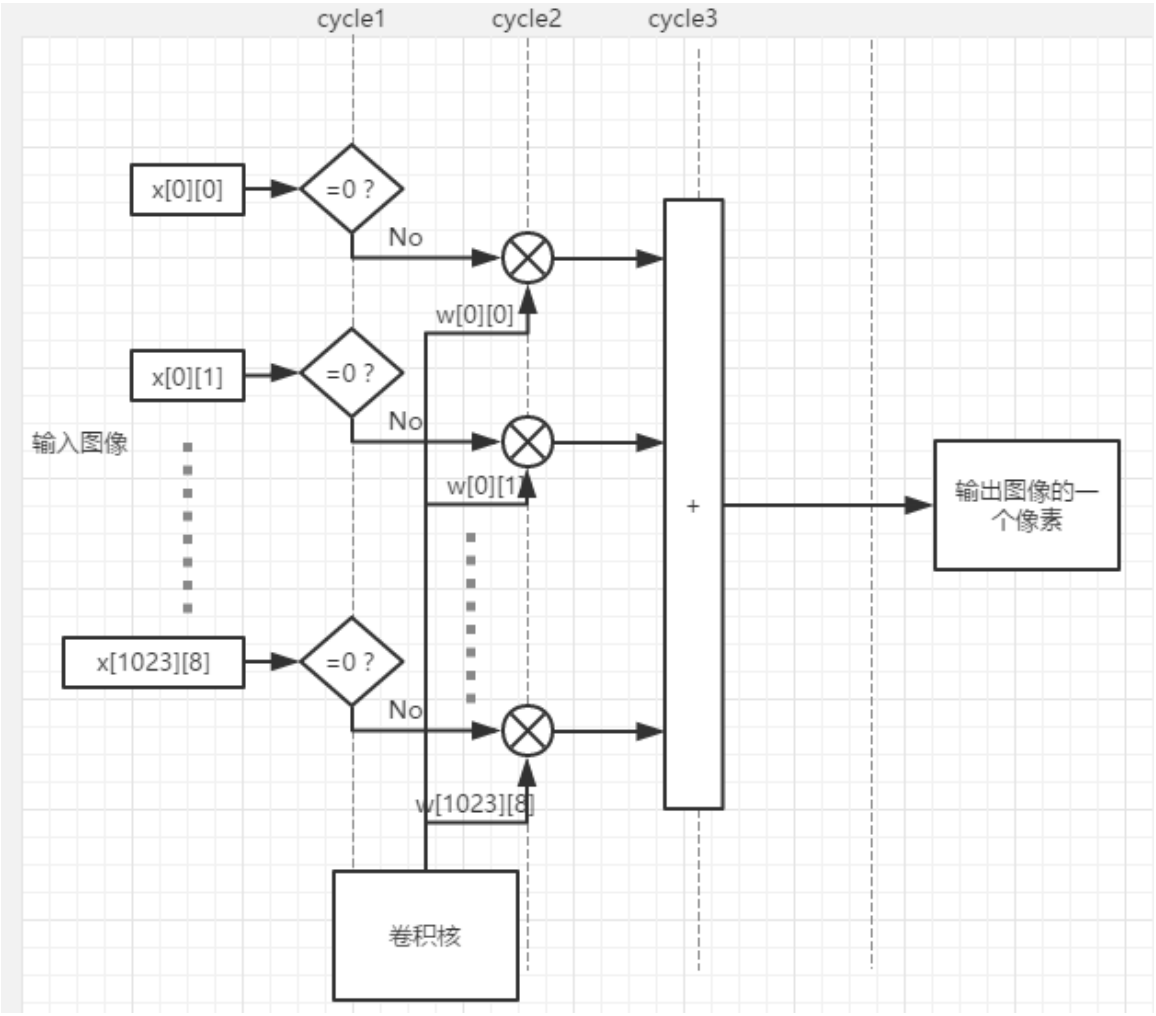

图 13 计算单元框图

输入图像的每个像素先进行是否为 0 的判断，若为 0，则直接跳过与卷积核的权重相乘，否则乘以卷积核。遍历了 1024 个 kernel 的所有元素（1024*3*3=9216）后，相加即得输出图像的一个像素值。此计算单元可以计算输出图像的一个像素，可以令这个计算单元整体为 cell，进而可以得到整个系统的硬件架构如下：

