第 7 章　综合实验

前述章节实验完成了图像处理领域中的风格迁移应用在智能处理器上的完整开发和优化。随着人工智能在更广泛领域取得了良好效果，本章的综合实验将涉及在多个不同领域（如目标检测、文本识别、自然语言处理和语音合成等）的人工智能应用在智能处理器上的开发和优化。具体而言，第7.1节介绍基于 YOLOv3 网络模型[20] 实现目标检测应用，并借助 TensorFlow 框架及流处理框架在智能处理器上进行部署。第7.2节介绍基于 EAST 网络模型[21] 实现文字识别应用在智能处理器上的部署。第7.3介绍基于 BERT 网络模型[22] 在智能处理器上的部署。

7.1　目标检测-YOLOv3

7.1.1　实验目的

本实验主要完成将目标检测的代表性算法——YOLOv3 网络移植到智能处理器 DLP 上，并进行性能优化与离线部署，使读者可以借助 DLP 处理器完成完整的目标检测任务。

因此，本实验的主要目的包括：

1. 通过用智能编程语言实现 YOLOv3 的部分后处理操作，深入掌握智能编程语言的算子开发和优化方法；
2. 通过在 TensorFlow 中添加大算子，掌握在 TensorFlow 框架中添加融合算子的方法；
3. 通过使用 TensorFlow 进行在线推理，掌握使用 TensorFlow 编写目标检测应用并在典型 DLP 上进行优化的方法；

实验时间预计为两周。

中科院计算所

7.1.2　背景介绍

7.1.2.1　目标检测

目标检测是指针对一个或多个特定类别在数字图像中进行视觉对象的分类和定位。目标检测属于多任务学习，也就是说一个任务分支需要通过分类算法将物体根据类别划分，另一个任务分支是在判断类别后标记出每一个类别的具体位置信息，即中心坐标位置或及坐标位置（以目标图片的左上角为原点（0,0））。随着技术的发展，目标检测逐渐成为其他计算机视觉任务的基础，如图像分割、图像字幕、物体追踪等；目前，目标检测已广泛应用于自动驾驶、机器人视觉、安防监控等领域。目标检测的两个任务分支如下图 7.2所示：

在介绍目标检测算法原理前，先列举一些目标检测中的常见概念如表 7.1所示，包括平均精度均值（mean Average Precision，mAP）、滑动窗口（Sliding Window）、选择搜索算法

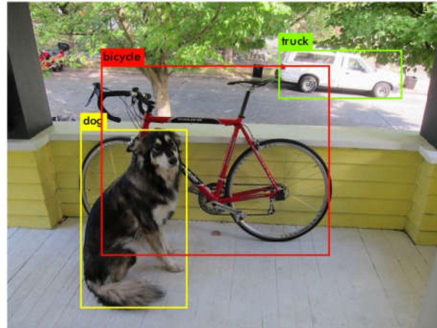

图 7.1　目标检测效果

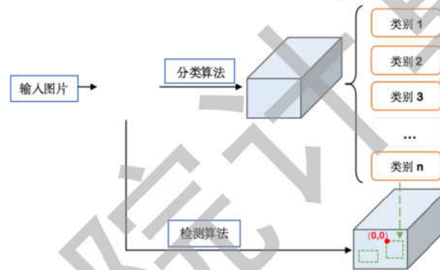

图 7.2　目标检测的多个任务分支

中科院计算所

表 7.1　目标检测常见知识

名称	定义
平均精度均值 mAP	AP 定义为 Precision-recall 曲线下的面积，mAP 为多个检测目标类别的 AP 均值，区间在 [0,1] 之间，越接近 1 证明精确度越高
滑动窗口 Sliding Window	采用不同大小和比例（宽高比）的窗口在全图片上以一定的步长进行滑动，以便于对应的区域做图像分类
选择搜索算法 Selective Search	将图片分为多个子区域，根据子区域的相似性不断迭代合并，在此过程中对合并区域实现外切矩形，得到候选框
交并比 IOU	用于预测 Bounding-box 和人工标注 Bounding-box 的重叠度，两个矩形框的 IOU 计算公式：IOU=(A∩B)/(A□B)
兴趣区域（RoI）	从被处理的图像以方框或不规则形状等方式勾勒出需要处理的区域
非极大值抑制 NMS	依据候选区域和 score 矩阵，判断置信度最高的 bounding box，若预测框重叠，则选择置信度得分高的边框

（Selective Search，SS）、交并比（Intersection over Union，IoU）、兴趣区域（Region of Interest，RoI）、非极大值抑制（Non-Maximum Suppressionm，NMS）等。

依据深度学习算法阶段不同，目标检测方法主要可以分为两类：一种是基于 Region Proposal 的两阶段检测，它首先由算法计算出目标候选框，然后再通过卷积神经网络对目标候选框进行分类与回归, 主流的算法有 R-CNN、Fast R-CNN、Faster R-CNN 等；另一种是单阶段检测，不再需要候选区域，仅使用卷积神经网络直接将目标定位框转为回归问题，预测不同类别的目标位置，主流的算法有 YOLO、SSD 等。

- **两阶段检测**

我们以典型的 Faster R-CNN 算法介绍两阶段检测算法的主要流程。

Faster R-CNN 针对 FAST RCNN 检测速度的瓶颈做了进一步改进，摆脱了传统意义上的 SS 算法的限制，取而代之的是使用 Region Proposal Network (RPN) 生成 RoI，计算步骤如下：

- 通过共享卷积层提取特征图
- 采用 RPN 提取特征图中的目标候选框，判断 RoI 后进行首次边框修正
- 利用 RoI Pooling Laye 在 RNP 输出的特征图上计算 RoI 对应的固定维度特征
- 将目标框使用全连接层进行分类，并对边框进行二次修正

Faster R-CNN 是第一个真正意义上的端到端的算法，将 RPN 替代 SS 使得检测速度大幅提升，同时在 VOC07 数据集上的 mAP 达到 73.2%[23]。Faster R-CNN 有一个主要问题在于后续的检测阶段候选框区域提取上耗时较长。

- **单阶段检测**

我们以典型的 YOLO 算法介绍单阶段检测算法的主要流程。

针对两阶段检测算法存在的计算冗余、检测速度较慢等问题，在 YOLO（You Only Look Once）算法中提出“一步完成”的单级检测器，这意味着将目标分类和目标检测在同一个步骤中实现，其算法的主要步骤如下：

- 图片缩放：输入图片格式调整为 448*448；
- 网络预测：采用单个卷积神经网络对整个图像进行 Region Proposal 的选取以及目标置信度和位置输出；
- 预测框筛选：利用 IOU 筛选 Bounding Box 得到目标检测的预测框。

和两阶段检测网络相比，单阶段检测网络的速度得到大幅度提升，但由于它对输入图像的分割较粗糙，导致后续目标定位精度有所下降，对于小目标的检测结果不尽如人意。

中科院计算所

7.1.2.2 YOLOv3

本节实验所使用的YOLOv3网络是经典的单阶段检测方法,它在延续之前版本YOLOv1和YOLOv2主要特点（包括：通过划分单元格来做检测，使用Leaky ReLU[24]作为激活函数，实现端到端训练，使用Batch Normalization[?]避免过拟合、使用多尺度训练等）的基础上，通过修改Backbone网络来提升精度与性能。

下面主要介绍YOLOv3的网络结构和实验中需要采用编程语言实现的采用NMS来进行候选框筛选的部分。

- **网络结构**

图7.3描述了YOLOv3的结构图，包含了从图片输入经过Darknet-53，Concat拼接到输出3个不同尺度的Feature Map的过程。

其中DBL是Yolov3的基本计算单元，它由一个DarknetConv2d、一个BatchNorm和一个LeakyRelu操作线性排列组成，BatchNorm和LeakyRelu是conv2d卷积层后必不可少的部分（不包括最后一个卷积层），每个算子的功能如下表7.2所示：

表 7.2 DBL 的基本计算单元

算子名称	功能
DarknetConv2d	Darknet是Yolov3的基础网络，1个Darknet的2维卷积Conv2D层，即DarknetConv2D()：操作流程：通过kernel_regularizer将卷积核的参数进行L2正则化，当Padding的strides=（2,2）时采用valid模式，其他Padding一般采用same模式；其余操作与常规二维卷积Conv2D()保持一致。
BN	BatchNormalization()，将输入数据进行正则化，把每层神经网络的输入值的分布强制拉回到均值为0方差为1，的标准正态分布，得到的输出值作为激活输入数据，分布在非线性函数的敏感区，加快收敛速度。
Leaky()	LeakyReLU激活函数是ReLU的变换，公式为:合 y = max(0, x) + leak*min(0,x)；（斜率为0.1）

Res_unit是Resblock_body的基本组件之一，它是由两个DBL和一个ADD组合，借鉴了Resnet的残差结构，即将每个Res_unit的输入经过两次DBL处理后生成的特征图与输入自身进行叠加（Add），Add操作仅是直接相加并不会导致Tensor维度的改变。引入残差结构使Yolov网络深度从Darknet-19（Yolov2）增加至Darknet-53（Yolov3），残差层可以减小网络过深而引起的梯度爆炸风险，提高网络的训练效率。

Resblock_body在网络结构中作为Yolov3的大组件，用Res_n表示，n为在此结构中Res_unit的个数，它是由一个ZeroPadding2D、一个DBL、N个Res_unit线性排列组成。ZeroPadding2D根据下一步操作的需要，将输入矩阵的边界进行0充填，以达到信息补齐的效果。

Concat是将不同层的Tensor进行拼接，在Yolov3中主要是将darknet中间层的输出与上采样结果进行维度扩张，以获得不同的输出。

Darknet.output输出的底层信息包含全局特征信息，DarkNet中间层包含局部特征信息，通过Concat拼接，可提高检测精度。

中科院计算所

值得注意的是 Yolov3 的对象分类预测并没有延续 Yolov2 中的 Softmax，而是采用了 Logistic Regression 对输出的 bbox 进行预测类别，这是因为 Softmax 要求每个目标对象具有相互独立的标签，而数据集中每个对象需要支持多个标签，将 Softmax 用 Logistic 可以预测每个类别的得分。

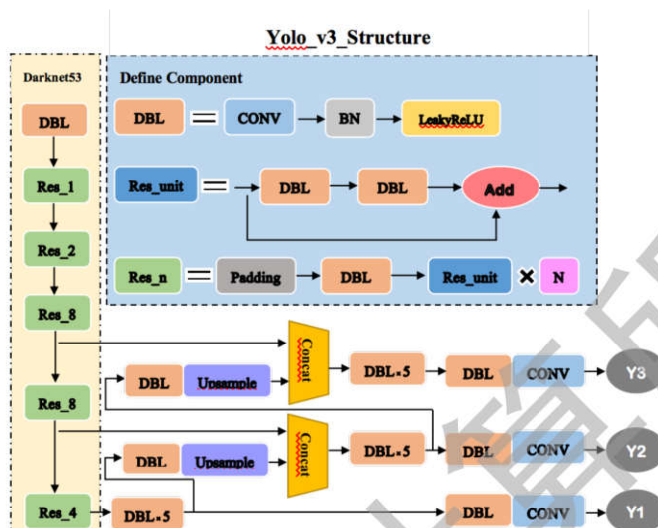

图 7.3　YOLOv3 结构图

对于网络输出，Yolov3 中的获得 anchorbox 的方案依旧使用了 Yolov2 的聚类法，首先计算出 9 个聚类中心，再按照大小均匀分给 3 个尺度，经过卷积层对输出采用多尺度检测技术，也就是说有个三个不同尺度的输出特征图，分别为 13×13、26×26、52×52，其深度都为 255。由于每个网络单元都有 3 个检测框，每个框包含检测框中心坐标（x,y）、检测框的长度（h）、检测框的宽度（w）以及检测框的置信度（confidence）五个基本参数，80 个检测类别都有对应的概率，所以特征图的深度为 255（3×(5 + 80) = 255）。三个不同尺度的输出如下表7.3所示，对应与图 7.3中的三个输出（Y1、Y2 和 Y3）：

表 7.3　YoloV3 的网络输出

尺度	检测类型	实现方式
13*13	大型目标	Darknet 的网络最终输出结构为 13131024，通过设置 512 通道个数和 $num_{a}nchors*(num_classes+5)$，得到预测矩阵 Y1。
26*26	中型目标	通过 DBL 将通道数由 512 调整至 256，再经过上采样 UpSampling2D，将 13×13 的网络结构转换为 26×26，得到的网络结构图与 DarkNet 的中间层输出结果进行拼接，最终结果即是第 2 个尺度特征图 Y2。
52*52	小型目标	通过 DBL 将通道数由 256 调整至 128，再经过上采样 UpSampling2D，将 26×26 的网络结构转换为 52×52，得到的网络结构图与 DarkNet 的中间层输出结果进行拼接，最终结果即是第 3 个尺度特征图 Y3。

YOLOv3 中使用 Darknet-53 作为 backbone 网络用以提取图片的特征信息，相较于 YOLOv2 中的 Darknet-19，其使用步长为 2 的卷积来替代之前的池化操作，同时采用 ResNet[?] 的残差结构（resblock）提升精度。与 ResNet-152 和 ResNet-101 相比，Darknet-53 网络层数较少，不仅分类精度没有下降，检测速度还有所提升。图 7.4展示了 Darknet-53 的结构，通过 5 次

中科院计算所

步长为 2 的卷积进行尺寸变换，最终输出的 Feature Map 为输入的 1/32。

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
	Convolutional	32	1 × 1	
1×	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
	Convolutional	64	1 × 1	
2×	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
	Convolutional	128	1 × 1	
8×	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
	Convolutional	256	1 × 1	
8×	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
	Convolutional	512	1 × 1	
4×	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

图 7.4　Darknet-53 结构图

Yolov3 中的 Darknet-53 网络结构有以下两个特点：

1. 将全卷积层应用于整个网路: Darknet-53 是 53 个卷积层和池化层的组合，但在 Yolov3 中并没有包含 darknet-53 最后一个全局平均池化层，所以减小张量维度是通过改变 5 次卷积核的步长来实现的。全卷积结构也应用于预测分支，最后一个卷积核的规模为 1×1×1024×255，其中 255 是针对 COCO 数据集的 80 类别确定的卷积核个数。

2. 借鉴 Resnet 思想，引入残差（Rsidual）结构: 网络深度越深，特征信息表达越好，但随着网路深度的加深，训练误差趋势会先下降再上升，也就意味着网络深度越深，模型训练难度越大；所以采用残差模块不仅可以保证在网络深度加深的情况仍能保持收敛状态，还在一定程度上大大减少了计算量。

- **非极大值抑制：NMS**

得到 3 种不同尺度的 Feature Map 后需要进行后处理操作来筛选出合适的框以完成检测，NMS 非极大值抑制是其中重要的操作。NMS[25]（非极大值抑制）是一种针对目标框的多重性而提出的检测技术，是目标检测网络中一个重要环节，由于目标检测相邻的交叉窗口经常出现分数相近的情况，使用 NMS 可以去掉重复的检测框。下图 7.5展示了从输出预选框到生成最终检测框的过程（待增加更多 NMS 原理流程部分介绍）。

