

## 将开源框架 darknet 移植到 fpga 上

目前有几个基于 OpenCL 的深度学习框架，但其实不是很好用，能在 fpga 上跑的就更少了。感觉最好且最直接的办法就是将当前主流深度学习框架进行源码修改使其支持 fpga。这里尝试对开源框架 darknet 修改源码，使其能在曙光的 fpga 卡上跑。目前可实现在 darknet 中调用 fpga 进行矩阵运算，但应该是 kernel 函数写的有点问题，最后的检测结果有点问题。这里对主要的实现过程进行说明，具体的可参考曙光机子里 darknet 中的源码。该实现方法是基于 darknet 框架的，对 caffe 应该也是适用的。

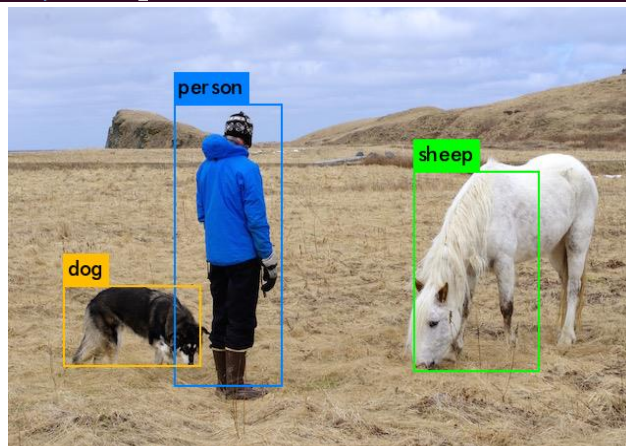
### 1 开源框架 darknet

Darknet 是一个轻量且易于部署的主流开源框架，可以很方便的实现目标检测、图像分类等任务。与 caffe 类似，darknet 是个纯 c 语言的深度学习框架。这个框架开始上手较难，但熟悉以后用起来很方便，因为是 c 语言写的，所以源码修改很方便。

### 2 Darknet 实现目标检测的具体例子

Darknet 支持 cpu 和 gpu，这里首先运行一个在 cpu 上实现目标检测的例子。安装 darknet 后，用 tiny-yolo 算法实现的目标检测如下图。输入测试命令后 darknet 运行检测网络，检测图片为 person.jpg，然后输出图片中各目标类别准确度和检测时间：sheep, 60%，person, 73%，dog, 53%，时间，2.88s。检测得到的图片如下所示，检测出的目标用包围框标记出来。

```
zimo@zimo: ~/darknet
zimo@zimo:~$ cd darknet/
zimo@zimo:~/darknet$ ./darknet detector test cfg/voc.data cfg/yolov2-tiny-voc.cfg ./tiny-yolo-voc.weights ./data/person.jpg
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
5 max     2 x 2 / 2  104 x 104 x 64  -> 52 x 52 x 64
6 conv  128        3 x 3 / 1  52 x 52 x 64  -> 52 x 52 x 128  0.399 BFL
OPs
7 max     2 x 2 / 2  52 x 52 x 128  -> 26 x 26 x 128
8 conv  256        3 x 3 / 1  26 x 26 x 128  -> 26 x 26 x 256  0.399 BFL
OPs
9 max     2 x 2 / 2  26 x 26 x 256  -> 13 x 13 x 256
10 conv  512        3 x 3 / 1  13 x 13 x 256  -> 13 x 13 x 512  0.399 BFL
OPs
11 max    2 x 2 / 1  13 x 13 x 512  -> 13 x 13 x 512
12 conv 1024        3 x 3 / 1  13 x 13 x 512  -> 13 x 13 x 1024 1.595 BFL
OPs
13 conv 1024        3 x 3 / 1  13 x 13 x 1024 -> 13 x 13 x 1024 3.190 BFL
OPs
14 conv  125        1 x 1 / 1  13 x 13 x 1024 -> 13 x 13 x 125  0.043 BFL
OPs
15 detection
mask scale: Using default '1.000000'
Loading weights from ./tiny-yolo-voc.weights...Done!
./data/person.jpg: Predicted in 2.883106 seconds.
sheep: 60%
person: 73%
dog: 53%
zimo@zimo:~/darknet$
```



