关于实验细节主要是讲 FPGA 实现部分的东西，也就是实验计划 5 的详细内容

综述：

整个的实现分为几个大的步骤，（1）：对在 GPU 上训练好的官方的权重文件做 BN 层融合以及分离 WEIGHT 和 BIAS，然后经过动态量化成为定点数，再使用定点后的文件进行计算成新的逐层输出的结果，与原结果做对比计算出逐层的输出在误差最小的时候的量化位数。这一步一共获得五个文件，分别是：量化权重文件，量化权重位数文件，量化偏置文件，量化偏置位数文件，量化输入输出位数文件。（2）：对 YOLOv2 的 IP 核进行修改，修改成符合 YOLOv3 的 IP 核。这一步的目的是需要做成自定义的 IP 核，下一步在 Vivado 工程里进行使用。（3）：按照设计好的输入输出管理方式进行 Vivado 工程的编译，进行 Petalinux 工程的编译，烧写系统。

这三个步骤中（1）与其他没有明显的顺序关系，但是（2）与（3）一定要是先（2）再（3），也就是说，可以先完成（2）再完成（3）最后做（1），但是要运行起来一定是三个都要有的。

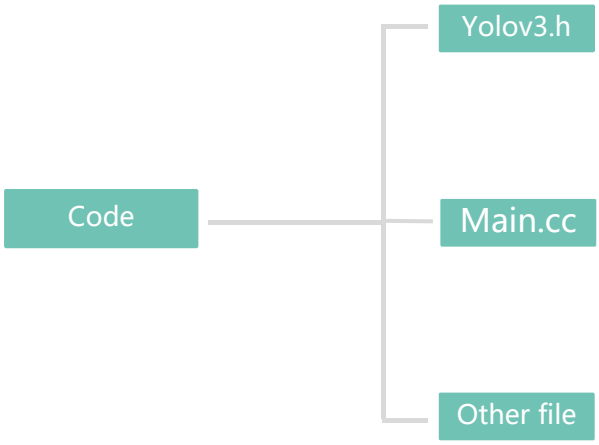
步骤一：

步骤一分为几个小的点，首先是 BN 层融合与权重偏置分离。为什么要这样是为了减少计算量，由于权重文件是本身就固定好的，那么 BN 层就可以在这一步中直接映射，与训练时不同。而权重与偏置分离则是为了后面 FPGA 的加载方便，由于权重每层是在计算完成后再加载的，因此选择分离开，这一步完成后会获得两个文件。更具体的细节可以参考之前的复现实验总结与 YOLO2 工程里的 README。

由于 YOLO3 与 YOLO2 的 BN 层方式相同，因此这一步不需要进行改动，也没有遇到什么问题。【注意：由于原作者在写入的时候是 W+，因此多次运行会使得文件大小成倍增加影响后面的结果，要么改成 W 要么就运行获得文件后删除】

接下来是对上一步获得好的权重与偏置进行量化并获得误差最小时的量化位数。

这一步骤中，代码的结构大致是下面的样子：



其中 Other file 包括了之前运行获得的两个文件，以及读写文件驱动的文件（这两个不需要改动，都是 darknet 做好的），还有一些 Label，test file 等等，其中，需要改动的是 yolov2.cfg 文件，也就是网络配置信息文件。

而 Main.cc 除了改一些文件名信息以外，别的也不需要改动多少。

最主要的改动在 Yolov3.h 里，这个文件的大致组织如下：

YOLOv3.h{

Function darknet...{

大概有一堆是 YOLO 模型用到的 darknet 函数的一锅烩，只需要找到你所需要的，按照顺序，拷贝到这块，就可以了

}

Function calculate{

这部分是在 PC 上自己写的计算函数，也就是说，你的网络包括了什么层，在这里就要按照 FPGA 的方式实现，方便后面的调用。也可以先不写，先按 darknet 的贴到这里，实现一个完整的功能，熟悉以后再实现也可。