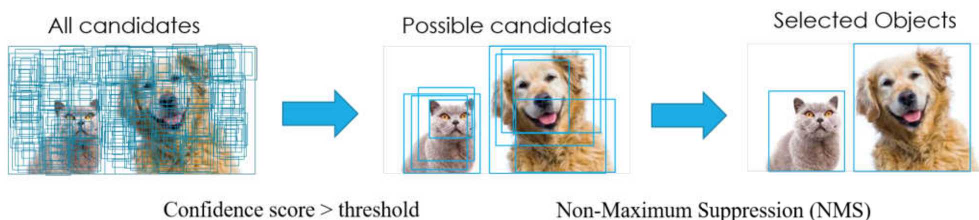

图 7.5　NMS 示意图

NMS 的计算过程可概括为以下四个步骤：

1. 对所有预测框的置信度降序排序；
2. 选出置信度最高的预测框，确认其为正确预测，并计算它与其他预测框的 IOU；
3. 根据（2）中计算的 IOU 去除重叠度高的框，即 IOU> $thresh_iou$ 就删除；
4. 剩下的预测框返回第 1 步，直到没有置信度 > $thresh_score$ 的框为止。

由于目标检测类算法一般都包含 NMS 计算环节，之外许多其它场景下也可以用到 NMS 计算过程（例如文本识别场景）。在这样的背景下，有必要将 NMS 模块在 DLP 硬件上的实现整体抽象剥离出来，单独封装成一个 NMS 通用模块，不仅适用于任何检测算法，还可以缩短其它检测算法的开发时间，增加代码的可维护性与稳定性。因此检测网络综合实验中设计了一个 NMS 通用模块任务，以.h 文件提供给调用程序，将 NMS 计算过程封装成 nms_detection 函数，供 YOLOv3 后处理直接调用。

7.1.3 流处理框架

为了方便在 DLP 上的离线部署，DLP 硬件在运行时库的基础上进一步提供了流处理框架 CNStream，用于完成从数据处理到离线推理等任务的流水线处理。CNStream 是面向 DLP 的数据流处理 SDK。用户可以根据 CNStream 提供的接口，开发实现自己的组件。还可以通过组件之间的互连，灵活地实现自己的业务需求。CNStream 能够大大简化 DLP 提供的推理和其他处理，如视频解码、神经网络图像前处理的集成。也能够在兼顾灵活性的同时，充分发挥 DLP 的硬件解码和机器学习算法的运算性能。

CNStream 基于模块化和流水线的思想，提供了一套基于 C++ 语言的接口来支持流处理多路并发的 Pipeline 框架。为用户提供可自定义的模块机制以及通过高度抽象的 CNFrame-Info 类型进行模块间的数据传输，满足用户对性能和可伸缩性的需求。

7.1.4 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：硬件平台使用第 **??**节介绍的云平台中的 DLP 硬件。
- 软件环境：所涉及的 DLP 软件开发模块包括编程框架 TensorFlow、高性能库 CNML、运行时库 CNRT、编程语言及编译器、运行时库 CNRT 以及流式编程框架 CNStream。

7.1.5 实验内容

本节实验主要是在 DLP 硬件上实现并优化以 YOLOv3 网络模型为核心的目标检测应用。针对用户从网络上下载的标准 YOLOv3 模型，经过模型量化并配置 DLP 相关参数，则可以采用 DLP 上的定制 TensorFlow 版本来运行。为了充分发挥 DLP 的计算能力，进一步采用智能编程语言 BCL 实现 YOLOv3 中的 NMS 部分，并将其作为一个大算子集成到 TensorFlow 框架中。考虑推理应用的实际需求，采用 DLP 的流式框架编写完整目标检测应用，基于离线模型进行部署。具体实验内容包括：

中科院计算所

1. **YOLOv3 DLP 运行**：将标准的 YOLOv3 网络模型进行量化后，通过定制的 TensorFlow 版本在 DLP 硬件上运行；
2. **NMS 的 BCL 实现**：采用 BCL 实现 NMS 后处理运算；
3. **框架算子集成**：将用 BCL 实现的 NMS 算子集成到 TensorFlow 框架中，进行在线推理。

7.1.6 实验步骤

如前所述，详细的实验步骤主要包括：YOLOv3 DLP 运行、NMS 的 BCL 实现、框架算子集成等。

7.1.6.1 YOLOv3 DLP 运行

为了使得标准的 YOLOv3 网络模型在 DLP 上运行，主要需要完成：YOLOv3 CPU 运行、模型量化和 DLP 运行等流程。

- **YOLOv3 CPU 运行**

通过在 CPU 上正确执行 YOLOv3 标准网络模型完成推理，确保相关依赖都已安装完成。具体步骤如下：

1. 获取开源 YOLOv3 工程：git clone https://github.com/YunYang1994/tensorflow-yolov3.git
2. 安装推理所需软件包：一些必备的 Python 安装包在所提供的云平台软件环境中已经安装完成，但对于每个综合实验的特有依赖还需进行单独安装，使用图 7.6中的命令完成依赖的安装与升级。

```
$ cd tensorflow-yolov3
$ pip install -r ./docs/requirements.txt
```

图 7.6 YOLOv3 依赖安装

3. 模型下载与转换：执行图 7.7中的命令完成模型的下载与转换。要执行这一步的原因是模型文件和 ckpt 文件之间有一些版本上的差异，需要将节点名等细节对齐。

```
$ cd checkpoint
$ wget https://github.com/YunYang1994/tensorflow-yolov3/releases/download/v1.0/
    yolov3_coco.tar.gz
$ tar -xvf yolov3_coco.tar.gz
$ cd ..
$ python convert_weight.py
$ python freeze_graph.py
```

图 7.7 YOLOv3 模型下载与转换

4. 模型推理与验证：执行图 7.8中的命令完成在 CPU 上的推理。

中科院计算所

```
$ python image_demo.py
```

图 7.8 YOLOv3 CPU 推理

- **模型量化**

下面介绍如何将开源的 pb 模型量化为 INT8 pb 模型。DLP 软件环境提供 fppb_to_intpb 工具来进行模型量化。该工具在 TensorFlow 根目录的/tensorflow/cambricon_examples/tools/fppb_to_intpb 目录下。具体的量化分为以下几个步骤：

1. **生成数据文件**:按照模型所需要的数据集以及数据集的具体路径来修改 generate_image_list.py 文件中的 image_libs_map，来生成相应的数据文件。YOLOv3 需要 coco 数据集，因此将 image_libs_map 进行如图 7.9所示的修改。完成修改后执行 generate_image_list.py 来生成数据文件。

```
import os
from os.path import isfile, join

max_image_count = 100
image_libs_map = {
      "coco":join(os.environ.get( "COCO_DATASET_HOME" ), "COCO/test2017")
      }
if __name__ == "__main__":
  for image_lib, image_path in image_libs_map.items():
    n = 0
    with open("image_list_{}".format(image_lib), "w") as image_list:
      for root, dirs, files in os.walk(image_path):
        for f in files:
          if n >= max_image_count:
            break
          if f.endswith("jpg"):
            image_list.write(join(image_path, f) + "\n")
            n = n + 1
```

图 7.9

2. **生成量化配置文件**：如第 ??所述，在进行量化前需设置相应的配置文件。以指定量化相关的参数。DLP 软件环境提供了 generate_ini.py 来生成指定模型的量化配置文件。针对 YOLOv3，需要将其中的 model_name 进行如图 7.10所示的修改，即指定要生成的配置文件是针对 YOLOv3 的。

```
models_name = ['yolov3']
```

图 7.10

执行 generate_ini.py 脚本后会生成 config 文件夹，其中包含了 yolov3_naive_int8.ini 参数配置文件。该配置文件中包含如图 7.11所示的信息。

其中，可以通过 activation_quantization_alg 和 weight_quantization_alg 来设置具体的量化模式。包括 naive 和 threshold_search 两种方式。

中科院计算所

```
[preprocess]
mean = 0, 0, 0        #均值，顺序依次为 mean_r、 mean_g、 mean_b
std = 255.0           #方差
color_mode = rgb      #网络的输入图片是 rgb 还是 bgr
crop = 416, 416       #前处理最终将图片处理为 416 * 416 大小
calibration = yolov3_preprocess_cali     #校准数据读取及前处理的方式，可以根据需求进行自
    定义，[preprocess] 和 [data] 中定义的参数均为 calibration 的输入参数

[config]
activation_quantization_alg = naive #输入量化模式，可选 naive 和 threshold_search，naive
    为基础模式，threshold_search 为阈值搜索模式
device_mode = clean           #可选 clean、mlu 和 origin，建议使用 clean，使用 clean
    生成的模型在运行时会自动选择可运行的设备
use_convfirst = False         #是否使用 convfirst
quantization_type = int8      #量化位宽，目前可选 int8 和 int16
debug = False                 #是否为debug模式
weight_quantization_alg = naive #权值量化模式，可选 naive 和 threshold_search，naive 为
    基础模式，threshold_search 为阈值搜索模式
int_op_list = Conv, FC, LRN   #要量化的 layer 的类型，目前只能量化 Conv、FC 和 LRN
channel_quantization = False  #是否使用分通道量化，目前不支持为 True

[model]
output_tensor_names = pred_sbbox/concat_2:0, pred_mbbox/concat_2:0, pred_lbbox/concat_2
    :0        #输出 Tensor 的名字，可以是多个，以逗号隔开
original_models_path = ./realpath-of-yolov3_coco.pb #输入pb
save_model_path = ./pbs/yolov3/yolov3_int8.pb       #输出pb
input_tensor_names = input/input_data:0     #输入 Tensor 的名字，可以是多个，以逗号隔开

[data]
num_runs = 2     #运行次数
data_path = ./image_list_coco     #校准数据文件路径
batch_size = 10 #每次运行的batch_size
```

图 7.11

native 模式对应于第**??**节的量化方式，直接统计数据的最大值和最小值完成量化。

threshold_search 阈值搜索模式用于处理存在异常值的待量化数据集，该模式能够过滤部分异常值，重新计算出数据集的最值，用新最值来计算数据集的量化参数，从而提高数据集整体的量化质量。在这种模式下，数据量化所使用的最大值和最小值不再由简单的统计得出，而是通过搜索的方式得到。但是对于不存在异常值，数据分布紧凑的情况下，不建议使用该算法，比如网络权值的量化。

其中 device_mode 可以设置输出 pb 所有节点的 device，有三种配置方式：clean、mlu 和 origin。其中，mlu 将输出 pb 的所有节点的 device 都设置为 MLU，即都在 MLU 上运行；clean 将输出 pb 的所有节点的 device 清除，运行时根据算子注册情况自动选择可运行的设备；origin 则使用和输入 pb 一样的设备指定。

3. **完成模型量化**：运行量化工具 python fppb_to_intpb.py config/yolov3_naive_int8,ini 完成模型的量化。

- **DLP 运行**

完成 CPU 侧的推理后，接下来需要将其移植在 DLP 上，以加快其推理速度。此时我

中科院计算所

们只需仿照第 7 章编程语言算子实验中的 DLP 推理移植部分，通过配置 session config 以及进行 INT8 量化即可完成 DLP 移植。下面将详细介绍这两个步骤：

修改上述执行 CPU 推理时的代码 image_demo.py，增加如下几个部分：

```
pb_file = "./realpath-to-int8-pb"
......
config = tf.ConfigProto(allow_soft_placement=True,
                inter_op_parallelism_threads=1,
                        intra_op_parallelism_threads=1)
config.mlu_options.data_parallelism = 1
config.mlu_options.model_parallelism = 1
config.mlu_options.core_num = 4
config.mlu_options.core_version = "MLU270"
config.mlu_options.precision = "int8"
config.mlu_options.save_offline_model = False
config.mlu_options.offline_model_name = "yolov3_int8.cambricon"
with tf.Session(config = config, graph=graph) as sess:
    ......
```

图 7.12　YOLOv3 DLP 移植（1）

7.1.6.2　NMS BangC 代码实现

在使用 BCL 设计 NMS 模块时，需要考虑以下涉及原则：

1. 能够处理的数据来源：GDRAM，NRAM，SRAM 中的一种
2. 能够将数据存放到：GDRAM，NRAM，SRAM 中的一种
3. 能够以不同的储存格式存放数据
4. 能够在 BLOCK 和 U1 的模式下工作
5. 能够处理任意数据规模

使用 BangC 实现时，如果按照背景介绍章节中 NMS 原理进行计算，则会显得较为复杂，比如第 (3) 步中删除交并比大于阈值的框，使用 BangC 语言实现这个操作需要来回拷贝数据，导致效率不高。为了解决此问题，标准程序中采用另外一种方式，即不改变之前的数据，将被删除框对应的分数置为 0，下一轮筛查的时，删除的框必然不会对此轮筛查造成影响。整体设计流程图如图 7.13所示：

下面以图 7.14中的代码为例进行说明。NMS 通用模块接口介绍：

- output_box_num：NMS 筛选出的框的总个数；
- output_data：NMS 计算结果存放的数据地址；
- dst：NMS 计算结果存放的数据地址类型：GDRAM/SRAM/NRAM；
- input_data_score：输入框 score 的数据地址；

中科院计算所

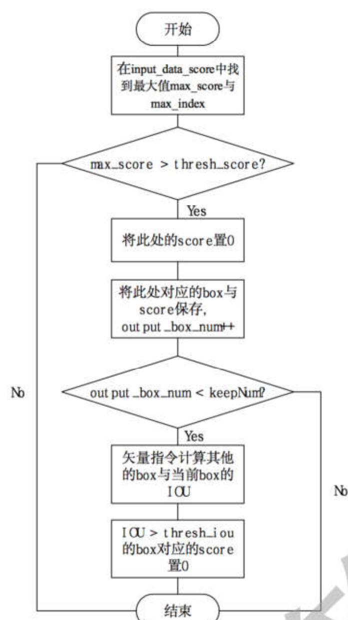

图 7.13　BangC NMS 设计流程图

```
__mlu_func__ void nms_detection(int& output_box_num,
                                NMS_DT* output_data,
                                Addr dst,
                                NMS_DT* input_data_score,
                                NMS_DT* input_data_box,
                                Addr src,
                                NMS_DT *buffer,
                                int buffer_size,
                                NMS_DT* sram,
                                SplitMode split_mode,
                                int input_box_num,
                                int input_stride,
                                int output_stride,
                                int keepNum,
                                NMS_DT thresh_iou,
                                NMS_DT thresh_score,
                                int save_method);
```

图 7.14　接口设计

- input_data_box：输入框坐标的数据地址，储存顺序是：x1，y1，x2，y2，同一个类型的数据储存在一起；
- src：NMS 输入数据存放的数据地址类型：GDRAM/SRAM/NRAM；
- buffer：NMS 计算使用的 NRAM 空间首地址；
- buffer_size：NMS 计算使用的 NRAM 空间大小，单位为字节，当 dst 为 GDRAM/SRAM，则 buffer_size 至少需要：(64 * 4 + 64 + 256 * 5) * sizeof(NMS_DT) 字节；当 dst 为 NRAM，则 buffer_size 至少需要：(64 * 4 + 64) * sizeof(NMS_DT) 字节。注意：如果 src == NRAM 并且 input_box_num 是对齐的，NMS 计算会更加高效，因为可以省去数据搬运工作。

中科院计算所

- sram：SRAM 的地址空间，用来在 U1 模式下通过核间通讯找 score 的最大值，大小至少为 30 * sizeof(NMS_DT) 字节；
- split_mode：拆分模式：NMS_BLOCK/NMS_U1；
- input_box_num：输入框的数量；
- input_stride：输入数据的 stride，即 x1，y1，x2，y2 之间的 stride；
- output_stride：输出数据的 stride，即 x1，y1，x2，y2 之间的 stride；
- keepNum：最多保留筛选框的个数；
- thresh_iou：交并比阈值；
- thresh_score：score 阈值；
- save_method：储存模式，分为 0/1/2