### 3 darknet 源码分析

```
~/darknet/src/convolutional_layer.c - Sublime Text (UNREGISTERED)
gemm_nn.c x convolutional_layer.c x
443 }
444
445 void forward_convolutional_layer(convolutional_layer l, network net)
446 {
447     int i, j;
448     fill_cpu(l.outputs*l.batch, 0, l.output, 1);
449
450     if(l.xnor){
451         binarize_weights(l.weights, l.n, l.c/l.groups*l.size*l.size, l.binary_weights);
452         swap_binary(&l);
453         binarize_cpu(net.input, l.c*l.h*l.w*l.batch, l.binary_input);
454         net.input = l.binary_input;
455     }
456
457     int m = l.n/l.groups;
458     int k = l.size*l.size*l.c/l.groups;
459     int n = l.out_w*l.out_h;
460     for(i = 0; i < l.batch; ++i){
461         for(j = 0; j < l.groups; ++j){
462             float *a = l.weights + j*l.nweights/l.groups;
463             float *b = net.workspace;
464             float *c = l.output + (i*l.groups + j)*n*m;
465
466             im2col_cpu(net.input + (i*l.groups + j)*l.c/l.groups*l.h*l.w,
467                 l.c/l.groups, l.h, l.w, l.size, l.stride, l.pad, b);
468             gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);
469         }
470     }
471
472     if(l.batch_normalize){
473         forward_batchnorm_layer(l, net);
474     } else {
475         add_bias(l.output, l.biases, l.batch, l.n, l.out_h*l.out_w);
476     }
477
478     activate_array(l.output, l.outputs*l.batch, l.activation);
479     if(l.binary || l.xnor) swap_binary(&l);
480 }
481 }
```

在 cnn 中有卷积层、池化层、全连接层等等，其中整个 cnn 的计算主要集中在卷积层，所以这里选择对卷积层进行修改。darknet 原版本支持 cpu 和 gpu，其卷积层实现代码主要在 src/convolutional\_layer.c 中。我们在 fpga 上跑深度学习算法时其实是运行预训练模型，即首先在 GPU 上将网络运行好，然后将训练好的网络（预训练网络）在 fpga 上运行，称为网络的前向推理 inference。在运行预训练模型时卷积层所调用函数主要是上图所示 forward\_convolutional\_layer()，该函数负责完成卷积层中矩阵运算，即输入特征矩阵 A\*权重矩阵 B 得到输出特征矩阵 C。进一步的，该函数中 gemm() 函数主要负责上述矩阵运算，我们继续

```
~/darknet/src/gemm.c - Sublime Text (UNREGISTERED)
gemm_nn.c x convolutional_layer.c x gemm.c x
65 void gemm(int TA, int TB, int M, int N, int K, float ALPHA,
66     float *A, int lda,
67     float *B, int ldb,
68     float BETA,
69     float *C, int ldc)
70 {
71     gemm_cpu( TA, TB, M, N, K, ALPHA,A,lda, B, ldb,BETA,C,ldc);
72 }
73
74 void gemm_nn(int M, int N, int K, float ALPHA,
75     float *A, int lda,
76     float *B, int ldb,
77     float *C, int ldc)
78 {
79     int i,j,k;
80     #pragma omp parallel for
81     for(i = 0; i < M; ++i){
82         for(k = 0; k < K; ++k){
83             register float A_PART = ALPHA*A[i*lda+k];
84             for(j = 0; j < N; ++j){
85                 C[i*ldc+j] += A_PART*B[k*ldb+j];
86             }
87         }
88     }
89 }
90
91 void gemm_nt(int M, int N, int K, float ALPHA,
92 {
93 }
94
95 void gemm_tn(int M, int N, int K, float ALPHA,
96 {
97 }
98
99 void gemm_tt(int M, int N, int K, float ALPHA,
100 {
101 }
102
103 void gemm_cpu(int TA, int TB, int M, int N, int K, float ALPHA,
104     float *A, int lda,
105     float *B, int ldb,
106     float BETA,
107     float *C, int ldc)
108 {
109     //printf("cpu: %d %d %d %d %d %f %d %f %d\n",TA, TB, M, N, K, ALPHA, lda, ldb, BETA, ldc);
110     int i, j;
111     for(i = 0; i < M; ++i){
112         for(j = 0; j < N; ++j){
113             C[i*ldc+j] *= BETA;
114         }
115     }
116     if(!TA && !TB)
117         gemm_nn(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
118     else if(TA && !TB)
119         gemm_tn(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
120     else if(!TA && TB)
121         gemm_nt(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
122     else
123         gemm_tt(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
124 }
```

