D1 开发板 YOLO 算法优化

1、实验目的

熟悉一个 RISC-V 架构的硬件平台，并在之上启动一个 Linux 操作系统。然后在此之上运行 YOLO 算法，之后需要对卷积层进行优化，在此过程会使用嵌入的汇编代码进行性能监测，并对优化的性能进行分析，在这个过程中我们可以把书中学习到的缓存原理知识应用到实际的工程开发中，并真切地感受到其发挥的作用。最后通过 HHB 流程对 RISC-V 的向量扩展的应用有一个初步的认识。

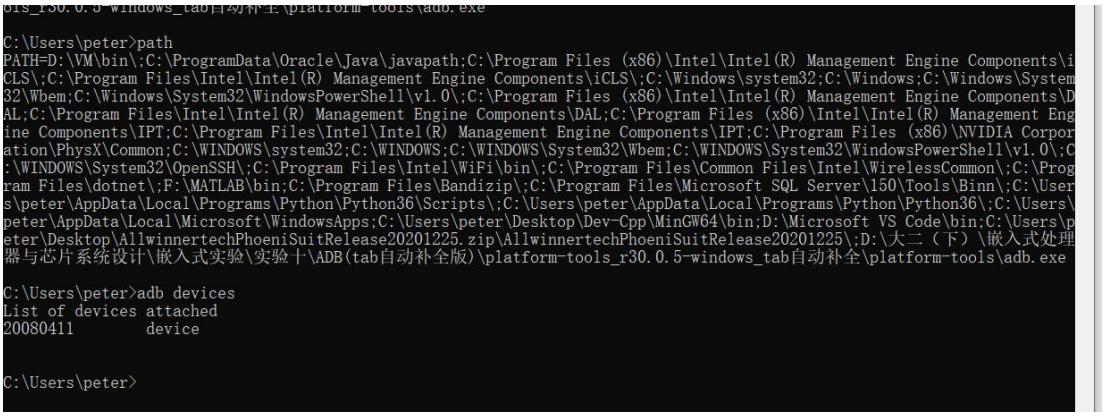
2、实验步骤

1) 将修改过的 Tina Linux 固件烧写到 D1 开发板中

首先下载 PhoenixSuit 固件烧写工具 PhoenixSuit，然后安装全志 USB 驱动，接下来用盒子中的数据线将全志开发板（type-c 口）与电脑（USB 口）连接起来后，运行 PhoenixSuit，过程中确保全志 USB 驱动安装成功。然后点击 PhoenixSuit 中的“一键刷机”，使用 tina_d1-nezha_uart0.img 并选择“全盘擦除升级”进行烧写。等待一会后烧写完成。

实验过程截图如下：

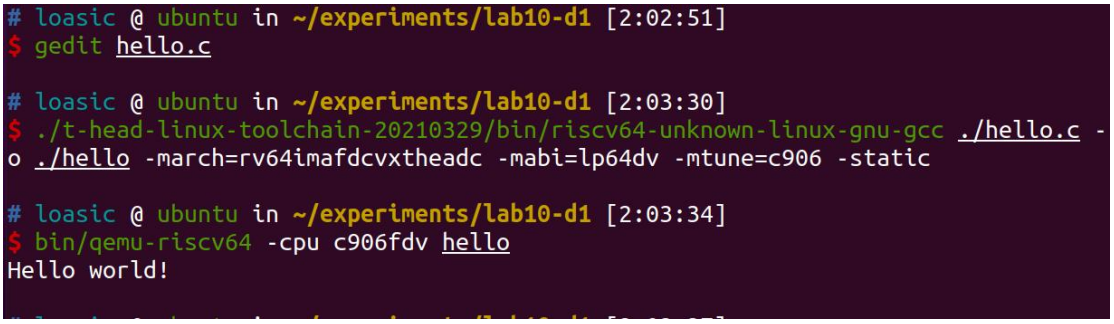
烧写完成后，在主机上使用 cmd 命令提示符查看设备连接情况。输入命令 adb devices 即可查看设备连接情况清单，成功连接后使用命令 adb shell 进入到全志开发板的板载 Linux 系统当中。