具体实现主要分为两个步骤：

1. 准备阶段：准备所需的变量，防呆检测，空间划分，多核拆分

   (a) 主要变量有：

   - core_limit：启用的核数，BLOCK 模式对应 1，U1 模式对应 4；
   - loop_end_flag：U1 模式结束标识；
   - nram_save_limit_count：NRAM 上临时储存筛选框的数量；
   - MODE：当 src=NRAM 时，0 代表数据需要先加载到制定的 NRAM 上进行计算，1 代表数据直接在 NRAM 上可以直接运算，保证高效，但需要满足两个条件：buffer 空间足够，input_box_num 是对齐的；
   - input_score_ptr：输入框 score 的数据指针；
   - input_x1_ptr：输入框 box 的 x1 坐标数据指针；
   - input_y1_ptr：输入框 box 的 y1 坐标数据指针；
   - input_x2_ptr：输入框 box 的 x2 坐标数据指针；
   - input_y2_ptr：输入框 box 的 y2 坐标数据指针；
   - x1：buffer 空间，存放 x1；
   - y1：buffer 空间，存放 y1；
   - x2：buffer 空间，存放 x2；
   - y2：buffer 空间，存放 y2；
   - score：buffer 空间，存放 score；
   - inter_x1：buffer 空间，IOU 筛选临时空间；
   - inter_y1：buffer 空间，IOU 筛选临时空间；
   - inter_x2：buffer 空间，IOU 筛选临时空间；

中科院计算所

- inter_y2：buffer 空间，IOU 筛选临时空间；
- max_box：buffer 空间，筛选框的信息储存空间，顺序为 max score, x1, y1, x2, y2；
- nram_save：buffer 空间，筛选框的临时储存空间；
- limit = 0：根据 buffer 进行 findlimit，一次最多能处理的输入框的个数；
- len_core = 0：每个核处理的框的个数；
- max_seg_pad = 0：每次处理输入框的个数，根据 limit 进行下补齐，满足硬件限制；
- repeat：整数段，需要处理几次 max_seg_pad；
- remain：余数段，剩下得需要处理的框的个数；
- remain_pad：余数段进行补齐后的框个数，满足硬件限制；
- input_offset：当前核处理的输入数据的起始地址偏移；
- nram_save_count：临时储存空间储存多少个框后再整体拷贝到实际的目标地址；

(b) 空间划分 MODE 为 0 的时候，x1，y2，x2，y2，score 指向的 buffer 空间用来加载数据，MODE 为 1 的时候，不再需要加载数据的空间，inter_x1，inter_y1，inter_x2，inter_y2 是 buffer 上的临时空间，用来计算 NMS，4 块临时空间是考虑了空间复用后最少需要的数量，以节省 buffer，可以一次处理更多的数据。

(c) 多核拆分 src==NRAM 模块内只支持单核模式，用户可根据需求在模块外按类拆分，即每个核负责计算一个类的 NMS 过程，因为模块内无法在 U1 模式时对某一个核上的 NRAM 上的数据进行访问；src==GDRAM/SRAM，模块内支持按数据块进行多核拆分，即一个类的 NMS 过程拆分到多个核上进行计算。多核拆分使用的拆分方式是：每个核上分到框的数量之间相差不超过 1，即尽可能的将数据平均分配到多个核上。

2. 执行阶段：主要操作都放在 for (int keep = 0; keep < keepNum; keep++) 循环中，一次找一个框，这个循环有两个退出条件：找到的框数量达到 keepNum，最大的 score 小于 thresh_score。

(a) 搜索最大值模块：通过 __bang_max 指令找到 score 存放数据的最大值与最大索引，如果是 BLOCK 模式，则直接即可找到 score 的最大值与最大索引，如果是 U1 模式，则需要每个核上计算所分到数据的最大值，然后通过 SRAM 进行通信，进一步找到四个核上的最大值与最大值索引。

(b) 保存模块：保存方式主要包括两种，一种是按 score，x1，y1，x2，y2 的顺序一组一组进行储存，一种是按 score，x1，y1，x2，y2 的顺序以此存放。两者区别如下图 7.15所示：

(c) 根据交并比筛选模块：以图 7.16中的代码为例进行说明。

中科院计算所

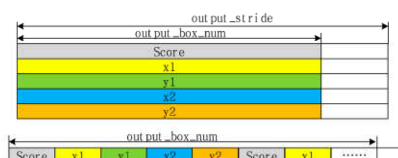

图 7.15　BangC NMS 两种保存方式

```
*IOU计算部分*
// 得到相交部分的面积：
__nramset(inter_y1, seg_len, max_box[1]); // max_x1
__svmax_relu(inter_x1, x1, inter_y1, seg_len); // inter_x1
__nramset(inter_y2, seg_len, max_box[3]); // max_x2
__svmin_relu(inter_x2, x2, inter_y2, seg_len); // inter_x2
__bang_sub(inter_x1, inter_x2, inter_x1, seg_len);
__bang_active_relu(inter_x1, inter_x1, seg_len); // inter_w
__nramset(inter_x2, seg_len, max_box[2]); // max_y1
__svmax_relu(inter_y1, y1, inter_x2, seg_len); // inter_y1
__nramset(inter_x2, seg_len, max_box[4]); // max_y2
__svmin_relu(inter_y2, y2, inter_x2, seg_len); // inter_y2
__bang_sub(inter_y1, inter_y2, inter_y1, seg_len);
__bang_active_relu(inter_y1, inter_y1, seg_len); // inter_h
__bang_mul(inter_x1, inter_x1, inter_y1, seg_len); //area_I
// 得到每个框的面积：area = (x2 - x1) * (y2 - y1)：
__bang_sub(inter_y1, x2, x1, seg_len);
__bang_sub(inter_y2, y2, y1, seg_len);
__bang_mul(inter_x2, inter_y1, inter_y2, seg_len); // area
// 得到相并部分的面积area_U： area + max_area - area_I：
__nramset(inter_y1, seg_len, max_area);
__bang_add(inter_x2, inter_x2, inter_y1, seg_len);
__bang_sub(inter_x2, inter_x2, inter_x1, seg_len); //area_U
*筛选*
// 如果IOU超过阈值，则将相应的框的score置0：area_U * thresh > area_I?
__bang_mul_const(inter_x2, inter_x2, thresh_iou, seg_len);
__bang_gt(inter_x1, inter_x2, inter_x1, seg_len);
__bang_mul(score, score, inter_x1, seg_len);
```

图 7.16　NMS 交并比筛选模块

7.1.6.3　TensorFlow 算子添加

TensorFlow 中添加算子可分为两种情况，一种为 TensorFlow 官方定义的标准流程，可以从官方文档中获取；另一种为添加 MLU 算子的方式，本节将对如何添加 MLU YOLOv3 后处理算子作出介绍。

添加 MLU 算子可分为以下几个步骤：

1.MLUlib 层，该部分代码在 tensorflow/stream_executor/mlu/mlu_api/lib_ops/目录下，主要作用是进一步对 SDK 层进行 API 封装。需要注意的是 MLU LIB 层禁止引用 TensorFlow 的头文件，目前只使用了 tensorflow::Status 类方便做返回值处理。对于该算子来说主要对该目录下的 mlu_lib_ops.cc 和 mlu_lib_ops.h 文件进行修改，示例代码如下所示。

2.MLUOps 层实现，MLUOps 层的职责是使用 LIB 层的 API 完成算子的实现。该部分代码在 tensorflow/stream_executor/mlu/mlu_api/ops 下，注意每个算子首先需要在该目录下的 mlu_ops.h 文件中添加算子类的声明。下图 7.18为添加算子类声明的程序。

中科院计算所

```
tensorflow::Status CreateYolov3DetectionOutputOp(
    MLUBaseOp** op, MLUTensor** input_tensors, MLUTensor** output_tensors,
    cnmlPluginYolov3DetectionOutputOpParam_t param) {
  CNML_RETURN_STATUS(cnmlCreatePluginYolov3DetectionOutputOp(
      op, param, input_tensors, output_tensors));
}

tensorflow::Status ComputeYolov3DetectionOutputOp(MLUBaseOp* op,
                                                   MLUCnrtQueue* queue,
                                                   void* inputs[], int input_num,
                                                   void* outputs[],
                                                   int output_num) {
  int dp = 1;
  cnrtInvokeFuncParam_t compute_forw_param;
  u32_t affinity = 0x01;
  compute_forw_param.data_parallelism = &dp;
  compute_forw_param.affinity = &affinity;
  compute_forw_param.end = CNRT_PARAM_END;

  cnmlComputePluginYolov3DetectionOutputOpForward(
      op, inputs, input_num, outputs, output_num, &compute_forw_param, queue);
}
```

(a)

```
tensorflow::Status CreateYolov3DetectionOutputOp(
    MLUBaseOp** op, MLUTensor** input_tensors, MLUTensor** output_tensors,
    cnmlPluginYolov3DetectionOutputOpParam_t param);

tensorflow::Status ComputeYolov3DetectionOutputOp(MLUBaseOp* op,
                                                   MLUCnrtQueue* queue,
                                                   void* inputs[], int input_num,
                                                   void* outputs[],
                                                   int output_num);
```

(b)

图 7.17 MLULib 层实现

添加完算子声明后还需同一个目录.cc 文件中添加算子的实现，每个算子都需实现 Create 和 Compute 两个方法。下图 7.19和 7.20程序显示了 yolov3detectionoutput.cc 的实现。

Create 方法的职责是将创建好的 baseop 指针调用 base_ops_.push_back(op_ptr); 储存起来；Compute 方法的职责在于调用 lib 层的 compute 函数进行计算，目前算子实现为同步方式，因此需要调用 SyncQueue。

3.MLUStream 实现，MLUStream 负责 MLUOps 算子类的实例化，其接口都定义在 tensorflow/stream_executor/mlu/mlu_stream.h 中。以 MLUOpKernel 和 MLUStream 层的接口特点，可以把 MLUStream 层的算子分为 2 类：通用模版算子与特例化算子。对于 Yolov3DetectionOutput 算子来说，其属于通用模版算子。

通用模版算子满足以下三个条件：MLU 算子的所有输入均来自 OpKernelContext；MLU 算子的所有输出顺序须与 OpkernelContext 的输出顺序一致；MLUTensor 可以被 CreateMLUTensorFromTensor 创建，即 MLU Tensor 的形状、数据类型与 TensorFlow tensor 一致。

4.MLUOpKernel 实现，OpKernel 为 TensorFlow 对算子对抽象定义。MLUOpKernel 继承了 OpKernel 类，其使用方法与 OpKernel 基本一致。对于每个 MLU 算子均需要实现其

中科院计算所

```
struct MLUYolov3DetectionOutputOpParam {
  int batchNum_;
  int inputNum_;
  int classNum_;
  int maskGroupNum_;
  int maxBoxNum_;
  int netw_;
  int neth_;
  float confidence_thresh_;
  float nms_thresh_;
  int* inputWs_;
  int* inputHs_;
  float* biases_;
  MLUYolov3DetectionOutputOpParam(int batchNum, int inputNum, int classNum,
                                  int maskGroupNum, int maxBoxNum, int netw,
                                  int neth, float confidence_thresh,
                                  float nms_thresh, int* inputWs, int* inputHs,
                                  float* biases)
      : batchNum_(batchNum),
        inputNum_(inputNum),
        classNum_(classNum),
        maskGroupNum_(maskGroupNum),
        maxBoxNum_(maxBoxNum),
        netw_(netw),
        neth_(neth),
        confidence_thresh_(confidence_thresh),
        nms_thresh_(nms_thresh),
        inputWs_(inputWs),
        inputHs_(inputHs),
        biases_(biases) {}

};
```

(a)

```
DECLARE_OP_CLASS(MLU...);
DECLARE_OP_CLASS(MLUYolov3DetectionOutput);
DECLARE_OP_CLASS(MLU...);
```

(b)

图 7.18　Yolov3DetectionOutput 算子 MLUOps 声明

中科院计算所

构造函数与 ComputeOnMLU 方法。MLUOpKernel 实现的主要功能有参数检查、参数处理、输出形状推断及输出内存分配、调用 MLUStream 层接口完成算子计算。下图 7.22、 7.23、7.24中的代码分别显示了这四个不同的步骤。input0、input1、input2 为 YOLOv3 结构图 7.3中的三个 feature map 输出。

5. 开始注册 MLU Op Kernel，在 tensorflow/core/kernels 中添加 yolov3_detection_output_op.cc 注册文件和添加 yolov3_detection_output_op_mlu.h 算子实现文件，Op Kernel 注册使用的均为 REGISTER_KERNEL_ BUILDER 宏，如下图 7.25 注意：当前只支持注册 MLU 支持的数据类型，不支持的无法注册；可以通过 HostMemory 对特定输入进行限制，如需要在 CPU 上做处理的输入数据。

6. 算子注册在 tensorflow/core/ops 目录下找到对应的 Op 注册文件，对于 Yolov3DetectionOutput 为 image_ops.cc 文件。通过这个文件可知 Yolov3DetectionOutput 算子有三个输入和一个输

```
Status MLUYolov3DetectionOutput::CreateMLUOp(std::vector<MLUTensor*> &inputs, \
    std::vector<MLUTensor*> &outputs, void *param) {
  TF_PARAMS_CHECK(inputs.size() > 0, "Missing input");
  TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
  MLUBaseOp *op_ptr = nullptr;
  MLUTensor* input0 = inputs.at(0);
  MLUTensor* input1 = inputs.at(1);
  MLUTensor* input2 = inputs.at(2);
  MLUTensor* output = outputs.at(0);
  MLUTensor* buffer = outputs.at(1);

  MLULOG(3) << "CreateYolov3DetectionOutputOp"
            << ", input0: " << lib::MLUTensorUtil(input0).DebugString()
            << ", input1: " << lib::MLUTensorUtil(input1).DebugString()
            << ", input2: " << lib::MLUTensorUtil(input2).DebugString()
            << ", output: " << lib::MLUTensorUtil(output).DebugString()
            << ", buffer: " << lib::MLUTensorUtil(buffer).DebugString();
    int batchNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->batchNum_;
    int inputNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->inputNum_;
    int classNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->classNum_;
    int maskGroupNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->maskGroupNum_;
    int maxBoxNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->maxBoxNum_;
    int netw = ((ops::MLUYolov3DetectionOutputOpParam*)param)->netw_;
    int neth = ((ops::MLUYolov3DetectionOutputOpParam*)param)->neth_;
    float confidence_thresh = ((ops::MLUYolov3DetectionOutputOpParam*)param)->
      confidence_thresh_;
    float nms_thresh = ((ops::MLUYolov3DetectionOutputOpParam*)param)->nms_thresh_;
    int* inputWs = ((ops::MLUYolov3DetectionOutputOpParam*)param)->inputWs_;
    int* inputHs = ((ops::MLUYolov3DetectionOutputOpParam*)param)->inputHs_;
    float* biases = ((ops::MLUYolov3DetectionOutputOpParam*)param)->biases_;

  cnmlPluginYolov3DetectionOutputOpParam_t mlu_param;
    const int num_anchors = 3;
      cnmlCoreVersion_t core_version = CNML_MLU270;
  cnmlCreatePluginYolov3DetectionOutputOpParam(
      &mlu_param,
      batchNum,
      inputNum,
      classNum,
      num_anchors,
      maxBoxNum,
      netw,
      neth,
      confidence_thresh,
      nms_thresh,
      core_version,
      inputWs,
      inputHs,
      biases);
    std::vector<MLUTensor*> input_tensors = {input0, input1, input2};
    std::vector<MLUTensor*> output_tensors = {output, buffer};
    TF_STATUS_CHECK(lib::CreateYolov3DetectionOutputOp(
        &op_ptr, input_tensors.data(), output_tensors.data(), mlu_param));

    base_ops_.push_back(op_ptr);

    return Status::OK();
}
```