看源码中 `gemm()` 函数的实现，`gemm()` 函数在 `gemm.c` 中，其源码如下所示。处 `gemm()` 函数外，该 `gemm.c` 还包含 `gemm_cpu()` 以及 `gemm_nn()`, `gemm_nt()`, `gemm_tn()`, `gemm_tt()` 等几个函数。`gemm()` 函数是一个总的接口，其各参数具体指：

TA-矩阵 A 是否转置，TB-矩阵 B 是否转置，M-矩阵 A 行数，N-矩阵 B 列数，K-矩阵 B 列数，\*A-input matrix A, \*B-input matrix B, float \*C-output matrix C, lda-columns of A, ldb-columns of B, ldc-columns of C.

`gemm()` 函数实现  $A(M, K) * B(K, N) = C(M, N)$  的运算。可以看出 `gemm()` 函数继续调用了 `gemm_cpu()`，而下面的 `gemm_cpu()` 函数则继续调用 `gemm_nn`, `gemm_tn`, `gemm_nt`, `gemm_tt` 四个函数，darknet 中默认不对矩阵进行转置，因此使用的是 `gemm_nn` 函数。每次卷积层运行时都会调用 `gemm_nn` 函数进行矩阵运算，所以这是卷积层矩阵运算的核心函数。移植到 fpga 上时，将该 `gemm_nn` 函数负责的运算部分用在 fpga 上实现的 `gemm_nn_opengl.cl` 核函数实现，让 fpga 负责运行这部分运算。

#### 4 源码修改

对源码进行分析后，开始对源码进行修改。

如下图所示，首先在最开始的 `convolutional_layer.c` 中将原 `gemm()` 函数用 `gemm_fpga()` 替换，接着在 `gemm.c` 中添加并具体实现 `gemm_fpga()`。

```
505
506     if (l.size == 1) {
507         b = im;
508     } else {
509         im2col_cpu(im, l.c/l.groups, l.h, l.w, l.size, l.stride, l.pad, b);
510     }
511     //gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);
512     gemm_fpga(0,0,m,n,k,1,a,k,b,n,1,c,n);
513 }
514
515
```

如上图所示，在 `gemm.c` 中添加两个接口，`gemm_fpga()` 和 `gemm_nn_fpga()`。`gemm_fpga()` 中所有参数与 `gemm()` 相同，接着 `gemm_fpga()` 调用写好的 `gemm_nn_fpga()`，而 `gemm_nn_fpga()` 则继续调用 `gemm_run()`，`gemm_run()` 用 `opencl` 编写，替代 `gemm_nn()` 并调用 `fpga` 实现矩阵运算部分。

#### 4.1 `gemm_run()` 函数

`gemm_run()` 函数负责在 `fpga` 上实现  $A*B=C$  的矩阵运算，这个函数用 `opencl` 实现。将 altera 官方的矩阵相乘例子进行修改实现 `gemm_run` 函数。如下图所示，将原 `host` 程序修改，得到 `gemm_fpga.h` 和 `gemm_fpga.cpp`，其主要包含三个接口，`gemm_init()`、`gemm_run()` 和 `gemm_cleanup()`。`gemm_init()` 负责 `opencl` 的一些初始化，`gemm_cleanup()` 负责资源的释放，`gemm_run()` 主要负责矩阵运算。

三个函数的具体实现较繁琐，有机会再细说，可具体参考已经写好的 `gemm_fpga.cpp`，这里只对 `gemm_run()` 函数中部分代码进行说明，如图所示，这部分代码实现参数传递，即将 `gemm_run()` 的参数传递到 `kernel` 函数中，其中参数顺序不能乱，包括 `A` 的行数，`B` 的列数，`A` 的列数，`ALPHA` 参数，输入矩阵 `A`，`lda`，输入矩阵 `B`，`ldb`，输出矩阵 `C`，`ldc`。

```
117 // Function prototypes
118
119 bool gemm_init();
120 void gemm_run(int M, int N, int K, float ALPHA, float *A, int a, float *B, int b, float *C, int c);
121 //M:A,C rows.
122 //N:B,C columns.
123 //K:A columns, B rows.
124 //void run();
125 void gemm_cleanup();
126
127
128 // Initializes the OpenCL objects.
129 bool gemm_init() {
130 }
131
132 // Initialize the data for the problem. Requires num_devices to be known.
133
134 void gemm_run(int M, int N, int K, float ALPHA, float *A, int lda, float *B, int ldb, float *C, int ldc) {
135 }
136
137 // Free the resources allocated during initialization
138 void gemm_cleanup() {
139 }
140
```