}

YOLOv2_FPGA{

这里有个 YOLOv2_FPGA 的函数，就是按照 FPGA 的 pingpong 方式来对上面的计算函数分层类型进行不同的调用。

}

YOLOv2_HLS{

这个部分是最终的主控部分，分为加载，内存控制，按层计算，结果返回。

}

}

其中，原作者又修改过一次文件的版本，现在 YOLOv2.h 只负责 darknet 的部分，其余部分在 YOLOv2_sim.h 里。

这一步首先需要对权重文件进行重组织，也就是按照 FPGA 的计算顺序来进行重新排列权重文件，然后才是量化。原作者选择的计算卷积层的方式是 caffe 框架中的 im2col 的方式，至于这种方式网上都有搜一搜，大致就是用空间换取计算更快。还有一些优化方式也可以选，FFT，winograd 等。FFT 主要在卷积层的卷积核大小超过 5 的时候才会有明显效果，而这些年最新的算法都是卷积核朝着更小的方向在走。Winograd 是 CVPR 的（好像是 16？）论文，大致就是通过理论可以证明，用更小的乘法次数就可以完成卷积运算（也就是矩阵乘法），这个方式是近几年比较新的，但是 Winograd 也有一些缺点，虽然算法很美好，但是实现的时候需要提前对权重做处理，这部分的实现似乎还没什么人做。而与 Winograd 对应的是对权重的定点数训练，这部分也是比较热的。

这里主要讲 YOLOv2_HLS 函数的一些情况

先把 YOLO2_FPGA 的一些参数解析贴在这，原作者一个注释都没给。。。实在是有点搞心态。

/*

float *Input0,float *Input1,float *Output,float *Weight,float *Beta,输入输出权值偏移数组地址

,const int InFM_num,const int OutFM_num, 输入的通道总数，输出通道总数，在 v2 代码里 reorg 的输入整个是按照一个 channel 处理的

const int Kernel_size,const int Kernel_stride,卷积核大小步长

const int Input_w,const int Input_h,const int Padding,const bool IsNL,const bool IsBN,输入长和宽，是否填充，是否 relu 是否批量正则化

const int TM,const int TN,const int TR,const int TC,计算好的参数

const int mLoops,const int nLoops,const int rLoops,const int cLoops,同上，也就是加速器的设计,const int LayerType 层类型
*/
void YOLO2_FPGA(float *Input0,float *Input1,float *Output,float *Weight,float *Beta,const int InFM_num,const int OutFM_num,
const int Kernel_size,const int Kernel_stride,
const int Input_w,const int Input_h,const int Padding,const bool IsNL,const bool IsBN,
const int TM,const int TN,const int TR,const int TC,
const int mLoops,const int nLoops,const int rLoops,const int cLoops,const int LayerType)//这是对 YOLOv3 修改好的函数原型

YOLOv2_HLS 主要就是利用 YOLO2_FPGA 函数，他负责计算参数，然后传给 YOLO2_FPGA 进行计算。
首先是 weight_offset 和 beta_offset 的获得，weight_offset 是在加载权重的时候输出的【在 darknet/src/parser.c/load_convolutional_weights 函数里输出 num 的值】，然后统计成数组。beta_offset 则是在网络加载的时候，会输出每一层的结点数目，也统计成数组。注意这两种都是只算卷积层的。

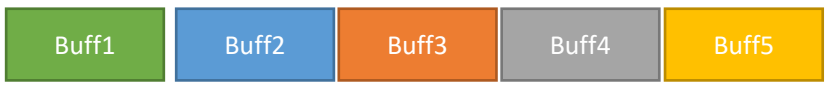
接下来加载权重与偏置，这里是查看文件属性，看下大小加进去就可以了。

接下来是是内存管理。由于原作者没给注释，看懂这个还是比较费力的。这部分的思想就是，上一层的输出作为这一层的输入，那么在计算完成以后，这部分就不再具有利用价值，又作为的新的空间。
图示如下：

逐层推演如下，在第 0 层之前，将输入送到 Buff1，而 Buff2 作为输出的地址；
第 1 层：Buff2 作为输入的地址，Buff1 在上一步已经计算完成失去了利用的价值，作为新的输出地址；
第 2 层：Buff1 又作为新的输入地址，而 Buff2 作为输出地址，以此往复。
但是上面的形式只是作为最简单的连续网络，而 YOLO2 与 YOLO3 都不是简单单向的连续网络。他们都有拼接层，也就是需要保留之前的结果，特别是 YOLO3 与 YOLO4，有大量的拼接层会利用上层的信息。
原作者在使用 YOLO2 时，在拼接层的时候将地址向后偏移了这层的输出大小的长度，也就演变成了如下形式：