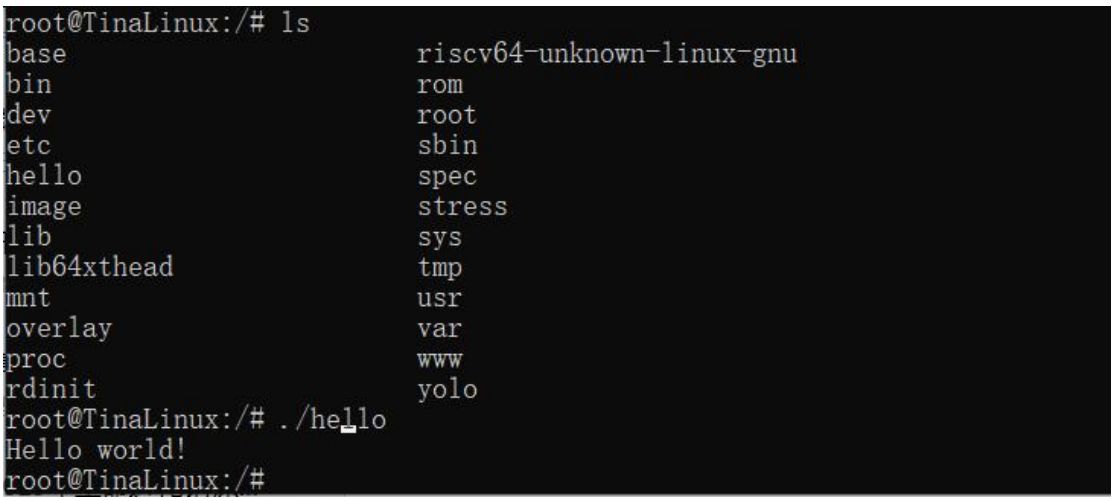
2) 在 D1 开发板上运行 hello_world 的全流程

现在虚拟机上安装好平头哥 RISC-V 交叉编译工具链以及与 C906 适配的 QEMU，此次实验选择 Linux 模拟器-QEMU (V4.1.0)。

编写一个简单的 hello.c 的 C 文件，使用工具链编译生成 riscv 的可执行程序后，我们先使用 qemu 进行模拟运行，得到正确的结果。



然后再在 D1 开发板上运行，也得到正确的输出结果。



3) 攥写带有缓存优化版本的卷积函数，并通过 qemu 的测试。（需要写出优化思路，以及优化的原理）

在 cache-opt 目录下修改 conventional_layer 当中的卷积函数部分。实现缓存优化的思路大致如下：通过数据重排的方法，将实际储存图像数据的一位数组不再以原有的 w、h、c 的顺序串联排列，而是掉换成 c、h、w 的顺序排列，相应的，在进行卷积计算时也不再是原来的 w*h*c 的遍历顺序去求乘积，而是 c*h*w 的顺序，这样一来就可以利用岛通道数 c 的移动上不存在步长，且这样的卷积和数据顺序可以大大缩小卷积计算

时读取一维数组当中的图像数据地址跳跃的间隔的，甚至在一定程度保持读取数据是连续或者说相邻的，如此大大提高了 cache 的访存效率和命中率，从而使得卷积函数的运行能够在缓存方面得到速度的提高和性能优化。