这里 `gemm_init()` 和 `gemm_cleanup()` 函数放在主函数 `darkent.c` 中，而 `gemm_run()` 函数放在 `gemm.c` 中的 `gemm_nn_fpga()` 中，这样 `darkent` 执行时就可只执行一次初始化和释放过程而执行多次 `kernel` 运算的过程。

```
status = clSetKernelArg(kernel[i], argi++, sizeof(A_height), &A_height);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(B_width), &B_width);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(A_width), &A_width);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(ALPHA), &ALPHA);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &input_a_buf[i]);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(lda), &lda);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &input_b_buf[i]);
checkError(status, "Failed to set argument %d", argi - 1);

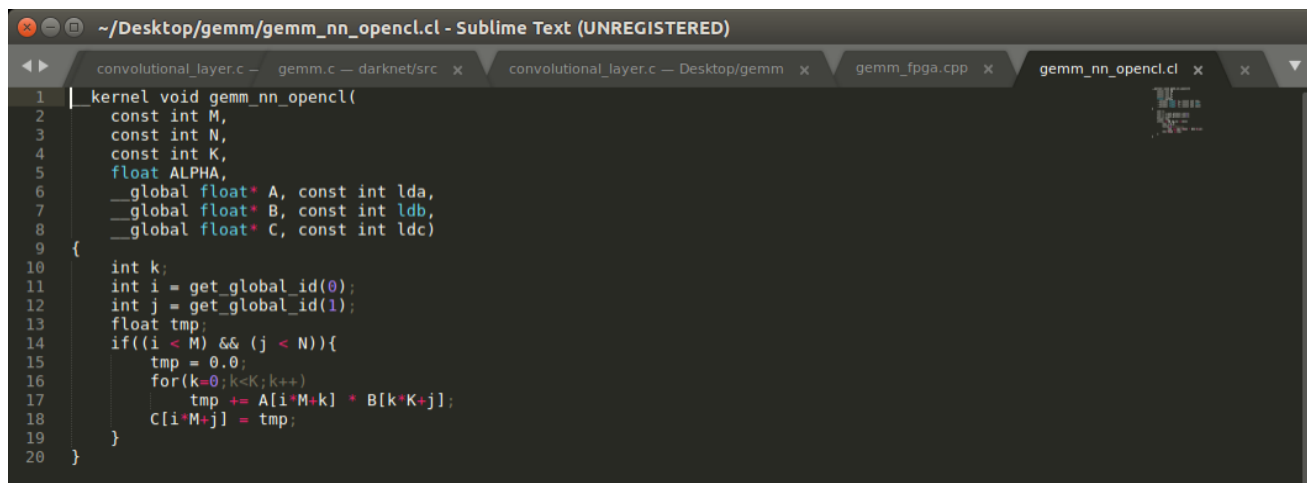
status = clSetKernelArg(kernel[i], argi++, sizeof(ldb), &ldb);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &output_buf[i]);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(ldc), &ldc);
checkError(status, "Failed to set argument %d", argi - 1);
```

## 4.2 kernel 函数

这里 kernel 函数的实现方法是按照书上的一种较为简单的实现方式，直接将 `gemm_nn()` 中 for 循环部分转换过来。这种实现方式比较简单，根据后面的测试，这个 kernel 函数可能写的有点问题。kernel 函数写好后编译生成 aocx 文件。



```
1 kernel void gemm_nn_opengl(  
2     const int M,  
3     const int N,  
4     const int K,  
5     float ALPHA,  
6     __global float* A, const int lda,  
7     __global float* B, const int ldb,  
8     __global float* C, const int ldc)  
9 {  
10     int k;  
11     int i = get_global_id(0);  
12     int j = get_global_id(1);  
13     float tmp;  
14     if((i < M) && (j < N)){  
15         tmp = 0.0;  
16         for(k=0; k<K; k++){  
17             tmp += A[i*M+k] * B[k*N+j];  
18             C[i*N+j] = tmp;  
19         }  
20     }
```