在拼接层后，新的地址使用被裁剪后的 Buff2，在最终计算再设置成原始的 Buff2 就可以了。

但是 YOLO3 的拼接操作是非常多的，要使用这样的方法会造成大量的不便，出错也很难排查。



推演 YOLO3 前几层：
第 0 层：输入来自 Buff1，输出到 Buff2；
第 1 层：输入来自 Buff2，输出到 Buff1；由于第一层的输出是需要在第四层做叠加，因此需要保留，也就是说，第二层以及第三层不能占用 Buff1，否则会覆盖结果。
第 2 层：输入来自 Buff1，输出到 Buff2；
第 3 层：输入来自 Buff2，输出到 Buff3；//注意不能占用 Buff1
第 4 层：输入分别来自 Buff1 和 Buff3，输出到 Buff2；
以此往复，已经证得 YOLO3 只需要 5 个缓冲区就可以完成全部计算，但是我在实现的时候不知道是什么原因，这一版的内存管理在 PC 上可以运行，但是在 FPGA 上不可，所以 FPGA 还是采用的旧版本。
直到 36 层，由于 98 层会利用到 36 层的结果，而且 98 是组合 97 与 36 层，所以 36 层的输出地址设置成 Buffi+LEN97(97 层的输出长度)，而 97 层的输出地址设置为 Buffi，这样子直接就在 97 层计算输出的瞬间完成了 98 层，不需要拷贝出来做剪切。
其余同理，如果更改到别的版本也是同理。

接下来是计算每一层传给 YOLO2_FPGA 的参数，则按照逐层的特点分别计算，这里需要微调一下 YOLO2_FPGA 里面的内容。大部分改动都可以理解，但是要遵循以下的原则，其中传入的 kernel_size 和 kernel_stride 不能为 0，因为加载输入输出是需要这些值的，为 0 的层需要设置为 1，upsample 除外，虽然 stride 也是 2，但是是放大的 2 ，也就是说实际上是相当于÷1/2，也是要设置为 1（换个角度，加载输入是逐个加载的，因此需要设置为 1）.
其中
TM * mLoops 是计算以后的通道数量 等于 l.n
TN * nLoops 是计算以前的通道数量 等于 l.c
TR * rLoops 是计算的行数量 等于 l.h
TC * cLoops 是计算的列数量 等于 l.w
这里需要满足的是 TN 一定要≤4，因为在 FPGA 计算核内，是设置为 4，TR 与 TC 则是要小于等于 53，同样也是在 FPGA 里设置好的；每次输出最大的 size 是 4*26*26 因此对于 Upsample 的设置，逐层输入都需要分块按 4*13*13 做输入，因此直接在代码里做固定设置。其中 T2Rate 和 trow_loops 的含义是啥还不清楚。这部分不确定的话就先在 PC 上实现然后测试。

至于别的细节就参考代码注释，再没有的就是我也不懂，可以在 YOLO2 的 git 上面问原作者，那个人还挺好的，基本上有问必答。

第二步是编译生成 IP 核。什么是 IP 核，也就是用高层（c 语言）语言编写的满足算法要求的，使用 Xilinx 软件自行生成的 VHDL 或者 Verilog 语言的 FPGA 程序，使用 HLS 可以大量的减轻工作量（比如 YOLO3 如果使用 VHDL 或者 verilog，你就需要对每一层都实现一次，然后调用，期间还包括大量的输入输出驱动程序，如果没有很强基础或者是有意，尽量避免。其一工作量极大，其二错误很难排查）。

这一步中，如果你是在第一步已经在 PC 上写好了自定义的计算函数，那么你需要修改一些数据类型就可以了（例如 PC 是 float，IP 里面是 short 诸如此类）。如果你是先完成功能部分，最后再写 IP，那你最好先在 PC 上写好。如果你不想在 PC 上写好 C 语言的，那么你需

要建立一个新的工程来实现你的 HLS 代码，在独立的 HLS 计算部分完成后直接拼接也是可以的。

HLS 软件本身的 Bug 是比较多的，在这里可以设置时钟周期，你的设置时钟周期需要＞预测时长和 uncertain 时间，否则会爆红，然后无法使用，由于开发板性能不同，这里也可以做修改，这里我们的经过测试为 10ns，这个时间也会因为你的计算复杂度而增加，计算越复杂，需要时间越大。此外 HLS 还需要在 Xilinx 社区上查看所有版本对应的 bug，尽可能选择一个 Bug 不是很恶心的版本来用，并不是说越新越好，这里我选了 2017.4，关于安装等步骤可以参考 YOLO2 复现部分的。

关于这部分代码的改动，比较主要的是写回函数的改动，由于在 PC 上写回是大面积的连续空间拷贝就可以。但是这里是逐个缓冲区的方式写回，又因为 pingpong 操作的存在，非 conv 层的层循环界限需要设置为 mLoops+2（第一个不写回，因为没有计算结果，最后一层也是）而 Upsample 的时候因为是增大（除此之外都是减小，原作者很明显没有考虑增大的情况，我想增大缓冲区但是 Buf 利用率直接爆到 200 多，时间增加到 12ns 是很不划算的）因此需要在主控函数里设置写回函数。我不清楚这种方式能否在别的网络中运行，但是这也是受到资源数量的限制。