图 7.19　Create 函数实现

中科院计算所

```
Status MLUYolov3DetectionOutput::Compute(const std::vector<void *> &inputs,
    const std::vector<void *> &outputs, cnrtQueue_t queue) {
  int num_input = inputs.size();
  int num_output = outputs.size();
  assert(num_input == 3);
  assert(num_output == 2);
  TF_STATUS_CHECK(lib::ComputeYolov3DetectionOutputOp(
      base_ops_.at(0), queue,
        const_cast<void**>(inputs.data()),
        num_input,
        const_cast<void**>(outputs.data()),
        num_output));

  // todo, delete sync queue after delay copy
  cnrtSyncQueue(queue);
  return Status::OK();
}
```

(a)

图 7.20 Compute 函数实现

出，三个输入分别用 input0、input1、input2 标识，输出用 predicts 标识，输入输出的数据类型均用 T 表示即该算子的输出与输入数据类型一致。Attr(”T: type”) 表示 T 允许的数据类型为 type，也就是 TensorFlow 支持的所有数据类型，其余为各超参数配置包括 confidence 阈值以及 NMS 阈值等。

以上 6 个步骤为添加 cnplugin 中 Yolov3DetectionOutput 算子的基本流程，除此之外，为了将该算子最终集成在 TensorFlow 中，编译时还需要在 tensorflow/core/kernels/BUILD 中添加以下信息。

7.1.6.4 DLP 执行

在将 BCL 实现的 Yolov3DetectionOutput 算子成功集成到框架后，需要修改之前转换完成的 int8 pb 模型，在模型中增加 Yolov3DetectionOutput 后处理算子，这一步可以使用编程框架机理章节中介绍的 pb 与 pbtxt 相互转换工具实现。具体在 pb 中添加如下图 7.28所示的节点：注意，由于后处理节点已经将三种类型的 feature map 做过一定处理，与 pb 原模型中的操作有一定冗余，为了确保精度的准确性，选择”conv_lbbox/BiasAdd”，”conv_mbbox/BiasAdd”，”conv_sbbox/BiasAdd” 作为后处理算子的输入节点。至此，只需修改推理代码，将后处理部分注释掉即可完成整个 YOLOv3 模型的在线加速推理工作。

7.1.7 实验评估

相比其它两个综合实验，YOLOv3 实验困难度中等，因此难度系数分为 1.1。学生最终得分为 1.1 乘以相应的得分。

- 60 分标准：补全 nms_detection.h 文件，cnplugin 可正常编过。
- 70 分标准：在 60 分的基础上完成 TensorFlow 的集成编译，完成 pb 模型添加后处理大算子的操作，执行测试脚本时 map 值高于 30%，单 batch 延时（包含后处理）低于

中科院计算所

```
Status Yolov3DetectionOutput(OpKernelContext* ctx,
                             Tensor* tensor_input0,
                             Tensor* tensor_input1,
                             Tensor* tensor_input2,
                             int batchNum,
                             int inputNum,
                             int classNum,
                             int maskGroupNum,
                             int maxBoxNum,
                             int netw,
                             int neth,
                             float confidence_thresh,
                             float nms_thresh,
                             int* inputWs,
                             int* inputHs,
                             float* biases,
                             Tensor* output1,
                             Tensor* output2){
  ops::MLUYolov3DetectionOutputOpParam op_param(
                             batchNum,
                             inputNum,
                             classNum,
                             maskGroupNum,
                             maxBoxNum,
                             netw,
                             neth,
                             confidence_thresh,
                             nms_thresh,
                             inputWs,
                             inputHs,
                             biases);
  return CommonOpImpl<ops::MLUYolov3DetectionOutput>(
      ctx,
      {tensor_input0, tensor_input1, tensor_input2},
      {output1, output2},
      static_cast<void*>(&op_param));
}
```

图 7.21　Yolov3DetectionOutput 算子 MLUOpKernel 实现

300ms。

- 80 分标准：在 70 分的基础上，执行测试脚本时 map 值高于 50%，单 batch 延时（包含后处理）低于 100ms。
- 90 分标准：在 80 分的基础上，执行测试脚本时 map 值高于 54%，单 batch 延时（包含后处理）低于 50ms。
- 100 分标准：在 90 分的基础上，执行测试脚本时 map 值高于 56%，单 batch 延时（包含后处理）低于 25ms。

7.1.8　实验思考

1.NMS BCL 相比 CPU 实现有何优势？2. 有哪些方法可以提升 BCL 算子的性能？

中科院计算所

```
namespace tensorflow {
template<typename T>
class MLUYolov3DetectionOutputOp: public MLUOpKernel{
    public:
        explicit MLUYolov3DetectionOutputOp(OpKernelConstruction* context):MLUOpKernel(
    context){
            OP_REQUIRES_OK(context, context->GetAttr("batchNum",&batchNum_));
            OP_REQUIRES_OK(context, context->GetAttr("inputNum",&inputNum_));
            OP_REQUIRES_OK(context, context->GetAttr("classNum",&classNum_));
            OP_REQUIRES_OK(context, context->GetAttr("maskGroupNum",&maskGroupNum_));
            OP_REQUIRES_OK(context, context->GetAttr("maxBoxNum",&maxBoxNum_));
            OP_REQUIRES_OK(context, context->GetAttr("netw",&netw_));
            OP_REQUIRES_OK(context, context->GetAttr("neth",&neth_));
            OP_REQUIRES_OK(context, context->GetAttr("confidence_thresh",&
    confidence_thresh_));
            OP_REQUIRES_OK(context, context->GetAttr("nms_thresh",&nms_thresh_));
            OP_REQUIRES_OK(context, context->GetAttr("inputWs",&inputWs_));
            OP_REQUIRES_OK(context, context->GetAttr("inputHs",&inputHs_));
            OP_REQUIRES_OK(context, context->GetAttr("biases",&biases_));
        }

        void ComputeOnMLU(OpKernelContext* context) override {
          auto* stream = context->op_device_context()->mlu_stream();
          auto* mlustream_exec =
              context->op_device_context()->mlu_stream()->parent();
//参数检查与处理
          Tensor* input0 = const_cast<Tensor*>(&context->input(0));
          Tensor* input1 = const_cast<Tensor*>(&context->input(1));
          Tensor* input2 = const_cast<Tensor*>(&context->input(2));
          string op_parameter = context->op_kernel().type_string();
          MLU_OP_CHECK_UNSUPPORTED(mlustream_exec, op_parameter, context);
          int c_arr_data[3] = {255, 255, 255};
          std::vector<int> input0_shape(4, 1);
          std::vector<int> input1_shape(4, 1);
          std::vector<int> input2_shape(4, 1);
          input0_shape[0] = batchNum_;
          input0_shape[1] = c_arr_data[0];
          input0_shape[2] = inputHs_[0];
          input0_shape[3] = inputWs_[0];

          input1_shape[0] = batchNum_;
          input1_shape[1] = c_arr_data[1];
          input1_shape[2] = inputHs_[1];
          input1_shape[3] = inputWs_[1];

          input2_shape[0] = batchNum_;
          input2_shape[1] = c_arr_data[2];
          input2_shape[2] = inputHs_[2];
          input2_shape[3] = inputWs_[2];
```

图 7.22　Yolov3DetectionOutput 算子 MLUOpKernel 实现

中科院计算所

```
1  //输出形状推断及输出内存分配
2          int buffer_size = 255 * (inputHs_[0] * inputWs_[0] +
3                                  inputHs_[1] * inputWs_[1] +
4                                  inputHs_[2] * inputWs_[2]);
5          std::vector<int> buffer_shape = {batchNum_, buffer_size, 1, 1};
6          std::vector<int> output_shape(4, 1);
7          output_shape[0] = batchNum_;
8          output_shape[1] = 7 * maxBoxNum_ + 64;
9          Tensor* output;
10         Tensor* buffer;
11         TensorShape tf_output_shape {output_shape[0],output_shape[1],output_shape[2],
    output_shape[3]};
12         TensorShape tf_buffer_shape {buffer_shape[0],buffer_shape[1],buffer_shape[2],
    buffer_shape[3]};
13         OP_REQUIRES_OK(context, context->allocate_output(0, tf_output_shape, &output))
    ;
14         OP_REQUIRES_OK(context, context->allocate_output(0, tf_buffer_shape, &buffer))
    ;
```

图 7.23　Yolov3DetectionOutput 算子 MLUOpKernel 实现

```
1  //调用MLUStream层接口完成算子计算
2          if(output->NumElements() > 0 &&buffer->NumElements() > 0 ){
3            OP_REQUIRES_OK(
4                context,
5                stream->Yolov3DetectionOutput(
6                    context, input0, input1, input2, batchNum_, inputNum_,
7                    classNum_, maskGroupNum_, maxBoxNum_, netw_, neth_,
8                    confidence_thresh_, nms_thresh_, inputWs_.data(), inputHs_.data(),
9                    biases_.data(), output, buffer));
10         }else{
11           mlustream_exec->insert_unsupported_op(context, op_parameter);
12         }
13       }
14     private:
15     int batchNum_;
16     ......
17 };
18 }
```

图 7.24　Yolov3DetectionOutput 算子 MLUOpKernel 实现

```
1  #if CAMBRICON_MLU
2  #include "tensorflow/core/kernels/yolov3_detection_output_op_mlu.h"
3  namespace tensorflow {
4  #define REGISTER_MLU(T)                              \
5    REGISTER_KERNEL_BUILDER(                           \
6        Name("Yolov3DetectionOutput")                  \
7        .Device(DEVICE_MLU)                            \
8        .TypeConstraint<T>("T"),                       \
9        MLUYolov3DetectionOutputOp<T>);
10 TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);
11 #undef REGISTER_MLU
12 #endif  // CAMBRICON_MLU
13 }
```

图 7.25　Yolov3DetectionOutput 算子注册

中科院计算所

```
REGISTER_OP("Yolov3DetectionOutput")
    .Output("predicts: T")
    .Input("input0: T")
    .Input("input1: T")
    .Input("input2: T")
    .Attr("batchNum:int")
    .Attr("inputNum:int")
    .Attr("classNum:int")
    .Attr("maskGroupNum:int")
    .Attr("maxBoxNum:int")
    .Attr("netw:int")
    .Attr("neth:int")
    .Attr("confidence_thresh:float")
    .Attr("nms_thresh:float")
    .Attr("inputWs: list(int) = [13, 26,52]")
    .Attr("inputHs: list(int) = [13, 26,52]")
    .Attr("biases: list(float) = [116, 90, 156, 198, 373, 326, 30, 61, 62, 45, 59, 119,
      10, 13, 16, 30, 33, 23]")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext *c){
        return SetOutputForYolov3DetectionOutput(c);
    });
```

图 7.26　Yolov3DetectionOutput 算子注册

```
......
cc_library(
    name = "image",
    deps = [
        ":adjust_contrast_op",
        ......
        ":yolov3_detection_output_op",
    ],
)
......
tf_kernel_library(
    name = "yolov3_detection_output_op",
    prefix = "yolov3_detection_output_op",
    deps = [
        "//tensorflow/core:core_cpu",
        "//tensorflow/core:framework",
        "//tensorflow/core:lib",
        "//tensorflow/core:lib_internal",
        "//third_party/eigen3",
        ] + if_mlu([
            "//tensorflow/stream_executor:mlu_stream_executor"]),
)
```

图 7.27　TensorFlow 编译

中科院计算所

```
node {
  name: "Yolov3DetectionOutput"
  op: "Yolov3DetectionOutput"
  input: "conv_lbbox/BiasAdd"
  input: "conv_mbbox/BiasAdd"
  input: "conv_sbbox/BiasAdd"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
  ...
  attr {
    key: "nms_thresh"
    value {
      f: 0.449999988079
    }
  }
}
versions {
  producer: 38
}
```

图 7.28　YOLOv3 增加后处理节点

中科院计算所

```
def predict_bak(self, images):

    org_h = [0 for i in range(self.batch_size)]
    org_w = [0 for i in range(self.batch_size)]
    for i in range(self.batch_size):
        org_h[i], org_w[i], _ = images[i].shape

    image_data = utils.images_preporcess(images, [self.input_size, self.input_size])

    start = time.time()
    pred_sbbox, pred_mbbox, pred_lbbox = self.sess.run(
        [self.pred_sbbox, self.pred_mbbox, self.pred_lbbox],
        feed_dict={
            self.input_data: image_data,
            #self.trainable: False
        }
    )
    np.savetxt("pred_sbbox.txt",pred_sbbox.flatten())
    np.savetxt("pred_mbbox.txt",pred_mbbox.flatten())
    np.savetxt("pred_lbbox.txt",pred_lbbox.flatten())
    end = time.time()

    batch_bboxes = []
    for idx in range(self.batch_size):
        pred_bbox = np.concatenate([np.reshape(pred_sbbox[idx], (-1, 5 + self.
num_classes)),
                                    np.reshape(pred_mbbox[idx], (-1, 5 + self.
num_classes)),
                                    np.reshape(pred_lbbox[idx], (-1, 5 + self.
num_classes))], axis=0)
        bboxes = utils.postprocess_boxes(pred_bbox, (org_h[idx], org_w[idx]), self.
input_size, self.score_threshold)
        batch_bboxes.append(utils.nms(bboxes, self.iou_threshold))
    print("bbox num : ",len(batch_bboxes))
    for i in range(len(batch_bboxes)):
        print(batch_bboxes[i])
    exit(0)
    return batch_bboxes, (end - start)
```

图 7.29 原始推理

中科院计算所

```
def predict(self, images):
    ......
    start = time.time()
    bbox_raw = self.sess.run(
        self.bbox_raw,
        feed_dict={
            self.input_data: image_data,
            #self.trainable: False
        }
    )
    end = time.time()
    batch_bboxes = []
    num_batches = 1
    num_boxes = 1024 * 2
    predicts_mlu = bbox_raw.flatten()
    print("org_h[0], ",org_h[0])
    print("org_w[0], ",org_w[0])
    for batchIdx in range(num_batches):
        result_boxes = int(predicts_mlu[batchIdx * (64 + num_boxes * 7)])
        current_bboxes = []
        print("result_boxes :",result_boxes)
        print("x1, y1, x2, y2, score, classId")
        for i in range(result_boxes):
            #   batchId, classId, score, x1, y1, x2, y2
            batchId = predicts_mlu[i * 7 + 0 + 64 + batchIdx * (64 + num_boxes * 7)]
            classId  = predicts_mlu[i * 7 + 1 + 64 + batchIdx * (64 + num_boxes * 7)
]
            score    = predicts_mlu[i * 7 + 2 + 64 + batchIdx * (64 + num_boxes * 7)
]
            x1       = predicts_mlu[i * 7 + 3 + 64 + batchIdx * (64 + num_boxes * 7)
] * org_w[0]
            y1       = predicts_mlu[i * 7 + 4 + 64 + batchIdx * (64 + num_boxes * 7)
] * org_h[0]
            x2       = predicts_mlu[i * 7 + 5 + 64 + batchIdx * (64 + num_boxes * 7)
] * org_w[0]
            y2       = predicts_mlu[i * 7 + 6 + 64 + batchIdx * (64 + num_boxes * 7)
] * org_h[0]
            print(x1, y1, x2, y2, score, classId)
            # bbox = [xmin, ymin, xmax, ymax, score, class]
            bbox = [x1, y1, x2, y2, score, classId]
            current_bboxes.append(np.array(bbox))
        batch_bboxes.append(current_bboxes)
    print("bbox num : ",len(batch_bboxes))
    for i in range(len(batch_bboxes)):
```

图 7.30　添加后处理大算子

中科院计算所

7.2 文本识别 OCR-EAST

7.2.1 实验目的

本实验主要完成光学字符识别的的代表性算法——EAST 网络移植到智能处理器 DLP 上, 并进行性能优化与离线部署, 使读者可以借助 DLP 处理器完成完整的学字符识别任务.