## 5 编译，修改 Makefile

将写好的 `gemm_fpga.cpp` 和其相应头文件 `gemm_fpga.h` 放在 `darknet/src` 目录下，将 kernel 的 aocx 文件放在 `darknet` 目录下。为了能将写好的程序加入到 `darknet` 中，需要进一步修改原 Makefile 文件，如下所示。

首先让 Makefile 查找 `opencl` 库文件的路径，并用两个变量表示：

```
38  
39 ifeq ($(wildcard $(ALTERAOCLSDKROOT)),)  
40 $(error Set ALTERAOCLSDKROOT to the root directory of the Intel(R) FPGA SDK for OpenCL(TM) so  
41 endif  
42 ifeq ($(wildcard $(ALTERAOCLSDKROOT)/host/include/CL/opencl.h),)  
43 $(error Set ALTERAOCLSDKROOT to the root directory of the Intel(R) FPGA SDK for OpenCL(TM) so  
44 endif  
45  
46 AOCL_COMPILE_CONFIG := $(shell aocl compile-config )  
47 AOCL_LINK_CONFIG := $(shell aocl link-config )  
48
```

然后在 `OBJ` 这一栏中添加 `gemm_fpga.o`，使得编译时将之前写的 `gemm_fpga.cpp` 编译成生成可执行文件，

```
79 OBJ=gemm_fpga.o opencl.o options.o fpga.o gemm.o utils.o cuda.o deconvolutional_layer.o convolutional_l  
80 EXECOBJ=captcha.o lsd.o super.o art.o tag.o cifar.o go.o rnn.o segmenter.o regressor.o classifier.o cc  
81 #EXECOBJ+=volo.o detector.o darknet.o classifier.o
```

并且额外添加一行：

```
$(OBJDIR).o: %.cpp $(DEPS) $(CC) $(COMMON) $(CPPFLAGS) $(CXXFLAGS) $(AOCL_COMPILE_CONFIG)  
-c $< -o $@ $(AOCL_LINK_CONFIG) $(foreach L, $(LIBS), -l$L),
```

这行代码负责将 `gemm_fpga.cpp` 编译成 `gemm_fpga.o`，因为原 Makefile 中 `.o:.c` 这个命令只能完成 `.c` 文件的编译，添加上述代码后可完成 `.cpp` 文件的编译。另外 Makefile 中编译器要由之前的 `gcc` 换成 `g++`，因为 `opencl` 是基于 `g++` 的，`darknet` 是基于 `gcc` 的，这里选择 `g++` 作为总的编译器。

最后在编译时需要添加之前的 `opencl` 库文件路径，库文件变量的位置顺序不能有错。

```
95  
96 $(EXEC): $(EXECOBJ) $(ALIB)  
97 $(CC) $(COMMON) $(CXXFLAGS) $(AOCL_COMPILE_CONFIG) $^ -o $@ $(LDFLAGS) $(ALIB) $(AOCL_LINK_CONFIG)  
98  
99 $(ALIB): $(OBJJS)  
100 $(AR) $(ARFLAGS) $@ $^  
101 #$(CC) $(AOCL_COMPILE_CONFIG) $(AOCL_LINK_CONFIG)  
102  
103 $(SLIB): $(OBJJS)  
104 $(CC) $(CPPFLAGS) $(CXXFLAGS) $(AOCL_COMPILE_CONFIG) -shared $^ -o $@ $(LDFLAGS) $(AOCL_LINK_CONFIG)  
105  
106 $(OBJDIR).o: %.c $(DEPS)  
107 $(CC) $(COMMON) $(CPPFLAGS) $(CXXFLAGS) $(AOCL_COMPILE_CONFIG) -c $< -o $@ $(AOCL_LINK_CONFIG) $(foreach L, $(LIBS), -l$L)  
108 $(OBJDIR).o: %.cpp $(DEPS)  
109 $(CC) $(COMMON) $(CPPFLAGS) $(CXXFLAGS) $(AOCL_COMPILE_CONFIG) -c $< -o $@ $(AOCL_LINK_CONFIG) $(foreach L, $(LIBS), -l$L)  
110
```



## 6 测试

测试结果如下，

输入测试命令后，可正常调用 fpga 并打印 fpga 相关信息，即 gemm\_init() 函数所实现的部分，接着 darknet 载入网络，开始运行网络，运行过程中每次运行卷积层时都输出一次信息，包括矩阵尺寸，时间、吞吐量等。