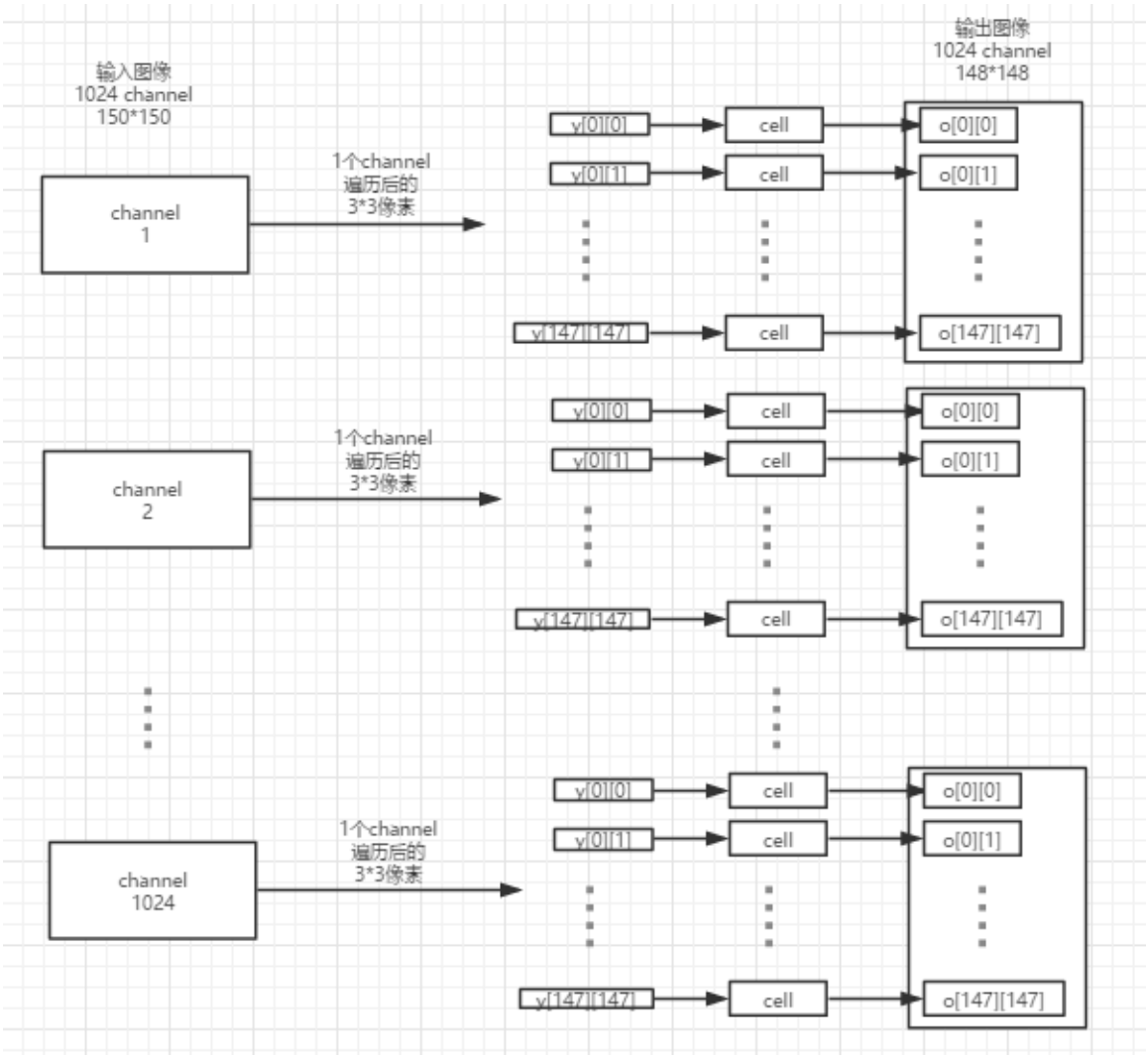

图 14 整体硬件架构

对输入图像数据进行 DATAFLOW 和 INLINE 处理，同时对卷积核进行优化（零元素判别，即若卷积核有 0，则跳过对该元素的运算），两部分数据作为 CNN 网络的参数，输入 CNN 网络。紧接着进行 pipeline，unroll 最外层循环，展开 CNN 网络，提高并行度。然后进一步指定板内硬件资源的使用，使其均衡化。最后高效、快速、准确地输出所需的图像数据。

3.5 RTL 仿真和生成 IP 核

C synthesis 生成 solution 后，进一步进行 RTL 仿真，仿真波形如下：