因此, 本实验的主要目的包括:

1. 通过使用 Tensorflow 在 CPU 和智能处理器 DLP 上进行在线推理, 掌握使用 Tensorflow 编写光学字符识别应用;
2. 通过用智能编程语言实现 EAST 网络中的 Split+sub+concat 合并算子, 深入掌握智能编程语言的算子开发和优化方法;
3. 通过在 Tensorflow 中添加合并算子, 代替单算子, 掌握在 Tensorlfow 框架中添加合并算子的方法并优化性能;

实验时间预计为两周.

7.2.2 背景介绍

7.2.2.1 OCR

图 7.31 OCR 应用举例

从传说中的仓颉造字以来，文字或文本就为人类的发展的进步带来了重要贡献。到现在为止，文本依旧是包含丰富而准确信息的一个大类场景，因此自然场景中的文本检测和识别已成为计算机视觉和文档分析中重要且活跃的研究主题。尤其是近年来，尽管仍然存在各种挑战（例如，噪声，模糊，失真，遮挡和变化），但社区在这些领域的研究工作激增并取得了实质性进展。

在实际应用中，在新闻出版，邮政快递，金融服务等领域都广泛使用了文本识别算法。例如对于如图 7.31所示的针对会计领域的发票文本识别就大大减轻了人工比对的负担。

光学字符识别（Optical Character Recognition,OCR）是指对文本资料的图像文本进行文字识别, 获取文本资料版面信息的过程. OCR 算法的研究有很长的历史，在近一些年基于各种深度学习的算法逐渐成为重要的研究方向。

OCR 过程大致可以分为: 图像获取-> 图像预处理-> 文字检测-> 文字识别-> 输出.

中科院计算所

7.2.2.2　典型算法

文本检测和识别旨在检测和定位场景图像或视频中的文本，并自动对其进行识别。目前学术界已经开发了许多技术来检测和识别场景图像和视频中的文本。这些技术大致可以分为三大类：文本检测与定位，文本识别和端到端文本识别系统。文本检测与定位的目的是确定给定图像或视频中是否存在文本并将其定位。文本识别旨在将图像中定位出的文本转换为字符编码。而端到端文本识别系统将检测和识别结合为一个完整的框架。

- **文本检测**

现有的基于 CNN 的方法可以大致分为几类：基于 region proposal 的方法，基于 segmentation 的方法以及使用多任务学习的混合方法。

基于 region proposal 的方法一般都利用了基于 region 的方法和深度神经网络的优势。Zhang 等人[26] 提出了一种可以检测多种方向，语言和字体文本的框架。它主要由两个完全卷积网络（FCN）组成：一个用于文本区域的 salient map，另一个 FCN 用于预测每个字符的中心。但是，在某些情况下可能会出现误报和字符丢失的情况，例如对比度极低，曲率大，反射光强，文本行太封闭或字符之间的巨大间隙。另一个限制是计算效率低。

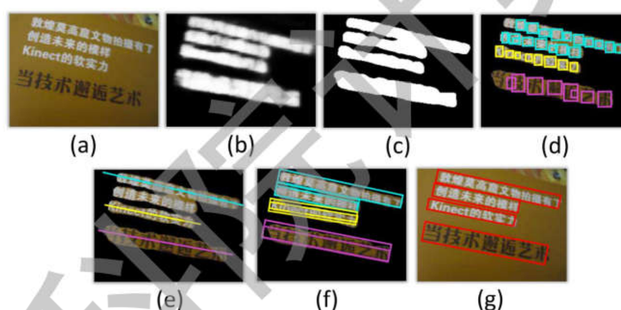

图 7.32　典型文本检测算法流程

如图7.32所示，这个算法主要分为以下几个步骤：

- （a）输入图像；
- （b）TextBlock FCN 预测的文本区域的 salient map；
- （c）文本块生成；
- （d）候选字符成分提取；
- （e）按组成部分预测的方向估计；
- （f）提取候选文本行；
- （g）算法的检测结果。

- **文本识别**

中科院计算所

对于 OCR 的文本识别，基于图像的序列识别问题，Shi[27] 等人提出了一种端到端的文本识别系统，称为卷积递归神经网络（CRNN），它由卷积层，递归层和转录组成。通过将 CNN 与 RNN 结合使用，CRNN 可以处理任意长度的序列，并且不依赖于其他词典。如图7.33所

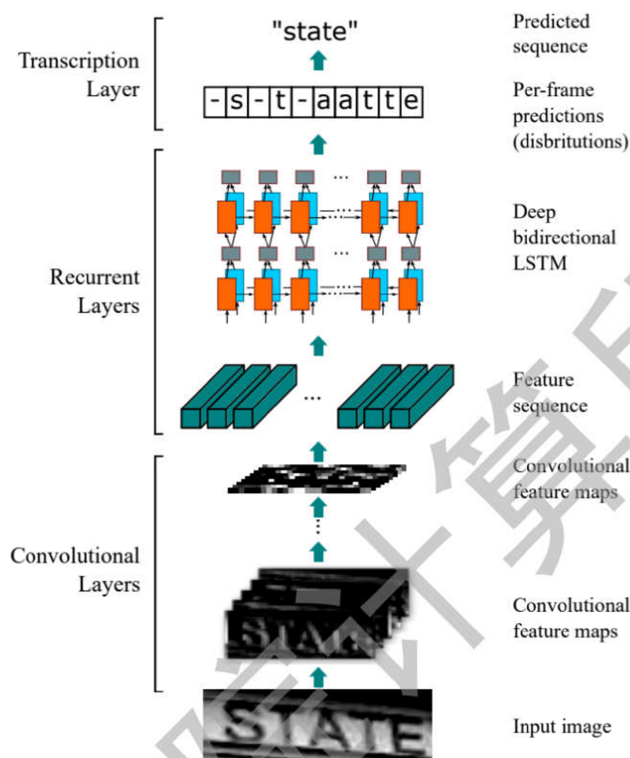

图 7.33　典型文本识别网络结构

示网络结构主要包括三个部分：

- 卷积层，从输入图像中提取特征序列；
- 循环层，预测每个帧的标签分布；
- 转录层，它将每帧的预测翻译成最终的标记序列。

- **端到端方法**

EAST 算法全名是 Efficient and Accuracy Scene Text, 它提出了一个快速而准确的轻量级场景文本检测 pipeline, 解决大多数 OCR 算法准确性和效率缺陷. 该 Pipeline 只有两个阶段: 第一阶段是基于 FCN 模型的文本框检测; 第二阶段是对生成的文本框（旋转或矩形）经过非极大值抑制得到最终结果, 从而免去其他中间步骤, 实现端到端训练.

- 网络结构

  EAST 的网络结构分为三大部分, 如图 7.34所示:

第一部分:Feature extractor stem(PVANet)

该部分是主干特征检测网络, 引入多尺度检测的思想, 提取不同尺寸卷积核下的特征并用于后期的特征组合, 以适应文本行尺度的变化. 作者采用了 PVANet 模型作为主干网络, 实际使用的时候也可以采用 VGG16 或者 Resnet;

第二部分:Feature-merging branch

该部分运用了 Unet 的思想, 主要是合并第一部分提取的特征, 通过池化和 Concat 来恢复尺寸.

其中, 特征合并主要通过逐层合并的方式, 首先将第一部分对应的特征图输入到一个池化层恢复尺度, 然后与当前层特征图进行 Concat, 再通过 1x1 卷积来减少通道数, 最后利用 3x3 卷积将局部信息融合以最终产生该合并阶段的输出;

第三部分:Output Layer

该部分输出分为 3 类:score_map,RBOX geometry,QUAD geometry:

Socre_map 代表此处是否有文字的可能性;

RBOX geometry 中的五个通道, 分别代表每个像素点到文本框边线的距离, 加上一个文本框的旋转角;

QUAD geometry 中的八个通道, 分别代表着每个像素点到文本框任意四边形的 xy 距离.

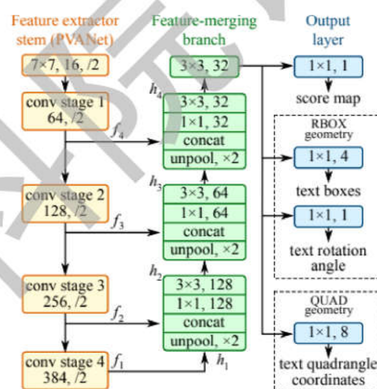

图 7.34 east 网络结构

- 后处理——局部感知 NMS

EAST 的后处理与通用目标检测类似, 需要将网络的结果经过非极大值抑制（NMS）来取得最终结果. 由于 EAST 的网络输出是成千上万个几何文本框, 需要根据阈值过滤, 然后将文本框的置信度得分加权平均, 得到合并后的文本框坐标.

将经过合并后的文本框经过通用的 NMS 处理, 详情可参考 7.1章节, 得到最终结果.

7.2.3 实验环境

本实验所涉及的硬件平台和软件环境如下:

中科院计算所

- 硬件平台: 硬件平台使用第?? 节介绍的云平台中的 DLP 硬件.
- 软件环境: 所涉及的 DLP 软件开发模块包括编程框架 TensorFlow、高性能库 CNML、运行时库 CNRT、编程语言及编译器、运行时库 CNRT 以及流式编程框架 CNStream.

7.2.4 实验内容

本实验主要是在 DLP 硬件上实现并优化以 EAST 网络模型为核心的目标检测应用. 针对用户从网络上下载的标准 EAST 模型, 经过模型量化并配置 DLP 相关参数, 则可以采用 DLP 上的定制 TensorFlow 版本来运行. 为了充分发挥 DLP 的计算能力, 进一步采用智能编程语言 BCL 实现 EAST 网络模型中的 Split+Sub+Concat 合并算子, 并将其添加至 TensorFlow 框架中, 代替原有的 Split、Sub、Concat 单算子. 考虑推理应用的实际需求, 采用 DLP 的流式框架编写完整目标检测应用, 基于离线模型进行部署. 具体实验内容包括:

1. EAST CPU 运行: 利用标准的与训练 CKPT 模型, 通过定制的 TensorFlow 版本在 CPU 上运行 EAST, 得到 Float16 精度的 PB 模型;
2. EAST DLP 运行: 将步骤 1 得到的 Float16 精度的 PB 模型进行 INT8 量化后, 通过定制的 TensorFlow 版本在 DLP 硬件上运行;
3. Split+Sub+Concat 合并算子的 BCL 实现: 采用 BCL 实现的 plit+Sub+Concat 合并运算;
4. 框架算子集成: 将用 BCL 实现的 Split+Sub+Concat 合并算子集成到 TensorFlow 框架中。

7.2.5 实验步骤

如前所述, 详细的实验步骤主要包括:EAST CPU 运行、EAST DLP 运行、Split+Sub+Concat 合并算子的 BCL 实现、框架算子集成等。

7.2.5.1 EAST CPU 运行

为了使得标准的 EAST 网络模型在 DLP 上运行, 需要通过运行 CPU 实现, 将 CKPT 预训练模型转换成 Float16 精度的 PB 模型. 具体步骤如下:

1. 获取开源数据集和 CKPT 模型:https://github.com/argman/EAST
2. 按照以下目录结构配置数据集和模型:

   数据集目录结构:

```
/home/Cambricon-MLU270/datasets/tensorflow_models/east
                |-- 2015Challenge4_Test_Task1_GT/
                |-- 2015ch4_test_images/
          |-- 2015ch4_training_images/
       |-- 2015ch4_training_localization_transcription_gt/
```

图 7.35 EAST 数据集目录结构

中科院计算所

```
/home/Cambricon-MLU270/models/tensorflow_models/east/east_icdar2015_resnet_v1_50_rbox
        |-- checkpoint
        |-- model.ckpt-49491.data-00000-of-00001
        |-- model.ckpt-49491.index
        |-- model.ckpt-49491.meta
```

图 7.36　EAST 模型目录结构

模型目录结构:

CPU 运行之前，需要 MODEL_PATH 指定到 east_icdar2015_resnet_v1_50_rbox 下，DATASET_PATH 指定到 2015ch4_test_images 下,GT_PATH 指定到 2015Challenge4_Test_Task1_GT 下。

3. DLP 软件环境中已经适配了 EAST CPU 运行环境和 DLP 运行环境, 需要进入 tensorflow_models/cambricon_examples/EAST/ 目录下;
4. 安装推理所需软件包: 一些必备的 Python 安装包在所提供的云平台软件环境中已经安装完成, 但对于每个综合实验的特有依赖还需进行单独安装, 使用 *pip* 命令完成依赖的安装与升级:pip install -r requirements.txt
5. 模型推理和转换: 执行如下命令完成在 CPU 上的推理, 并将 CKPT 模型转换为 Float16 精度的 PB 模型:./run_cpu.sh
6. 整理模型路径: 生成的 PB 模型在./cpu_pb 中, 需要将得到的 east.pb 拷贝至/home/Cambricon-MLU270/models/tensorflow_models/EAST/east.pb

7.2.5.2　EAST DLP 运行

下面介绍如何将7.2.5.1小节得到的 Float16 精度 PB 模型量化为 INT8 精度的 PB 模型.DLP 软件环境提供 fppb_to_intpb 工具来进行模型量化.

1. 进入 tensorflow/cambricon_examples/tools/fppb_to_intpb 目录;
2. 生成数据文件: 按照模型所需要的数据集以及数据集的具体路径来修改 generate_image_list.py 文件中的 image_libs_map, 具体如图7.37所示, 来生成相应的数据文件.
3. 生成量化配置文件: 如第 **??**所述, 在进行量化前需设置相应的配置文件. 以指定量化相关的参数.DLP 软件环境提供了 generate_ini.py 来生成指定模型的量化配置文件. 针对 EAST, 需要将 generate_ini.py 中的 model_name 进行修改, 即指定要生成的配置文件是针对 EAST 的:models_name = ['EAST']

   执行 DLP 定制化的 PB 模型量化脚本 generate_ini.py 后会生成 config 文件夹, 其中包含了 EAST_naive_int8.ini 参数配置文件, 该配置文件包含如图7.38所示的信息. 同时为了规范模型路径, 需要修改 save_model_path;

   而提高数据集整体的量化质量. 但是对于不存在异常值, 数据分布紧凑的情况下, 不建议使用该算法, 比如网络权值的量化. 其中 device_mode 可以设置输出 pb 所有节点的 device, 有三种配置方式:clean、mlu 和 origin. 其中,mlu 将输出 pb 的所有节点的 device

```
import os
from os.path import isfile, join

max_image_count = 100
image_libs_map = {
      "east":join(os.environ.get( "TENSORFLOW_MODELS_DATA_HOME" ), "east/2015
     ch4_training_images")
      }
if __name__ == "__main__":
  for image_lib, image_path in image_libs_map.items():
    n = 0
    with open("image_list_{}".format(image_lib), "w") as image_list:
      for root, dirs, files in os.walk(image_path):
        for f in files:
          if n >= max_image_count:
            break
          if f.endswith("jpg"):
            image_list.write(join(image_path, f) + "\n")
            n = n + 1
```

图 7.37 EAST 量化生成数据文件

都设置为 MLU, 即都在 MLU 上运行;clean 将输出 pb 的所有节点的 device 清除, 运行时根据算子注册情况自动选择可运行的设备;origin 则使用和输入 pb 一样的设备指定.