但是在执行到第八层左右出现 -nan 的错误，应该是运算值过大，运算出现错误，导致最后无法输出准确的目标信息，只输出执行时间。这个应该是 kernel 函数写的有问题，并且因为现在整个过程实现的还比较粗糙，所以执行时间目前还较慢。

总体来说修改源码的方式是可行的，进一步写个好点的 kernel 函数后应该能正常输出目标信息，实现目标检测或者其他深度学习任务。

包括一些头文件之类的实现这里没有说明，具体的实现细节可以进一步参考曙光机子里 /home/test/darknet 的源码。

```
root@046099:/home/test/darknet
[roo@046099 darknet]#
[roo@046099 darknet]#
[roo@046099 darknet]# ./darknet detector test cfg/voc.data cfg/yolov2-tiny-voc.
cfg ./tiny-yolo-voc.weights ./data/person.jpg
Initializing OpenCL
Platform: Intel(R) FPGA SDK for OpenCL(TM)
Using 1 device(s)
  fa510q : Arria 10 Reference Platform (acla10_ref0)
Using AOCC: gemm_nn_openccl.aocc
Reprogramming device [0] with handle 1
layer   filters  size  input          output
0 conv   16  3 x 3 / 1  228 x 228 x 3  -> 228 x 228 x 16  0.045 BFL Throughput: 0.28 GFLOPS
OPs
1 max    2  2 x 2 / 2  228 x 228 x 16  -> 114 x 114 x 16
2 conv   32  3 x 3 / 1  114 x 114 x 16  -> 114 x 114 x 32  0.120 BFL
OPs
3 max    2  2 x 2 / 2  114 x 114 x 32  -> 57 x 57 x 32
4 conv   64  3 x 3 / 1  57 x 57 x 32    -> 57 x 57 x 64  0.120 BFL
OPs
5 max    2  2 x 2 / 2  57 x 57 x 64    -> 28 x 28 x 64
6 conv  128  3 x 3 / 1  28 x 28 x 64    -> 28 x 28 x 128  0.116 BFL
OPs
7 max    2  2 x 2 / 2  28 x 28 x 128   -> 14 x 14 x 128
8 conv  256  3 x 3 / 1  14 x 14 x 128   -> 14 x 14 x 256  0.116 BFL
OPs
9 max    2  2 x 2 / 2  14 x 14 x 256   -> 7 x 7 x 256
10 conv 512  3 x 3 / 1  7 x 7 x 256     -> 7 x 7 x 512  0.116 BFL Throughput: 0.68 GFLOPS
OPs
11 max   2  2 x 2 / 1  7 x 7 x 512     -> 7 x 7 x 512
12 conv 1024 3 x 3 / 1  7 x 7 x 512     -> 7 x 7 x 1024  0.462 BFL
OPs
13 conv 1024 3 x 3 / 1  7 x 7 x 1024    -> 7 x 7 x 1024  0.925 BFL
OPs
14 conv 125 1 x 1 / 1  7 x 7 x 1024    -> 7 x 7 x 125  0.013 BFL
OPs
15 detection
mask_scale: Using default '1.000000'
Loading weights from ./tiny-yolo-voc.weights...Done!
Matrix sizes:
  A: 16 x 27
  B: 27 x 51984
  C: 16 x 51984
Generating input matrices
Launching for device 0 (global size: 16, 51984)

Time: 65.570 ms
Kernel time (device 0): 65.507 ms

Throughput: 0.68 GFLOPS

device:0
output_size:8
C_size:8
output1: -0.183877
Matrix sizes:
  A: 32 x 144
  C_size:8
output1: 492.317
Matrix sizes:
  A: 1024 x 9216
  B: 9216 x 49
  C: 1024 x 49
Generating input matrices
Launching for device 0 (global size: 1024, 49)

Time: 3348.849 ms
Kernel time (device 0): 3348.777 ms

device:0
output_size:8
C_size:8
output1: nan
Matrix sizes:
  A: 125 x 1024
  B: 1024 x 49
  C: 125 x 49
Generating input matrices
Launching for device 0 (global size: 125, 49)

Time: 18.476 ms
Kernel time (device 0): 18.414 ms

Throughput: 0.68 GFLOPS

device:0
output_size:8
C_size:8
output1: nan
./data/person.jpg: Predicted in 6.662858 seconds.
[roo@046099 darknet]#
```