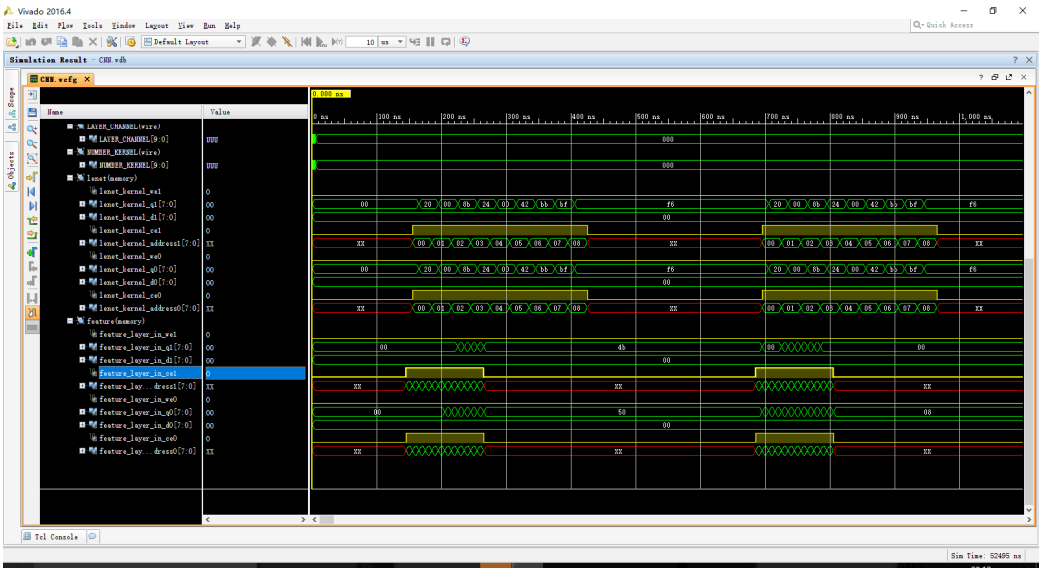

图 15 RTL 仿真波形

通过观察输入输出端口的仿真波形，我们有一些发现。从仿真图上可以清晰地看到每个时钟周期、每个输入输出口的波形变化。vivado HLS 将输入图像数据拆分成几个 array 并行输入，而对每个 array 数据是串行输入的，表现为时序的变化。每一路的波形疏密相间，稀疏的部分是进行读取的过程，密集的部分是进行数据处理和运算的过程。仿真波形是比较复杂的，导致我们很难对其进行检查，以及和软件运行的结果进行比对。所以，我们采用最终的生成文件和软件的输出对比，来验证其优化的准确性。