优化版本的卷积函数代码如下：

```
float *tmp_pad;
//padding
int size_of_tmp1;
size_of_tmp1 = (w + 2*pad) * (h + 2*pad) * c;
tmp_pad = (float*)malloc(size_of_tmp1*sizeof(float));
for (k=0;k<c;k++)
for(j=0;j<h + 2*pad; j++)
        for(i=0;i<w + 2*pad; i++)
        {
            if(i<pad || j<pad || i>= (w + pad) || j >= (h + pad))
                tmp_pad[i + j*(w + 2*pad) + k * (w + 2*pad)*(h + 2*pad)] = 0.0;
                else
                  tmp_pad[i + j*(w + 2*pad) + k * (w + 2*pad)*(h + 2*pad)] = input[(i-pad) + (j-pad) * w + k * w* h];
            }
    //convolution
    int padding_size_w = w + 2*pad;
    int padding_size_h = h + 2*pad;
    int padding_size_c = c;

    int conv_size_w = (padding_size_w - kernel_size) / stride + 1;
    int conv_size_h = (padding_size_h - kernel_size) / stride + 1;
    int conv_size_c = n;

    float conv_tmp;
float *tmp_pad_sort;
    tmp_pad_sort = (float*)malloc(size_of_tmp1*sizeof(float));
    for(i=0; i<padding_size_w; i++)
    {
        for(j=0; j<padding_size_h; j++)
        {
            for(k=0; k<padding_size_c; k++)
                tmp_pad_sort[k + j*padding_size_c + i*padding_size_w*padding_size_c]=tmp_pad[i + j*padding_size_w + k*padding_size_h*padding_size_w];
    }
    }
    free(tmp_pad);

   for(k=0; k<conv_size_c; k++)
       for(i=0; i<conv_size_w; i++)
           for(j=0; j<conv_size_h; j++)
           {
               conv_tmp = 0;
               for(c_i=0; c_i<kernel_size; c_i++)
                   for(c_j=0; c_j<kernel_size; c_j++)
                       for (c_k=0; c_k<padding_size_c; c_k++)
                       {
                           conv_tmp += tmp_pad_sort[c_k + (c_j*padding_size_c + j*stride*padding_size_c) + \
                                       (i*stride*padding_size_c*padding_size_w + c_i*padding_size_c*padding_size_w)] *
                                       weight[c_i + c_j*kernel_size + c_k*kernel_size*kernel_size + k*kernel_size*kernel_size*padding_size_c];

                        /*conv_tmp += tmp_pad[(i*stride + c_i) + (j*stride*padding_size_w + c_j*padding_size_w) + (c_k*padding_size_w*padding_size_h)] *
                                       weight[c_i + c_j*kernel_size + c_k*kernel_size*kernel_size + k*kernel_size*kernel_size*padding_size_c];*/

                       }
               output[i + j*conv_size_w + k*conv_size_h*conv_size_w] = conv_tmp;
           }
   free(tmp_pad_sort);
}
```

然后使用 python 脚本进行一键编译，编译完成后使用 qemu 运行，得到预测结果和执行时间的反馈，执行时间为 39.894828s。

```
# loasic @ ubuntu in ~/experiments/lab10-d1/yolo_test_static_new [10:26:54]
$ qemu-riscv64 -cpu c906fdv yolo_test_riscv_qemu_cache-opt
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFL
OPs
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32  0.399 BFL
OPs
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFL
OPs
```

```
Net 19 (none) forwarding...     Done.
Net 20 (route) forwarding...    Done.
Net 21 (convolutional) forwarding...
Function: convolutional_compute
cycles: 13395656516
insts retire: 13395657047
        Done.
Net 22 (convolutional) forwarding...
Function: convolutional_compute
cycles: 1030125862
insts retire: 1030125816
        Done.
Net 23 (yolo) forwarding...     Done.
data/dog.jpg: Predicted in 39.894828 seconds.
dog: 57%
car: 52%
truck: 56%
car: 62%
bicycle: 59%
```

4) 在 D1 开发板上进行三个版本的 YOLO 性能测试，并攥写分析报告(关于 GEMM 库版本的分析，同学们需要额外查找资料，重点放在 img2col 算法上。下面表格中统计的数据除了“执行时间”，其他的数据可以使用计算量最大的一个层 conv12 来代表。)