4. 执行命令:python fppb_to_intpb.py 完成模型量化最终得到 east_int8.pb.

完成 CPU 侧的推理后, 接下来需要将其移植在 DLP 上, 以加快其推理速度. 此时我们只需仿照第 7 章编程语言算子实验中的 DLP 推理移植部分.DLP 软件栈中已经完成了 EAST 的移植, 只需要将 MODEL_PATH 指定到 east_int8.pb, 执行得到精度和性能数据:

./run.sh 16 MLU270 int8 100 1

7.2.5.3 BANGC 代码实现

1. 设计原则

在使用 BCL 设计 Split+Sub+Concat 合并算子的时候, 需要考虑以下 BCL 设计原则:

(a) 能够处理的数据来源:GDRAM,NRAM,SRAM 中的一种

(b) 能够将数据存放到:GDRAM,NRAM,SRAM 中的一种

(c) 每个 Core 都有一块片上存储空间 NRAM, 其大小约为 512KB

(d) 每个 Cluster 上分配一块 SRAM, 其大小 2MB, 并且单核任务不能 持 SRAM 的使 ,UNION1 任务和单核任务不 持 SRAM 通信 UNION2 以上任务才 持 SRAM 通信

保证算子正确性的情况下, 需要考虑以下四种优化方向:

(a) 对访存进 优化以充分利 上存储, 比如 NRAM 复用

(b) 对计算逻辑进 优化以充分削减计算量, 充分利 张量运算器

```
[preprocess]
mean = 0, 0, 0    #均值,顺序依次为 mean_r、 mean_g、 mean_b
std = 1.0        #方差
color_mode = rgb        #网络的输入图片是 rgb 还是 bgr
crop = 544, 544         #前处理最终将图片处理为 416 □ 416 大小
calibration = east_preprocess_cali      #校准数据读取及前处理的方式,可以根据需求进行自 定
    义, [preprocess] 和 [data] 中定义的参数均为 calibration 的输入参数

[config]
activation_quantization_alg = naive     #输入量化模式,可选 naive 和 threshold_search,
    naive 为基础模式, threshold_search 为阈值搜索模式

device_mode = clean               #可选 clean、mlu 和 origin ,建议使用 clean,使用 clean
    生成的模型在运行时会自动选择可运行的设备

use_convfirst = False                  #是否使用 convfirst
quantization_type = int8               #量化位宽,目前可选 int8 和 int16
debug = False                        #是否为debug模式
weight_quantization_alg = naive        #权值量化模式,可选 naive 和 threshold_search,
    naive 为 基础模式, threshold_search 为阈值搜索模式

int_op_list = Conv, FC, LRN             #要量化的 layer 的类型,目前只能量化 Conv、FC 和
    LRN
channel_quantization = False            #是否使用分通道量化,目前不支持为 True

[model]
output_tensor_names = feature_fusion/Conv_7/Sigmoid:0, feature_fusion/concat_3:0  #输出
    Tensor 的名字,可以是多个,以逗号隔开
original_models_path = /home/Cambricon-MLU270/models/tensorflow_models/east/east.pb  #输
    入pb
save_model_path = /home/Cambricon-MLU270/models/tensorflow_models/east/east_int8.pb    #
    输出pb
input_tensor_names = input_images:0                  #输入 Tensor 的名字,可以是多个,以
    逗号隔开

[data]
num_runs = 2      #运行次数
```

图 7.38　EAST 量化配置文件

(c) 充分利　多核并　（计算任务拆分）来提升并　度以及隐藏访存延迟

(d) 充分利　编译器提供的各种编译优化选项, 比如 O2 或者 O3 编译优化

2. 设计原理

首先, 我们需要分析输入数据的规模, 通过开源模型可视化软件 Netron 查看 PB 模型, 可以确定整个网络模型的结构, 并在 DLP MLU 执行过程中, 打印出 input_images 的输出维度为 (1,672,1280,3), 即是 Split 的输入维度;

由于 BCL 使用的数据类型是 Half 类型, 占据 2 个字节, 所以计算出输入规模是 1x672x1280x3x2/1024 在充分利用多核并行的情况下, 将数据分块到每个 core, 每块 NRAM 为 5040/16=315KB, 在 512KB 的限制内;

使用 BangC 实现 Split 的时候, 需要注意, 输入数据的摆数是 NHWC,C 方向的数据是连续的. 如果在 C 方向进行 Split, 则需要先转置为 NCHW, 或者使用 __bang__maskmove 来进行 Split, 再去做 Sub, 这样会增加计算耗时或者 NRAM 的使用. 而最简洁的方法是

中科院计算所

使用 __bang_cyclesub, 对 C 方向进行 cyclesub, 省略掉了 Split 和 Concat 部分, 大大提升效率, 减少 NRAM 的使用. 整体设计流程图如图 7.39所示:

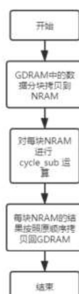

图 7.39　SBC 运行流程

下面以图7.40中的代码为例子进行说明, 介绍 Split+Sub+Concat 合并算子的接口介绍:

```
1        __mlu_entry__ void SBCKernel(half* input_data_,
2        half* output_data_,
3        int batch_num_)
4
```

图 7.40　Split+Sub+Concat 合并算子接口

input_data_: 输入数据存放的数据地址;

output_data_: 输出数存放的数据地址;

batch_num_:Batchsize 的规模.

3. 实现解析

   代码实现分为两部分:Kernel 函数实现的.mlu 和 cnrt 实现的.cc

7.2.5.4　Tensorflow 算子添加

本节介绍如何将实现的 BANGC kernel 函数集成至 Tensorflow, 为 EAST 添加 Split+Sub+Concat 合并算子. 添加 MLU 算子主要分为以下几个步骤, 步骤之间有承接关系:

1. CNPlugin 编译

   如前所述,CNML 通过 PluginOp 相关接口提供了用户自定义算子和高性能库已有算子协同工作（如算子融合）的机制. 因此, 在完成 Split+Sub+Concat 算子的开发后, 可以利用 PluginOp 相关接口封装出方便用户使用的接口（主要包括 PluginOp 的创建、计算和销毁等接口）, 使得用户自定义算子和高性能库已有算子有一致的编程接口和模式.

   图7.43给出了 PluginOp 接口封装的部分示例代码, 主要包括 Create、Compute 和 Destroy 三类函数的具体实现.

   -creat 函数:

中科院计算所

```
#define HWC_SPLIT ((((HEIGHT*WIDTH/16) - 1) / ALIGN_SIZE + 1) * ALIGN_SIZE)*CHANNELS
#define CHANNELS 3
#define HEIGHT 672
#define WIDTH 1280
#define ALIGN_SIZE 64
#define DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))

#include "mlu.h"

__mlu_entry__ void SBCKernel(half* input_data_,half* output_data_, int batch_num_){

    int batch_num = batch_num_;

    //struct timeval start;
    //struct timeval end;
    //gettimeofday(&start, NULL);

    __nram__ half split_sub_concat[HWC_SPLIT];
    __nram__ half tmp0[192];

    //多核拆分
    int core_loop = 16 / taskDim;

    //循环创建 cycle_sub mask
    for (int i =0;i<192;i++){
        tmp0[i*3] = 123.68;
        tmp0[i*3+1] = 116.78;
        tmp0[i*3+2] = 103.94;
    }

    for (int i = 0; i<batch_num; i++){
        for (int j = 0; j < core_loop; j++){
            //数据拆分至每个 Core 的 NRAM
            __memcpy(split_sub_concat ,input_data_ + (j * taskDim + taskId) * HWC_SPLIT + i * DATA_COUNT, HWC_SPLIT*sizeof(half),GDRAM2NRAM);
            //cycle_sub 代替 split+sub
            __bang_cycle_sub(split_sub_concat,split_sub_concat,tmp0,HWC_SPLIT,192);
            //按顺序拷贝回GDRAM
            __memcpy(output_data_ + (j * taskDim + taskId) * HWC_SPLIT + i * DATA_COUNT ,split_sub_concat,HWC_SPLIT*sizeof(half),NRAM2GDRAM);
        }
        __sync_all();
    }

    //计算耗时
    //gettimeofday(&end, NULL);
    //uint32_t time_usec = (uint32_t)end.tv_usec - (uint32_t)start.tv_usec;
    //printf("Hardware Total Time: %u us\n", time_usec);
    //printf("batch_size: %d us\n", batch_num);
    //printf("core_num: %d us\n", taskDim);

}
```

图 7.41　EAST BANGC 算子实现

-Compute 函数:

-Destroy 函数:

2. 算子注册

中科院计算所

算子注册在 tensorflow/core/ops 目录下找到对应的 Op 注册文件, 对于 Split+Sub+Concat 合并算子, 在 math_ops.cc 中注册. 如图7.45,Split+Sub+Concat 合并算子包含一个输入节点 input 和一个输出节点 output, 输入输出的数据类型均用 T 表示即该算子的输出与输入数据类型一致. 其中,Attr(“T:type”) 表示 T 允许的数据类型为 type, 也就是 TensorFlow 支持的所有数据类型.

3. MLUlib

   MLUlib 层主要是对 CNML 和 CNRT 接口的封装, 该部分代码在 tensorflow/stream_executor/mlu/mlu_ 目录下, 需要注意的是 MLULIB 层禁止引用 TensorFlow 的头文件, 目前只使用了 tensorflow::Status 类方便做返回值处理. 添加 Split+Sub+Concat 合并算子需要对该目录下的 mlu_lib_ops.cc 和 mlu_lib_ops.h 文件进行修改, 示例代码如图7.46所示.

4. MlUops

   MLUOps 层主要是使用 MLULib 层封装好的 API 完成算子的实现, 代码文件在 tensorflow/stream_executor/mlu/mlu_api/ops 下, 需要在目录下的 mlu_ops.h 文件中添加新算子类的声明, 如图7.47所示.

   同时, 需要新建一个 sbc.cc 文件来添加 Split+Sub+Concat 合并算子的实现, 其中包含 Creat 方法和 Compute 方法, 如图7.48所示.

   Create 方法的职责是将创建好的 baseop 指针调用 base_ops_.push_back(op_ptr); 储存起来:Compute 方法的职责在于调用 lib 层的 compute 函数进行计算, 目前算子实现为同步方式, 因此需要调用 SyncQueue.

5. MLUStream

   MLUStream 层负责 MLULOps 算子类的实例化, 其接口都定义在 tensorflow/stream_executor/mlu/mlu 中. 可以把 MLUStream 层的算子分为 2 类: 通用模版算子与特例化算子. 对于 Split+Sub+Concat 合并算子来说, 其属于通用模版算子.

   通用模版算子满足以下三个条件:MLU 算子的所有输入均来自 OpKernelContext;MLU 算子的所有输出顺序须与 OpkernelContext 的输出顺序一致;MLUTensor 可以被 CreateMLUTensorFromTensor 创建, 即 MLUTensor 的形状、数据类型与 TensorFlowtensor 一致. 如图7.49

6. MLUOpKernel

   MLUOpKernel 层负责对算子抽象定义, 其继承了 OpKernel 类, 其使用方法与 OpKernel 基本一致. 对于每个 MLU 算子均需要实现其构造函数与 ComputeOnMLU 方法。MLUOpKernel 实现的主要功能有参数检查、参数处理、输出形状推断及输出内存分配、调用 MLUStream 层接口完成算子计算。需要在 cwise_op_sbc_mlu.cc 文件中进行注册和 cwise_op_sbc_mlu.h 实现。在下图**??**和7.51中的代码分别显示了这四个不同的步骤

7. BUILD 为了将该算子最终集成在 TensorFlow 中, 编译时还需要在 tensorflow/core/kernels/BUILD 中添加以下信息.

中科院计算所

7.2.5.5 DLP 执行

在将 BCL 实现的 Split+Sub+Concat 合并算子成功集成到框架后, 需要修改之前转换完成的 int8pb 模型, 在模型中增加 Split+Sub+Concat 合并算子, 这一步可以使用编程框架机理章节中介绍的 pb 与 pbtxt 相互转换工具实现. 具体在 pbtxt 中用 Split+Sub+Concat 节点7.52替换原生 Split、Sub、Concat 节点7.53

7.2.6 实验评估

相比其它两个综合实验，EAST 实验困难度较低，因此难度系数分为 1.0。学生最终得分为 1.0 乘以相应的得分。

- 60 分标准：完成 BANGC 算子的实现与 CNRT 测试，CNRT 测试时延时低于 30ms。
- 70 分标准：在 60 分的基础上完成 TensorFlow 的算子集成，包括 cnplugin 集成与 TensorFlow 的编译，完成 pb 模型的修改操作。执行测试脚本时，修改后的模型精度与修改前误差在 5% 以内，且延时不高于原始的模型。
- 80 分标准：在 70 分的基础上，执行测试脚本时，修改后的模型精度与修改前误差在 1% 以内，且延时至少优于原始的模型 10ms。
- 90 分标准：在 80 分的基础上，执行测试脚本时，修改后的模型精度与修改前误差在 0.1% 以内，且延时至少优于原始的模型 25ms。
- 100 分标准：在 90 分的基础上，执行测试脚本时，修改后的模型精度于修改前完全一致，且延时至少优于原始的模型 40ms。

7.2.7 实验思考

1. 为什么 BCL 算子相比 Split+Sub+Concat CNML 单算子有显著优势？

中科院计算所

```
...
#include "macro.h"
#include "cnrt.h"
#include "utils.h"

#define CHANNELS 3
#define HEIGHT 672
#define WIDTH 1280
#define BATCH_SIZE 1
#define DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))

using namespace std;
typedef unsigned short half;

extern "C" {
    void SBCKernel(half* input_data_,half* output_data_ ,int batch_num_, int core_num_
   );
}

int main() {
    const int data_count = DATA_COUNT*BATCH_SIZE;
    int batch_num_ = BATCH_SIZE;
    int core_num_ = NUM_MULTICORE;
    const int channels_ = CHANNELS;
    const int height_ = HEIGHT;
    const int width_ = WIDTH;

    //开辟CPU 内存
    ...

    //读取数据文件
    ...

    //初始化设备
    ...

    //选择 UNION 模式
    switch (core_num_) {
    ...
    }

    //float2half
    ...

    // Passing param
    ...

    // create cnrt Notifier
    ...

    // hardware time
    cnrtPlaceNotifier(Notifier_start, pQueue);
    CNRT_CHECK(cnrtInvokeKernel_V2(...));
    cnrtPlaceNotifier(Notifier_end, pQueue);
    CNRT_CHECK(cnrtSyncQueue(pQueue));

    gettimeofday(&end, NULL);
    float time_use = ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.
   tv_usec))/1000.0;
    printf("time use: %.3f ms\n", time_use);

    float* output_tmp = (float*)malloc(data_count * sizeof(float));
    ...

    // save data
    FILE* mluOutputFile = fopen("./mluoutput.txt", "w");
    for (int i = 0; i < data_count; i++) {
```