紧接着我们生成 IP 核，导入 Vivado 里，准备进行硬件实现。

4 硬件实现

将生成的 IP 核导入到 vivado 中，生成硬件可执行文件之后，烧入 Zynq 平台（这里我们采用的是 ZedBoard 开发板）。输入 1024 个 channel 的 150*150 图像，卷积核采用 3*3 的，最终生成 1024channel 的 148*148 的图像。和软件仿真的结果一致。

硬件运行的情况如下图所示：

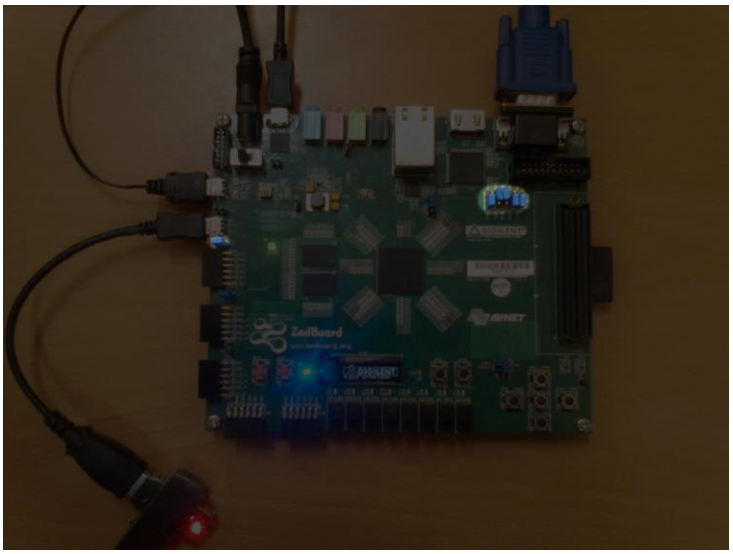

图 16 Zedboard 开发板运行情况

下面是输出的图像像素（由于 1024 较大，这里我们只截取了一部分）：

output.dat

```
00000000  5C 12 B5 27 48 40 29 8E  46 71 D8 A9 80 E0 96 AF
00000010  7D 73 9C C6 06 89 D6 DA  90 74 D6 F1 AB 7A 44 99
00000020  F9 D2 D7 7E 73 7F 5F 23  8C A2 99 BD E2 6D 13 20
00000030  D0 A1 43 4A 5E F2 36 03  4F 9A E8 92 38 62 93 F7
00000040  0B AC D4 63 ED E0 53 A2  E9 DB FC BC 1D 67 99 A3
00000050  87 E8 D6 96 A0 72 69 7A  58 7D 2D 81 5B 00 BC AC
00000060  EB 04 72 D4 29 91 CF 5D  14 F5 31 FF 7B F3 A3 33
00000070  33 F0 39 36 08 DA 53 8D  91 73 E7 63 15 DF B1 59
00000080  DA CC 55 B9 59 C8 18 C9  92 2C 46 11 41 BC A8 4B
00000090
```