再有就是关于位数对其的情况，第一步生成的那些位数就是在这里使用的，因此 shortcut 也就是加法的 res 层需要设置位数对齐，（举例第四层来自于第一层和第三层，但是第一层的位数是 11，第三层的位数可能是 12，这时候就需要做对其），那么究竟是如何对其呢？为了便于下一层卷积层使用，选择 l.index 与 i-1 层对其，也就是第一层向第三层对齐。注意这里是加法的 Res，所以我选择先向同一个位数对齐，运算后再对齐回去，这里就是会出现溢出的原因。（还是以上面的例子，因为第一层在 11 的时候不溢出，可能在 12 的时候溢出了，同时对到 12 再计算，很有可能就溢出了），如果是乘法的 Res 层，那么你需要仿照 Conv 层的模式来进行计算。

关于位数 INTERWIDTH 设置成 20 的原因，原作者说是经过测试，可以修改根据自己的进行调整，都是以精度为准。

第三步基本上是按照 YOLO2 复现中设置的步骤做，没什么，注意 petalinux 的安装就可以了（wiki 里有）

这里说一下 HLS 模拟，HLS 的模拟比较慢，但是 HLS 模拟没问题的上板子一般问题不会很大（因为可以保证 IP 没问题，不需要编译，基本上就是软件的问题）。之前一直不懂，用整个网络做模拟，HLS 相对很慢，如果可以的话，可以使用 IP 核的部分功能，逐个功能逐个功能的做模拟，举个例子，YOLO3 有 Res，Upsample，conv 三种需要计算的，可以写一个只有三层的网络分别是这三层然后模拟一下，这里原作者是全网络模拟，当时也不太懂，被坑了。这里自己写调试代码就可以了，前提是你懂一些 HLS 代码规范以及软件使用，这块也是在快做完才想通了。（HLS 模拟的 bench 不是在板子上运行的东西，虽然代码比较像）

关于在板子上运行的代码，编译时注意 wiki 里面的信息就可以。原作者用 SDK 编译，那个太麻烦了，因为需要启动 vivado 然后才能启动 SDK，不如使用命令直接编译，快得多。

板子上运行的代码除了与 PC 上的有类型差异，还有就是我修改了位数的设置，由于 Res 需要对齐，而原作者只需要用到 Conv 层，因此需要做改进。除此之外，Main 函数的地址也需要做改进。

```
unsigned int WEIGHT_BASE = 0x10000000;//不用改
//基址地址，要和设备树文件里面的预留地址开头对应起来
unsigned int BETA_BASE = 0x1EC1F000;
//BETA_BASE=hex( ceil( ( sizeof(weight)+sizeof(BIAS) )/4096 )*4096 )
//也就是权重大小+偏移量文件大小对 4096 的商向上取整再乘以 4096
unsigned int MEM_BASE    = 0x1ECF0000;// 49770*1024*4 = 203857920 = C26A000
//BIAS 的大小对 4096 的商进行取整后得到的
```

为什么是 4096 那是因为 Zynq 开发板一个页面的大小是 4096，这也是为什么在代码里拷贝函数会显示 Access Failed，
而在 IPcore 生成了系统以后，之前 IP 的函数也就固化在了开发板里面，那么你接下来只需要在这部分的程序里，把 IP 需要的东西准备好，放在那就可以了，这也就是为什么 elf 文件的 YOLOv2_FPGA 只传地址。关于 xconv_hw.h 这个文件，你如果没有改动 IP 里面 YOLOv2_FPGA 函数的声明，那么这个以及 Vivado 工程的一切就都不需要改动，如果你改了 IP 里面 YOLOv2_FPGA 的声明，那么你就需要对这些都做改动，直到满足你的开发板需要。

关于量化，浮点数的计算相对于定点数相当慢，因此大部分 FPGA 都是使用定点数，关于定点到多少也是有研究的，有固定到 4 位，8 位，16 位。这里使用的是上限固定 16 位，但是实际上动态的过程。关于量化误差评判的标准，权重与 BIAS 都是平均值，而 INOUT 的部分则是贪心的极值，注意这里会在 SHORTCUT 对其的时候产生很大的麻烦，因此最好在 INOUT 量化的时候限制在 13 或者 14 左右，否则最终测试的时候调整位数会非常复杂（如果你的网络本身值比较均匀就另当别论）

大体上的就这一些。