Base:

```
Net 12 (convolutional) forwarding...
Function: convolutional_compute
cycles: 48689683039
insts retire: 8136282930
L1 Icache access: 9568652984
L1 Icache access begin: 99506313083
L1 Icache miss: 3983849
L1 Dcache read access: 1636181581
L1 Dcache read miss: 270606642
L1 Dcache write access: 27148150
L1 Dcache write miss: 1044213
        Done.
```

```
L1 Dcache write miss: 57051
        Done.
Net 23 (yolo) forwarding...     Done.
data/dog.jpg: Predicted in 140.631002 seconds.
dog: 57%
car: 52%
truck: 56%
```

Cache-opt

```
Net 12 (convolutional) forwarding...
Function: convolutional_compute
cycles: 13939036196
insts retire: 3438232400
L1 Icache access: 3736548222
L1 Icache access begin: 202269079317
L1 Icache miss: 1511703
L1 Dcache read access: 1018157644
L1 Dcache read miss: 68762903
L1 Dcache write access: 17049787
L1 Dcache write miss: 488899
```

```
L1 Dcache write access: 1000664
L1 Dcache write miss: 40334
        Done.
Net 23 (yolo) forwarding...     Done.
data/dog.jpg: Predicted in 44.866882 seconds.
dog: 57%
car: 52%
truck: 56%
```

Gemm:

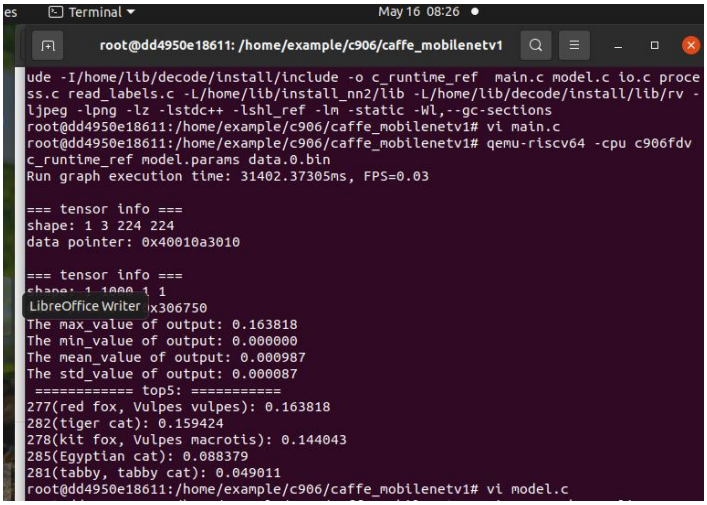
```
Net 11 (maxpool) forwarding...  Done.
Net 12 (convolutional) forwarding...
Function: convolutional_compute
cycles: 8092752985
insts retire: 5652094599
L1 Icache access: 5756173782
L1 Icache miss: 679783
L1 Dcache read access: 1607079211
L1 Dcache read miss: 2708513
L1 Dcache write access: 803343801
L1 Dcache write miss: 264617
        Done.
Net 13 (convolutional) forwarding...
```

```
        Done.
Net 23 (yolo) forwarding...     Done.
data/dog.jpg: Predicted in 30.834883 seconds.
dog: 57%
car: 52%
truck: 56%
car: 62%
bicycle: 59%
```