图 17 硬件实际运行输出图像部分数据

可以看出，跟软件仿真输出的数据一致。

5 题目解答

5.1 整体框图

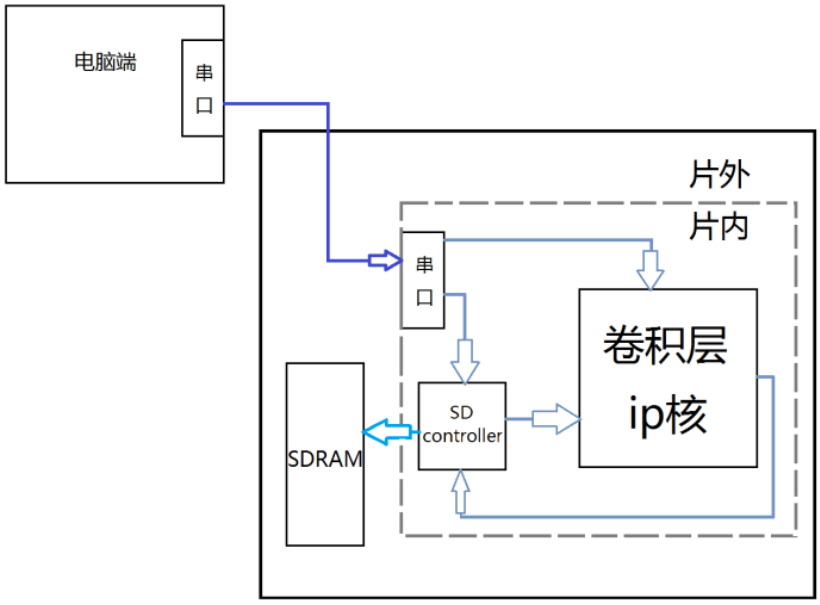

图 18 整体框图

我们在这里对我们的框图做一些具体的解释。Zynq 通过串口读取用来进行处理的输入图像，由于板内存储器不足以一次性大批量存储 1024 个 channel 的 150*150 的图像，所以我们这里采用动态存储的方法。串口将图像数据串行输入，Zynq 边对图像进行卷积处理，边读图像数据，一个 channel 处理完成后立即输出，并空出相应的存储资源供后续 channel 的图像使用。这样来保证硬件能够高效、流畅地运行。

5.2 pipeline 设计

对最外层循环进行 pipeline，unroll 所有循环。对最外层循环 pipeline 的原因是，在板内资源充足的情况下，能够最大程度地提高并行度和数据吞吐率，降低整体的 latency。

5.3 总线带宽

总线的带宽是数据在总线传输的速度，是用字节每秒做单位的。总线带宽由系统的时钟频率与单位处理数据位数决定[4]。

总线带宽=总线频率 * 位宽 /8

在本系统中，读取的处理数据分批次进入 FPGA 开发板的片外存储 SDRAM 中，在片内的 SDcontrolor 的控制下实现数据的读入，采用 16 位处理，即单次可处理两个字节。系统时钟通过 rtl 仿真达到为 3.693us，时钟频率达到 270.78MHz，最后得到总线带宽为 541.56Mb/s。

5.4 内部缓存

根据我们的设计思路，在本系统中，采用多通道共享缓存技术，整个缓存空间被分割成若干个独立的存储颗粒以后，首先需要有一个管理调度模块用来分配、调度、回收存储颗粒。管理模块根据各个通道的使用需求，动态的分配存储颗粒，并在通道释放该存储颗粒时将它回收，以供下一次分配使用[5]。

5.5 计算单元

我们通过 vivado HLS 对 C++代码进行优化，使得板内的计算和存储单元能够被充分利用，如 DSP，RAM、LUT、FF 等。进而我们生成了计算卷积的 IP 核，其内部有四层循环，以完成最基本的卷积运算。可参考 3.4，有详细介绍。

5.6 100G MAC 加速单元

在系统时钟确定下，进行乘加运算的次数与时钟频率的乘积是 MAC。本系统中，100G MAC 加速单元的设计没有达到要求。这个我们会在后续的工作中进行改进。

6 应用前景

我们在进行硬件架构优化之后，在得到题目的答案的同时，也生成了相应的 IP 核，这意味着我们的工作有更多的现实应用价值。我们的 IP 核不仅能够用来计算卷积，还可以用来计算反卷积。卷积和反卷积在神经网络优化、图像处理等领域都有重要地位，甚至是一些项目的核心算法。神经网络和图像处理领域，在当前的时代，都是热门领域，在生产和生活中有广泛的应用。我们的卷积核和反卷积核，作为其中的重要算法，在各行各业，如安防监控、刑侦破案、资源采集、