中科院计算所

```
cnmlStatus_t cnmlCreatPluginSBCOpParam(
  cnmlPluginSBCOpParam_t *param,
  int batch_num_
){
  *param = new cnmlPluginSBCOpParam();
  (*param)->batch_num_ = batch_num_;

  return CNML_STATUS_SUCCESS;
}

cnmlStatus_t cnmlDestroyPluginSBCOpParam(
  cnmlPluginSBCOpParam_t *param
  ){
  delete (*param);
  *param = nullptr;

  return CNML_STATUS_SUCCESS;
}

```

图 7.43　CNPlugin 添加头文件接口声明

```
cnmlStatus_t cnmlCreatePluginSBCOp(
  ...
  ){

  int input_num = 1;
  int output_num = 1;

  void** InterfacePtr;
  InterfacePtr = reinterpret_cast<void**>(&SBCKernel);

  cnrtKernelParamsBuffer_t params;
  ...

  cnmlCreatePluginOp(...);

  cnrtDestroyKernelParamsBuffer(params);
  return CNML_STATUS_SUCCESS;
}

cnmlStatus_t cnmlComputePluginSBCOpForward(
  ...
  ){

  cnrtInvokeFuncParam_t compute_forw_param;
  ...
  cnmlComputePluginOpForward_V3(...);

  return CNML_STATUS_SUCCESS;
}


```

图 7.44　CNPlugin 算子注册

中科院计算所

```
REGISTER_OP("SBC")
.Input("input: T")
.Output("output: T")
.Attr("T:type")
.SetShapeFn(shape_inference::UnchangedShape);

```

图 7.45 EAST 算子注册

```
tensorflow::Status CreateSBCOp(MLUBaseOp** op, MLUTensor* input,
                               MLUTensor* output,int batch_num_) {

MLUTensor* inputs_ptr[1] = {input};
MLUTensor* outputs_ptr[1] = {output};

CNML_RETURN_STATUS(cnmlCreatePluginSBCOp(op, inputs_ptr, outputs_ptr,batch_num_));
}

tensorflow::Status ComputeSBCOp(
                    MLUBaseOp* op,
                    void* input,
                    void* output,
                    MLUCnrtQueue* queue) {

void* inputs_ptr[1] = {input};
void* outputs_ptr[1] = {output};

CNML_RETURN_STATUS(cnmlComputePluginSBCOpForward(
                            op, inputs_ptr, 1, outputs_ptr, 1,queue));
}

tensorflow::Status CreateSBCOp(MLUBaseOp** op, MLUTensor* input,
                          MLUTensor* output,int batch_num_);

tensorflow::Status ComputeSBCOp(
                      MLUBaseOp* op,
                      void* input1,
                      void* output,
                      MLUCnrtQueue* queue);

```

图 7.46 EAST MLULib 层

```
DECLARE_OP_CLASS(MLU...) ;
DECLARE_OP_CLASS(MLUSBC);
DECLARE_OP_CLASS(MLU...) ;

```

图 7.47 east-mluops1

中科院计算所

```
/*Copyright 2018 Cambricon*/
#if CAMBRICON_MLU

namespace stream_executor {
namespace mlu {
namespace ops {

Status MLUSBC::CreateMLUOp(std::vector<MLUTensor *> &inputs,
                           std::vector<MLUTensor *> &outputs, void *param) {
  TF_PARAMS_CHECK(inputs.size() > 0, "Missing input");
  TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");

  MLUBaseOp *op_ptr = nullptr;
  MLUTensor *input = inputs.at(0);
  MLUTensor *output = outputs.at(0);

  int batch_num_ = *((int*)param);

  MLULOG(3) << "CreateSBCOp"
            << ", input: " << lib::MLUTensorUtil(input).DebugString()
            << ", output: " << lib::MLUTensorUtil(output).DebugString();

  TF_STATUS_CHECK(lib::CreateSBCOp(&op_ptr, input, output,batch_num_));

  base_ops_.push_back(op_ptr);

  return Status::OK();
}

Status MLUSBC::Compute(const std::vector<void *> &inputs,
                       const std::vector<void *> &outputs, cnrtQueue_t queue) {
  void *input = inputs.at(0);
  void *output = outputs.at(0);
  TF_STATUS_CHECK(lib::ComputeSBCOp(base_ops_.at(0), input, output, queue));

  TF_CNRT_CHECK(cnrtSyncQueue(queue));

  return Status::OK();
}

}  // namespace ops
}  // namespace mlu
}  // namespace stream_executor

#endif  // CAMBRICON_MLU


```

图 7.48 east-mluops2

```
Status SBC(OpKernelContext* ctx, Tensor* input, Tensor* output,int batch_size) {
  return CommonOpImpl<ops::MLUSBC>(ctx, {input}, {output},static_cast<void*>(&
 batch_size));
}

```

图 7.49 east-mlustream

```
#ifndef TENSORFLOW_CORE_KERNELS_CWISE_OP_POWER_DIFFERENCE_MLU_H_
#define TENSORFLOW_CORE_KERNELS_CWISE_OP_POWER_DIFFERENCE_MLU_H_
#if CAMBRICON_MLU

namespace tensorflow {
template <typename T>
class MLUSBCOp : public MLUOpKernel {
public:
  explicit MLUSBCOp(OpKernelConstruction* ctx) :
        MLUOpKernel(ctx) {}

  void ComputeOnMLU(OpKernelContext* ctx) override {

    if (!ctx->ValidateInputsAreSameShape(this)) return;
    auto* stream = ctx->op_device_context()->mlu_stream();
    auto* mlustream_exec = ctx->op_device_context()->mlu_stream()->parent();
    Tensor input = ctx->input(0);

    // 参数检查与处理
    const Tensor& a = ctx->input(0);
    int batch_size = a.dim_size(0);

    // MLU_OP_CHECK_DIM(input, 4, mlustream_exec, ctx);

    string op_parameter = ctx->op_kernel().type_string()
                          + "/" + input.shape().DebugString();
    MLU_OP_CHECK_UNSUPPORTED(mlustream_exec, op_parameter, ctx);

    TensorShape shape = TensorShape(input.shape());
    //std::cout<<shape<<std::endl;

    //输出形状推断及输出内存分配
    Tensor* output;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, shape, &output));

    // 调用MLUStream层接口完成算子计算
    OP_REQUIRES_OK(ctx, stream->SBC(ctx, &input, output, batch_size));

  }
};

}  // namespace tensorflow

#endif  // CAMBRICON_MLU
#endif  // TENSORFLOW_CORE_KERNELS_CWISE_OP_SQUARED_DIFFERENCE_MLU_H_

```

图 7.50　east-MLUOpKernel

中科院计算所

```
namespace tensorflow {
#if CAMBRICON_MLU
#define REGISTER_MLU(T)                                              \
  REGISTER_KERNEL_BUILDER(Name("SBC")                                \
                              .Device(DEVICE_MLU)                    \
                              .TypeConstraint<T>("T"),               \
                          MLUSBCOp<T>);
TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);
#undef REGISTER_MLU
#endif
}

```

图 7.51　east-MLUOpkernel2

中科院计算所

```
node {
  name: "concat"
  op: "SBC"
  input: "input_images"
  batch_num_: "1"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
}
```

图 7.52　east-pbtxt1

```
node {
  name: "split/split_dim"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_INT32
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_INT32
        tensor_shape {
        }
        int_val: 3
      }
    }
  }
}
……
node {
  name: "concat"
  op: "ConcatV2"
  input: "sub"
  input: "sub_1"
  input: "sub_2"
  input: "concat/axis"
  attr {
    key: "N"
    value {
      i: 3
    }
  }
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "Tidx"
    value {
      type: DT_INT32
    }
  }
}
```

图 7.53 east-pbtxt2

中科院计算所

7.3 自然语言处理-BERT

7.3.1 实验目的

本实验主要完成将自然语言处理的代表性算法——BERT 网络移植到智能处理器 DLP 上，并进行性能优化与比较，使读者可以借助 DLP 处理器完成完整的目标检测任务。

因此，本实验的主要目的包括：

1. 通过用智能编程语言实现 BERT 模型中的 BatchMatMulV2 算子，深入掌握智能编程语言的算子开发和优化方法；
2. 通过在 TensorFlow 中添加大算子，掌握在 TensorFlow 框架中添加融合算子的方法；
3. 通过使用 TensorFlow 进行在线推理，掌握使用 TensorFlow 编写目标检测应用并在典型 DLP 上进行优化的方法；
4. 通过与 BANGC 大算子实现的比较，体会 DLP 相比 GPU 和 CPU 的优势。

实验时间预计为两周。

7.3.2 背景介绍

7.3.2.1 NLP 介绍

总的来说，NLP 的目标是要让计算机正确处理人类的自然语言。常见的 NLP 问题一般包括阅读理解，问答系统，对话系统，文本摘要和文本分类等。与一般的分类或者检测网络不同，NLP 相关的深度学习技术一般都关注于字符之间的序列关系。

斯坦福问答数据集（Stanford Question Answering Dataset，SQuAD）是一种阅读理解数据集，由工作人员在一组 Wikipedia 文章上提出的问题组成，其中每个问题的答案是对应阅读文章或问题的一段文本，而问题本身可能是无法回答的。例如对于文本：

James Bryant Conant led the university through the Great Depression and World War II and began to reform the curriculum and liberalize admissions after the war.

有对应问题：

What was the name of the leader through the Great Depression and World War II?

算法应当正确回答：

James Bryant Conant

7.3.2.2 NLP 常见算法、原理

1. **LSTM** 关于 LSTM 的相关原理，在智能计算系统理论书中已经有所介绍。其是在循环神经网络 (RNN) 的基础上，将 RNN 的隐层单元用 LSTM(Long Short Term Memory) 单元来代替，使其能保留更长的时间信息，减缓梯度消失的现象。相较于卷积神经网络，循环神经网络有存储信息的能力，可有效处理序列数据。此外由于其权值共享的特性，循环神经网络的参数一般要少许卷积神经网络，其结构图如下图 7.54所示。

中科院计算所

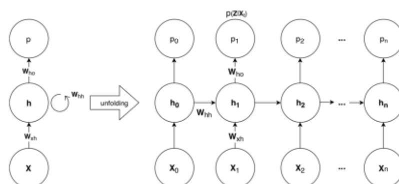

图 7.54　循环神经网络结构图

2. **Attention** Attention 是一种用于提升基于 RNN（LSTM 或 GRU）的 Encoder + Decoder 模型效果的的机制，一般也称为注意力机制。注意力机制现已广泛应用于机器翻译、语音识别、图像标注（Image Caption）等领域。Attention 之所以受欢迎的原因在于其给模型赋予了区分辨别的能力，例如，在机器翻译、语音识别应用中，为句子中的每个词赋予不同的权重，使神经网络模型的学习变得更加灵活（soft）。具体来说，注意力机制可以帮助模型对输入的每个部分赋予不同的权重，抽取出更加关键及重要的信息，使模型做出更加准确的判断，同时不会对模型的计算和存储带来更大的开销。

   对于一般的 Seq2Seq 模型，其通常在编码阶段（Encoder）把输入编码成一个固定的长度，这样做会带来两个问题：当输入序列很长时，模型性能会有影响；其对输入的每个元素赋予相同的权重，无法体现重要程度。而注意力机制一定程度上可以解决此问题，其会对输入的每一个元素计算出其对输出端各个元素之间的权重，表示源端第 i 个元素对目标端各元素的影响程度。

   常见的注意力机制可分为许多种类，常见的有 soft Attention 和 Hard Attention；Global Attention 和 Local Attention；以及 Self Attention 等。其中 Self Attention 为 Transformer 的重要部分，此处简单对齐作出介绍。

   Self Attention 与传统的 Attention 机制不同：传统的 Attention 如上所述是基于输入端和输出端的隐层单元计算 Attention 的，得到的结果是源端的每个词与目标端每个词之间的依赖关系。而 Self Attention 是分别对输入端与输出端自身进行计算，捕捉输入端或输出端自身的词与词之间的依赖关系；然后再把输入端得到的 self Attention 加入到输出端得到的 Attention 中，捕捉输入端和输出端词与词之间的依赖关系。因此 Self Attention 最终得到的信息中不仅包含输入与输出的依赖关系，还包含自身词与词之间的依赖关系，其效果要好于传统的注意力机制。

3. **Transformer** 如图 7.55所示，其编码部分输入 (*inputs*) 对应训练的数据集，解码部分输入 (outputs-shifted right) 对应数据集的标签。

   像大多数 Seq2Seq 模型一样，Transformer 也分为 Encoder 和 Decoder 两个部分组成（分别对应图 7.55中的左半部分和右半部分）。

   对于 Encoder 来说，其由 6 个相同的 Layer 组成（即 N=6），每个 Layer 又由两个 Sub-Layer 组成，分别是 multi-head self-attention mechanism 和 fully connected feed-forward network。其中每个 Sub-Layer 还增加了 Residual Connection 和 Layer Normalisation。

   对于 Decoder 来说，其在 Encoder 的基础上，增加了额外的 Attention Sub-Layer。该子

中科院计算所

层同时增加了 Mask 操作，以此确保预测第 i 个位置时仅可以获取到 i 之前的输出（不会接触到未来的信息）。在训练时其输入为 Encoder 的输出，以及编码器输入对应的标签（整体向右偏移一位），由于标签是始终存在的，因此训练过程是可以做到并行的。在推理时，Decoder 没有 ground truth 作为输入，其输入来自上一个位置的输出，也就意味着需要逐个位置进行解码，无法并行。

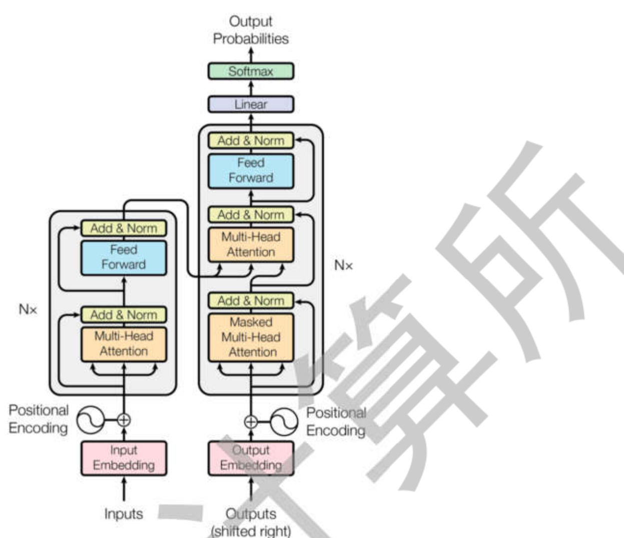

图 7.55　Transformer 结构图 (1)

从图 7.55和图 7.56的结构示意图可以发现，Encoder 和 Decoder 是层叠多个 Multi-Head Attention 单元构成，而每一个 Multi-Head Attention 单元由多个结构相似的 Scaled Dot-Product Attention 单元组成。Self Attention 也是在 Scaled Dot-Product Attention 单元里面实现的，如上图左图所示，首先把输入经过不同的线性变换分别得到 Q、K、V（Q、K、V 都来自于相同的输入）。然后把 Q 和 K 做 dot Product 相乘，得到输入词与词之间的依赖关系，然后经过尺度变换（scale）、掩码（Mask）和 Softmax 操作，得到最终的 Self Attention 矩阵。方程式表达如下，其中 key(K) 意味着原始隐层单元的输出，value(V) 代表每一个隐层单元之间的关系，Q 为 outputs-shifted right 部分的输出。

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

4. **BERT** BERT 的全称为 Bidirectional Encoder Representation from Transformers。其采用了 Transformer 的 Encoder 部分，并且进行双向的 Block 连接，与 Transformer 相比，不仅可获取到语句之前的信息，还可获取到未来的信息。其结构图如图 7.57所示。