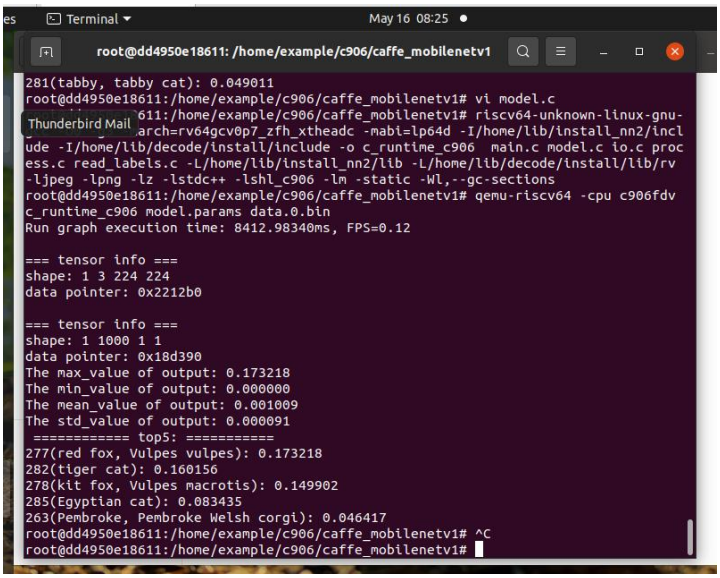
5) 使用平头哥HBB流程编译mobilenet生成两个版本的可执行文件(一个带向量扩展，一个不带扩展)，并分别在qemu和D1开发板运行。另行选取三张图片重复上述过程，并将实验结果统计在如下表格中。

图片 1：

Qemu:ref



Qemu:C906



D1:C906

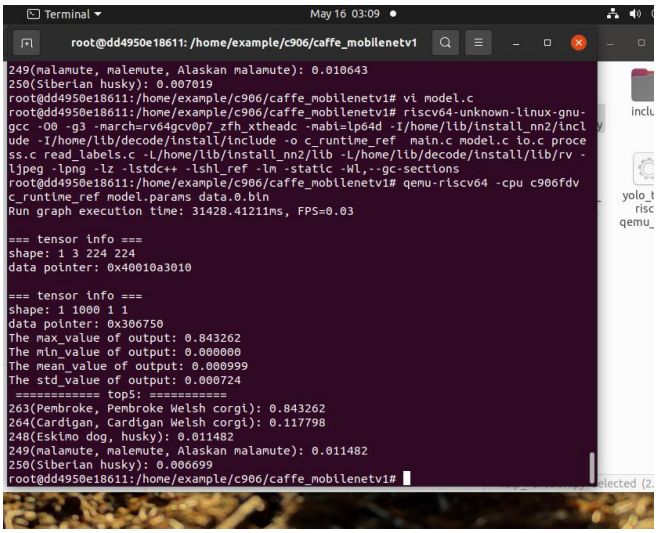
```
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_c906
Please set valide args: ./model.elf model.params [tensor1/image1 ...] [tensor2/image2
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_c906 model.params data.0.bin
Run graph execution time: 37725.26172ms, FPS=0.03

=== tensor info ===
shape: 1 3 224 224
data pointer: 0x3fc8732010

=== tensor info ===
shape: 1 1000 1 1
data pointer: 0x3ef64d30
The max_value of output: 0.163818
The min_value of output: 0.000000
The mean_value of output: 0.000987
The std_value of output: 0.000087
 ============ top5: ===========
277(red fox, Vulpes vulpes): 0.163818
282(tiger cat): 0.159424
278(kit fox, Vulpes macrotis): 0.144043
285(Egyptian cat): 0.088379
281(tabby, tabby cat): 0.049011
root@TinaLinux:/caffe_mobilenetv1#
```