无人驾驶等领域都有重要的应用价值。

7 总结

在这次的比赛中，我们在对 CNN 算法进行优化，既从软件层面对代码进行了优化，如对稀疏的卷积核采用跳过 0 的操作；又从硬件层面进行了优化。硬件层面的优化是主要的，主要是从 Pipeline，Dataflow，Inline 等方面提高并行度、数据吞吐率，减小 latency，提高板内资源利用率。同时，我们还对反卷积算法进行了研究，拓展了我们的工作。当然，由于时间问题，我们没能实现时间调度机制，我们计划在后续的工作中进一步完善。总之，在这次比赛中，我们收获良多，丰富了我们的工程经验，对我们的科研能力也是很好的锻炼。

参考文献

[1] 附件 2 比赛题目及评分标准.pdf

[2] 张俊涛,付芳琪,曹梦娜. 基于 Vivado HLS 的 FFTIP 核设计与实现[J]. 电子器件,2016,(02):374-378.

[3] ug902-vivado-high-level-synthesis_.pdf

[4]http://baike.baidu.com/link?url=4fHzI2dienHiAjwobznUxf2SQ30NAEKJbOHoEp87wnOundhrX4kQl0r4Z4DXNENMIQidr5gXJ8ASUKg70iYSQrWzkF4PaTXh7C3zB3g5uVPDRv01DCLdLZ39pkpf9uq-

[5] http://blog.163.com/fpga_ip/blog/static/204443024201221314126695/

附录

Vivado HLS 优化后的代码：

（1） CNN.h

```
#pragma warning(disable:4996)
#define CHANNEL 1024
#define SUM_KERNEL 1024
#define LENGTH_KERNEL 3
#define LENGTH_FEATURE_IN 150
#define LENGTH_FEATURE_OUT (LENGTH_FEATURE_IN-LENGTH_KERNEL+1)
#define FILE_IMAGE     "train-images-idx3-ubyte"
#define FILE_LENET     "model.dat"
#define FILE_output "output.dat"
#include <stdio.h>
#include <stdlib.h>
```

```
#include "ap_cint.h"

typedef struct Feature
{
        unsigned char
layer_in[CHANNEL]      [LENGTH_FEATURE_IN][LENGTH_FEATURE_IN];
        unsigned char
output[CHANNEL][LENGTH_FEATURE_OUT][LENGTH_FEATURE_OUT];
}Feature;

typedef struct LeNet5
{
        unsigned char   kernel
 [SUM_KERNEL][CHANNEL][LENGTH_KERNEL][LENGTH_KERNEL];
}LeNet5;

void CNN(Feature *feature, LeNet5 *lenet,uint10 NUMBER_KERNEL,uint10
LAYER_CHANNEL);
int read_image(unsigned
char(*data)[LENGTH_FEATURE_IN][LENGTH_FEATURE_IN], const char
data_file[]);
int read_kernal(unsigned
char(*data)[CHANNEL][LENGTH_KERNEL][LENGTH_KERNEL], const char
data_file[]);
int write_image(unsigned
char(*data)[LENGTH_FEATURE_OUT][LENGTH_FEATURE_OUT], const char
data_file[]);
```

（2） CNN.cpp

```
#include "CNN.h"

void CNN(Feature *feature, LeNet5 *lenet,uint10 NUMBER_KERNEL,uint10
LAYER_CHANNEL)
{
#pragma HLS DATAFLOW
#pragma HLS INLINE
 int Y_FEATURE;
 CNN_label3:for (Y_FEATURE = 0; LENGTH_FEATURE_OUT > Y_FEATURE;
Y_FEATURE++)          //一层内卷积点 y 轴的循环
 {
#pragma HLS PIPELINE
      int X_FEATURE;
      CNN_label4:for (X_FEATURE = 0; LENGTH_FEATURE_OUT > X_FEATURE;
X_FEATURE++)    //一层内卷积点 x 轴的循环
      {
```

```
            int Y_KERNAL;
            CNN_label5:for (Y_KERNAL = 0; LENGTH_KERNEL > Y_KERNAL;
Y_KERNAL++)     //在某点卷积时 Y 轴相加
            {
                int X_KERNAL;
                CNN_label6:for (X_KERNAL = 0; LENGTH_KERNEL > X_KERNAL;
X_KERNAL++)
                {
                    if      (!feature->layer_in[LAYER_CHANNEL][Y_FEATURE      +
Y_KERNAL][X_FEATURE + X_KERNAL])
                    {
                        continue;
                    }
                    int product;
#pragma RESOURCE HLS variable=product core=DSP_48
                    product=feature->layer_in[LAYER_CHANNEL][Y_FEATURE     +
Y_KERNAL][X_FEATURE + X_KERNAL] *
lenet->kernel[NUMBER_KERNEL][LAYER_CHANNEL][Y_KERNAL][X_KERNAL];

 feature->output[LAYER_CHANNEL][Y_FEATURE][X_FEATURE] += product;

                }
            }
        }
 }
};
```

（3） main.cpp

```
#include "CNN.h"

int main()
{
        Feature  *feature  =  (Feature*)malloc(sizeof(Feature));  //(Feature  *)calloc(1,
sizeof(Feature));
        LeNet5     *lenet = (LeNet5*)malloc(sizeof(LeNet5));  //  (LeNet5  *)calloc(1,
sizeof(LeNet5));
        FILE *fp_feature, *fp_lenet;
        if  (read_image(feature->layer_in,  FILE_IMAGE)||  read_kernal(lenet->kernel,
FILE_LENET))
        {
                printf("program will be terminated!");
                return 0;
        }

        uint10 NUMBER_KERNAL;
```

```
            for (NUMBER_KERNAL = 0; SUM_KERNEL > NUMBER_KERNAL;
NUMBER_KERNAL=NUMBER_KERNAL+1)  //多个个卷积核的循环
            {
                uint10 LAYER_CHANNEL;
                for (LAYER_CHANNEL = 0; CHANNEL > LAYER_CHANNEL;
LAYER_CHANNEL++)        //卷积核内多层(对 128 层的输入)的循环
                {
                    CNN(feature,lenet,NUMBER_KERNAL,LAYER_CHANNEL);
                }

            }
            write_image(feature->output, FILE_output);
        return 0;
}


int read_image(unsigned
char(*data)[LENGTH_FEATURE_IN][LENGTH_FEATURE_IN], const char data_file[])
{
        FILE *fp_image;
        fp_image = fopen(data_file, "rb");
        if (!fp_image)
        {
                printf("error in loading image file!");
                return 1;
        }
        fseek(fp_image, 16, SEEK_SET);
        fread(data, sizeof(*data)*CHANNEL, 1, fp_image);
        fclose(fp_image);
        return 0;
}

int read_kernal(unsigned
char(*data)[CHANNEL][LENGTH_KERNEL][LENGTH_KERNEL],  const  char
data_file[])
{
        FILE *fp_kernal;
        fp_kernal = fopen(data_file, "rb");
        if (!fp_kernal)
        {
                printf("error in loading kernel file!");
                return 1;
        }
        fseek(fp_kernal, 0, SEEK_SET);
```

```
        fread(data, sizeof(*data)*CHANNEL, 1, fp_kernal);
        fclose(fp_kernal);
        return 0;
}

int write_image(unsigned
char(*data)[LENGTH_FEATURE_OUT][LENGTH_FEATURE_OUT], const char
data_file[])
{
        FILE *fp_out;
        fp_out = fopen(data_file, "wb");
        if (!fp_out)
        {
                printf("error in loading kernel file!");
                return 1;
        }
        fseek(fp_out, 0, SEEK_SET);
        fwrite(data, sizeof(*data)*CHANNEL, 1, fp_out);
        fclose(fp_out);
        return 0;
}
```