D1:ref

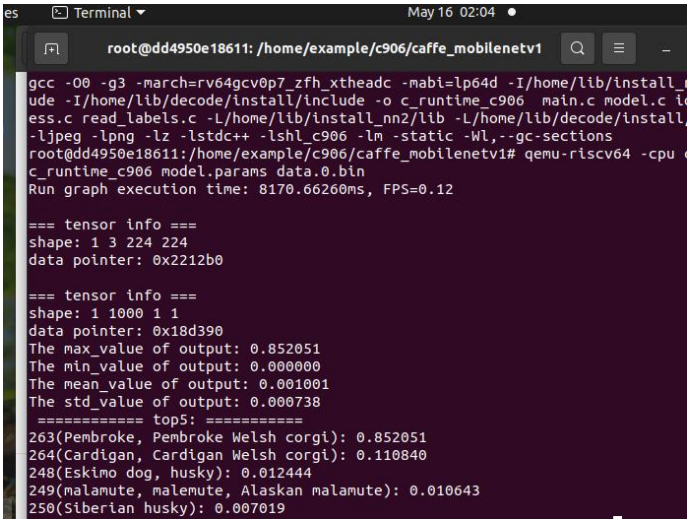
```
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_ref model.params data.0.bin
Run graph execution time: 41196.75781ms, FPS=0.02

=== tensor info ===
shape: 1 3 224 224
data pointer: 0x3fecb20010

=== tensor info ===
shape: 1 1000 1 1
data pointer: 0x273c9c00
The max_value of output: 0.163818
The min_value of output: 0.000000
The mean_value of output: 0.000987
The std_value of output: 0.000087
 ============ top5: ===========
277(red fox, Vulpes vulpes): 0.163818
282(tiger cat): 0.159424
278(kit fox, Vulpes macrotis): 0.144043
285(Egyptian cat): 0.088379
281(tabby, tabby cat): 0.049011
```

图片 2：

Qemu:ref：



Qemu:C906



D1 C906

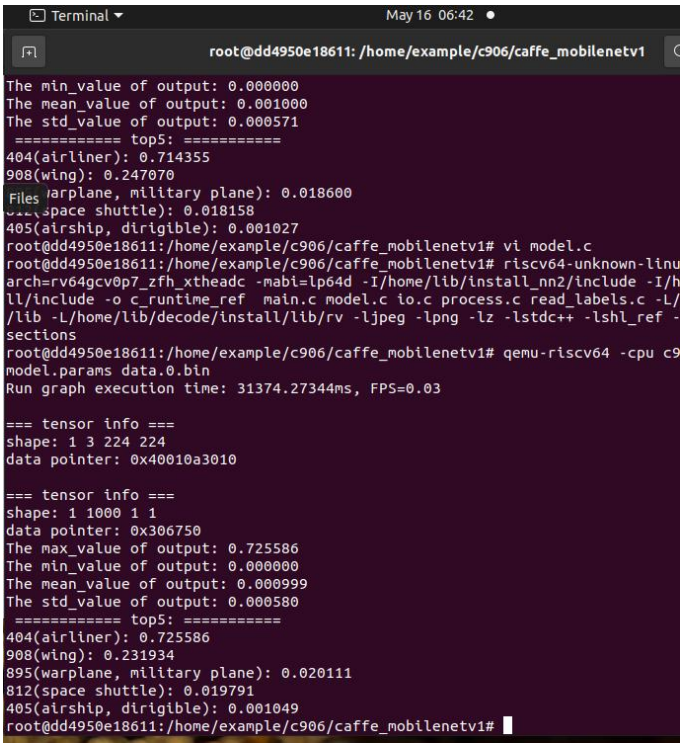
```
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_c906 model.params data.0.bin
Run graph execution time: 357.26550ms, FPS=2.80

=== tensor info ===
shape: 1 3 224 224
data pointer: 0x859a760

=== tensor info ===
shape: 1 1000 1 1
data pointer: 0x8506840
The max_value of output: 0.852539
The min_value of output: 0.000000
The mean_value of output: 0.001000
The std_value of output: 0.000738
 ============ top5: ===========
263(Pembroke, Pembroke Welsh corgi): 0.852539
264(Cardigan, Cardigan Welsh corgi): 0.110107
248(Eskimo dog, husky): 0.012070
249(malamute, malemute, Alaskan malamute): 0.010483
250(Siberian husky): 0.006916
```

d1 ref:

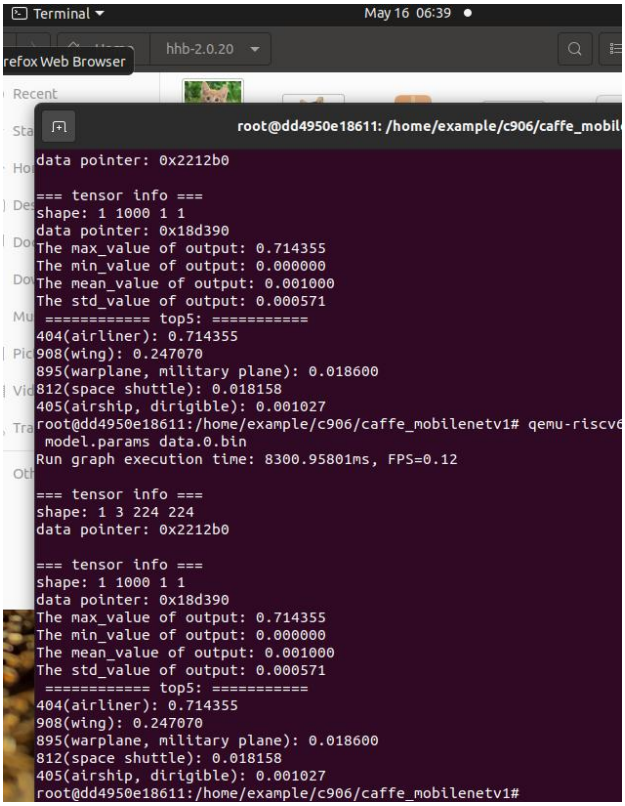
```
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_ref model.params data.0.bin
Run graph execution time: 41371.75391ms, FPS=0.02

=== tensor info ===
shape: 1 3 224 224
data pointer: 0x3fcb933010

=== tensor info ===
shape: 1 1000 1 1
data pointer: 0x11dcc00
The max_value of output: 0.843262
The min_value of output: 0.000000
The mean_value of output: 0.000999
The std_value of output: 0.000724
 ============ top5: ===========
263(Pembroke, Pembroke Welsh corgi): 0.843262
264(Cardigan, Cardigan Welsh corgi): 0.117798
248(Eskimo dog, husky): 0.011482
249(malamute, malemute, Alaskan malamute): 0.011482
250(Siberian husky): 0.006699
root@TinaLinux:/caffe_mobilenetv1#
```

图片 3：

Qemu:ref



Qemu:C906



D1 C906

```
root@TinaLinux:/caffe_mobilenetv1# chmod +x c_runtime_c906
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_c906 model.params data.0.bin
Run graph execution time: 357.71790ms, FPS=2.80

=== tensor info ===
shape: 1 3 224 224
data pointer: 0x47cf760

=== tensor info ===
shape: 1 1000 1 1
data pointer: 0x473b840
The max_value of output: 0.863281
The min_value of output: 0.000000
The mean_value of output: 0.001000
The std_value of output: 0.000756
============ top5: ===========
404(airliner): 0.863281
908(wing): 0.106384
895(warplane, military plane): 0.016312
812(space shuttle): 0.012222
405(airship, dirigible): 0.000996
root@TinaLinux:/caffe_mobilenetv1#
```

D1 ref

```
root@TinaLinux:/caffe_mobilenetv1# ./c_runtime_ref model.params data.0.bin
Run graph execution time: 41343.38672ms, FPS=0.02

=== tensor info ===
shape: 1 3 224 224
data pointer: 0x3fce719010

=== tensor info ===
shape: 1 1000 1 1
data pointer: 0x315a7c00
The max_value of output: 0.857422
The min_value of output: 0.000000
The mean_value of output: 0.001000
The std_value of output: 0.000747
 ============ top5: ===========
404(airliner): 0.857422
908(wing): 0.110718
895(warplane, military plane): 0.015701
812(space shuttle): 0.014183
405(airship, dirigible): 0.001155
root@TinaLinux:/caffe_mobilenetv1#
```

D1:

表 1 在不同平台下不同输入的执行时间

	qemu_ref	qemu_c906	D1_ref	D1_c906
图片 1	31008.34375ms	8412.98340ms	41196.75781ms	37725.26172ms
图片 2	31428.41211ms	8170.66260ms	41371.75291ms	357.26650ms
图片 3	31374.27344ms	8300.95801ms	41343.38672ms	357.71790ms

表 2 在不同平台下不同输入的分类结果

	qemu_ref	qemu_c906	D1_ref	D1_c906
图片 1	Red fox, vulpes vulpes	Red fox, vulpes vulpes	Red fox, vulpes vulpes	Red fox, vulpes vulpes
图片 2	Pembroke, Pembroke Welsh corgi	Pembroke, Pembroke Welsh corgi	Pembroke, Pembroke Welsh corgi	Pembroke, Pembroke Welsh corgi
图片 3	airliner	airliner	airliner	airliner

3、实验分析和总结

熟悉了 RISC-V 架构的硬件平台，并在之上启动一个 Linux 操作系统。然后在此之上运行 YOLO 算法，之后对卷积层进行了优化，并对优化的性能进行分析，在这个过程把书中学习到的缓存原理知识应用到实际的工程开发中，并真切地感受到了其发挥的作用。最后通过 HHB 流程对 RISC-V 的向量扩展的应用有一个初步的认识。

使用 qemu 和 D1 开发板运行 yolo 算法的不同卷积函数版本时我们发现，qemu 与 D1 开发板的一个是通过软件模拟 riscv 架构，一个是真实的 riscv 的硬件架构 cache-opt 的缓存优化版本的 yolo 算法在 qemu 上运行完成分类预测需要 40s 左右，而在 D1 开发板上完成分类预测则需要 25s 左右，这说明了 qemu 的对 riscv 处理器架构的软件模拟和真实的 riscv 处理器是有着实际性能差别的存在。通过实验数据可以发现，两个优化版本的卷积函数通过

减少有效指令数量来减少处理器执行的周期数，同时通过减少缓存访存的 miss 数来提高缓存命中率，从而有效减少整个 yolo 算法的执行时间。

编写的 cache-opt 的缓存优化版本的优化思路如下：通过数据重排的方法，将实际储存图像数据的一位数组不再以原有的 w、h、c 的顺序串联排列，而是掉换成 c、h、w 的顺序排列，相应的的，在进行卷积计算时也不再是原来的 w*h*c 的遍历顺序去求乘积，而是 c*h*w 的顺序，这样一来就可以利用岛通道数 c 的移动上不存在步长，且这样的卷积和数据顺序可以大大缩小卷积计算时读取一维数组当中的图像数据地址跳跃的间隔的，甚至在一定程度保持读取数据是连续或者说相邻的，如此大大提高了 cache 的访存效率和命中率，从而使得卷积函数的运行能够在缓存方面得到速度的提高和性能优化。

使用 HHB 对 MobileNet 算法进行算子向量扩展算子的转换能够有效提高分类算法的运行效率，对于根据数据表格我们很清晰的看到，对于执行时间来说，支持向量扩展的算子库即 c906 版本的执行速度相对于不支持向量扩展的算子库即 ref 版本得到了很大的提高。

4、实验收获、存在问题、改进措施或建议等

通过此次实验我们熟悉一个 RISC-V 架构的硬件平台，以及在其之上启动的 Linux 操作系统。然后我们通过在此之上运行 YOLO 算法，编写了在缓存角度对卷积层进行优化的卷积函数，同时使用嵌入的汇编代码进行性能监测，对优化的性能对比 base 未作优化的版本和使用了 img2col 算法优化的 gemm 版本进行分析，在这个过程中我们把书中学习到的缓存原理知识应用到实际的工程开发中，并且真切地感受到其发挥的对提高运行效率重要作用。