   图中的黄色部分为输入的嵌入（Embedding）表示，其由三种不同的 Embedding 求和而成，分别为 Token Embeddings，Segment Embeddings 以及 Position Embeddings。Token Embeddings 是模型输入中每个元素的词向量表示；Segment Embeddings 用来区分输入的前后两个语句；Position Embeddings 含义与 Transformer 相同，但 Transformer 中使用

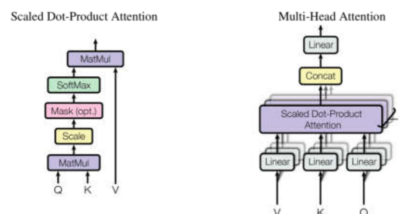

图 7.56　Transformer 结构图 (2)

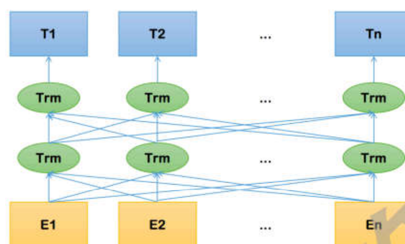

图 7.57　BERT 结构图

中科院计算所

不同频率的三角函数直接计算得出，而 BERT 是通过训练学习得到的。图 7.57中每一个黄色框均代表模型输入的一个元素对应的 Embedding 表示。紧接着将 Embedding 输入到 Transformer 的编码（Encoder）部分，由于 Encoder 没有对 Multi-Head Attention 进行 Mask 操作，这也是 BERT 能获取到双向信息的原因。经过多层串联的 Transformer 后即得到最终的输出（图 7.57中的蓝色框部分）。

在训练时，BERT 包含两个预训练子任务。第一个子任务为 Masked Language Model，通过随机遮挡每一个句子中 15% 的词，用其上下文来做预测，在计算损失函数时只计算被遮挡掉的输入元素。第二个子任务为 Next Sentence Prediction，可以理解简单的分类任务，用以判断两个输入语句是否为前后关系。

使用预训练模型进行微调时，BERT 针对不同的预测任务给予了不同的方法。如下图 7.58所示，由此可以看出 BERT 不仅适合做简单的文本分类、序列标注等常见任务，对于问答系统、信息检索、聊天机器人等任务都有着重大的突破。在本实验中，我们完成的是问答系统的任务，对应与图 7.58中的（c）部分。

7.3.2.3　BatchMatMul 算子

BatchMatMulV2 算子是 BERT 模型中重要的一个操作，其主要进行两个张量相乘的操作，其中这两个输入张量的 shape 可能不一致，因此还涉及到 BroadCast 操作，其还有两个参数可对两个输入张量进行共轭转制操作。

BatchMatMulV2 的输入输出如下：

Arguments: x：2 维或更高维度，输入 shape 为 [...,r_x,c_x] y: 2 维或更高维度，输入 shape 为 [...,r_y,c_y]

Attrx: adj_x: 当为 True 时，对输入 x 进行共轭转制，默认为 False adj_y: 当为 True 时，

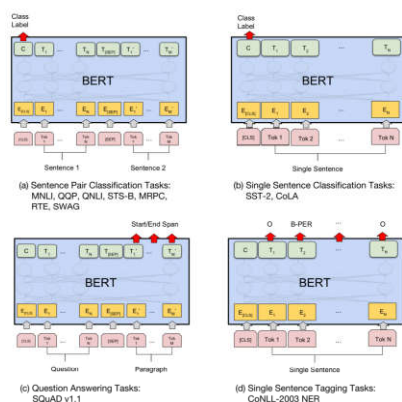

图 7.58 BERT Fine Tuning

对输入 y 进行共轭转制，默认为 False

Return：Output：返回 3 维或更高维度的张量，输出 shape 为 [...,r_o,c_o]

其和 BatchMatMul 算子的区别在于 BatchMatMulV2 支持 batch 层面的 broadcasting 操作。

7.3.3 实验环境

本实验所涉及的硬件平台和软件环境与其它综合实验相同，参照附录章节 **??**云平台的使用。

7.3.4 实验内容

1. BERT 重训练（可选）：通过 Fine-Tuning 的方式，进一步提升模型的精度。目前 DLP 还不支持训练，学生可以使用 CPU 或 GPU 完成模型的重训练工作，也可直接使用教材提供的模型做下一步实验。
2. BERT DLP 移植：将 BERT 网络移植在 DLP 中，包含模型量化，config 设置等操作。
3. BANGC BatchMatMulV2 算子实现与集成。
4. 分别比较 CPU 性能、MLU 性能 (BatchMatMulV2 放在 CPU 上计算)、MLU 性能 (BatchMatMulV2 使用 BANGC 实现)、BERT 大算子网络性能。

7.3.5 实验步骤

7.3.5.1 BERT 重训练

由于 BERT 只提供了预训练模型，在做特点任务时，需要先做微调训练。

首先在 GitHub 中，拉取工程：https://github.com/google-research/bert.git

根据提示，分别下载模型和训练测试数据集。在这里，我们选择 SQuAD 1.1 任务，因此需要分别下载 Bert-Base(Uncased, 12-layer, 768-hidden, 12-heads, 110M parameters) 模型以及 train-v1.1.json、dev-v1.1.json 数据集。

中科院计算所

执行如下所示的命令，在 CPU 或 GPU 环境中进行训练，并验证精度。

```
export BERT_BASE_DIR=/path/to/bert-model/uncased_L-12_H-768_A-12
export SQUAD_DIR=/path/to/dataset/squad
python run_squad.py \
  --vocab_file=$BERT_BASE_DIR/vocab.txt \
  --bert_config_file=$BERT_BASE_DIR/bert_config.json \
  --init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
  --do_train=True \
  --train_file=$SQUAD_DIR/train-v1.1.json \
  --do_predict=True \
  --predict_file=$SQUAD_DIR/dev-v1.1.json \
  --train_batch_size=12 \
  --learning_rate=3e-5 \
  --num_train_epochs=2.0 \
  --max_seq_length=384 \
  --doc_stride=128 \
  --output_dir=/path/to/squad_base/

python $SQUAD_DIR/evaluate-v1.1.py $SQUAD_DIR/dev-v1.1.json /path/to/squad_base/
    predictions.json
```

图 7.59　BERT Fine Tuning

执行完上述指令后，会打印相应的输出精度："f1": xx.xx, "exact_match": xx.xx。

由于微调训练实验耗时较长且对硬件环境有所要求，不作为本章的重点，学生只需了解训练的流程，在实际实验阶段可直接使用 DLP 提供的训练模型进行推理实验。

7.3.5.2　BERT DLP 移植

为了将 BERT 网络推理在 DLP 硬件上，需要对相关代码进行修改移植。与之前的综合实验类似，首先需要增加 DLP 专属的 config 配置，其次还需要对模型进行量化操作。为了方便推理，还在原有的 run_squad.py 文件中增加了 checkpoint 模型转 pb 模型的操作，以此简化后续调试的流程。该部分的代码修改增加部分如下图 7.60所示：

从上图中代码可知，在将 checkpoint 模型转 pb 模型时，由于 checkpoint 格式不包含模型的结构信息，所以需要首先在 session 实例中构造图的结构，然后加载相应的模型并作初始化。紧接着需要将计算图中的变量 (variable) 转为常量 (constant)，指定输出节点并做序列化为字符的操作。最终生成的 pb 模型相比原模型仅保留了与输出节点相关的节点，相当于在原模型的基础上做了剪枝的操作，并且保存的常量在推理时不需要初始化操作，进一步提高了推理时的性能。

在配置 DLP 相关的 config 时，增加了 optype_black_list 设置，经过该设置会将配置的节点自动执行在 CPU 而非 DLP 设备上。可以将一些 DLP 支持但由于性能精度暂不满足要求的算子暂时执行在 CPU 上。

在进行量化时，为了兼顾性能与精度的要求，本次实验选择 INT16 作为量化精度。首先进行配置文件 bert_int16.ini 的编写，然后执行 DLP 提供的量化脚本 fppb_to_intpb.py 得到量化后的模型。在该实验中，DLP 会将 MatMul、MLP、BatchMatMulV2 等操作量化为 INT16 精度，其它节点仍然以 FP32 或 FP16 的精度进行运算。该部分代码如图 7.61所示：

中科院计算所

7.3.5.3 BatchMatMulV2 BANGC 代码实现

完成 DLP 移植后，推理时会发现 DLP 暂时不支持 BatchMatMulV2 算子，由于 BERT 中存在大量的 BatchMatMulV2 操作，不仅会导致 CPU 利用率的升高，还会由于大量的分段造成性能的下降。为了解决该问题，我们设计了 BatchMatMulV2 算子的 BCL 实现，在减小分段的同时通过 INT16 定点计算减小时延提高效率。

1. 设计目标使用 BCL 实现 BatchMatMulV2 算子，并以 cnplugin 封装，供框架直接调用。此模块的参数规格设计为：

   输入矩阵 1：input_0[dim_0, dim_1, m, k]，fp16；输入矩阵 2：input_1[dim_0, dim_1, n, k], fp16；输出矩阵：output[dim_0, dim_1, m, n], fp16

2. 设计思路考虑到使用循环向量乘指令不能很好地体现 MLU 的矩阵运算性能，因此设计使用卷积指令来执行 MatMul，右矩阵将会在量化完成后拷贝至 wram 空间，使用 __bang_conv 指令进行计算。因为最终是要调用卷积指令来进行矩阵乘法，所以代码中 n，ci，co 是沿用的卷积指令符号系统，这与接口中 m，n，k 的对应关系是：$m \rightarrow n_$；$n \rightarrow co_$；$k \rightarrow ci_$。其中 $n_ \times ci_$ 为卷积的输入，$ci_ \times co_$ 为卷积的 filter，$n_ \times co_$ 为卷积的输出。

3. 函数设计

   - **__mlu_entry__ void BatchMatMulV2Kernel_MLU270_half ()：**
     入口函数，主要用于开辟 NRAM，WRAM 和 SRAM 的空间，遍历 dim_0 和 dim_1 循环进行卷积，并调用 int_matmul_wrap 函数进行计算，其主要参数如下:
     void* left_ddr：GDRAM 上的左矩阵
     void* right_ddr：GDRAM 上的右矩阵
     void* dst_ddr：GDRAM 上的输出矩阵
     int dim_0：输入维度，左右矩阵的第一维
     int dim_1：输入维度，左右矩阵的第二维
     int m：输入维度，左矩阵的第三维
     int n：输入维度，右矩阵的第四维
     int k：输入维度，左矩阵的第四维，右矩阵的第三维
     float scale_0：左矩阵的量化 scale 参数
     int pos_0：左矩阵的量化 pos 参数
     float scale_1：右矩阵的量化 scale 参数
     int pos_1：右矩阵的量化 pos 参数
   - **template <typename IT, typename FT>**
     __mlu_func__ void int_matmul_wrap()：

主要计算流程函数，将左右矩阵分别量化并拷贝至对应的空间，调用 compute 函数进行计算，然后将结果拷贝回 GDRAM 上，其主要参数如下（省略与上文重复的参数，下同）：

IT *nram_buf：NRAM 上开辟的用于计算的空间

IT *wram_buf：WRAM 上开辟的用于计算的空间

IT *sram_buf：SRAM 上开辟的用于计算的空间

- **template <typename T> __mlu_func__ void load_left()：**

  将左矩阵拷贝至 NRAM 上。并进行量化操作，然后拷贝至 SRAM 上，将 NRAM 的空间空余出来用于量化和摆放右矩阵。

- **_mlu_func__ void SegStrategyByFactor()：**

  m，n 拆分策略函数。按照预设的 factor 搜索 n 值，并按照此 n 值拆分 m 值。

  int factor：预设的 n 的拆分倍数

  int k_up：k 维度进行 64 位对齐的结果

  int byteit：输入数据类型的字节数

  int byteft：输出数据类型的字节数

  int &work_m：（输出）拆分后每次读取的 m 值

  int &work_n：（输出）拆分后每次读取的 n 值

- **__mlu_func__ void SegStrategyByMaxCo():**

  最大空间的 m，n 拆分策略函数，按照最大 WRAM 空间寻找 n 值，并按照此 n 值拆分 m 值。

- **template <typename T> __mlu_func__ void load_right():**

  将右矩阵拷贝至 NRAM 上。并进行量化操作，然后拷贝至 WRAM 上。

  T *right_inchip：WRAM 上的最大空间

  T *right_ddr：GDRAM 上的右矩阵

  T *nram_buf：临时 NRAM 空间，用于量化右矩阵

  int n_in_nram：根据 k_up 计算出的 NRAM 上的剩余空间一次所能计算的最大 n 值

- **template <typename IT, typename FT> __mlu_func__ void compute()：**

  计算卷积，得到 MatMul 的结果

  FT *dst：存放结果的 NRAM 空间

  IT *left：存放在 NRAM 上的左矩阵

  IT *right：存放在 WRAM 上的右矩阵

  FT recip_scale：量化 scale 参数

  int pos：量化 pos 参数

4. 具体实现

中科院计算所

- 准备所需的变量，进行空间划分。
- 最外层对 n 方向和 c 方向使用两重循环，保证多维度输入结果的准确性。
- 按照最大 SRAM 空间进行 m 维度拆分: work_m = SRAM_BUF_SIZE / sizeof(TYPE) / k

  记为 work_m，循环调用 int_matmul_wrap 计算维度为 [work_m, n，k] 的 MatMul。如下图所示：图中蓝色阴影部分即为 *SRAM_BUF_SIZE*
- 在每个 int_matmul_wrap 运算中，work_m * k < SRAM_BUF_SIZE，所以能保证完全载入到 SRAM 空间上，故先执行 load_left，将 input_0 数据 load 到 SRAM 空间上。
- 调用 SegStrategyByFactor 和 SegStrategyByMaxCo 函数寻找 n 和 wrok_m 的拆分策略，其中一种拆分策略为：

  int work_n_1 = WRAM_BUF_SIZE / byteit / k_up;

  int work_n_2 = (NRAM_BUF_SIZE - byteit * k_up) / byteft;

  work_n = ALIGN_DN(MIN(work_n_1, work_n_2), 64);

  int work_m_1 = NRAM_BUF_SIZE / (byteit * k_up + byteft * work_n);

  int work_m_2 = (NRAM_BUF_SIZE - byteit * 64 * k_up) / (byteit * k_up);

  work_m’ = MIN(work_m_1, work_m_2);

  将上述的计算结果记为 work_n，work_m’。

  通过循环 for (int cur_n = 0; cur_n < n; cur_n += taskDim * work_n) 计算维度为 [work_m, work_n, k] 的 MatMul，流程图如 7.64所示：

  图中黄色阴影部分即为 work_n * k_up * byteit，根据选取的拆分策略不同可能为 WRAM_BUF_SIZE 也可能为较小值。
- 在每个 [work_m, work_n, k] 的 MatMul 计算中，因为拆分策略中已经保证 work_n * k_up * byteit < WRAM_BUF_SIZE，所以能保证完全载入到 WRAM 空间上，故执行 load_right，将 input_1 数据 load 到 WRAM 空间上。
- 在每个 [work_m, work_n, k] 的的 MatMul 计算中，还需要按照 work_m’ 进行拆分，以保证左矩阵和输出矩阵有足够的空间放在 NRAM 上，如下图 7.65所示：

  上图中将 output 展示出来是由于 output 的结果与 input_0 共用 NRAM 空间，拆分策略中已经保证 (work_n * byteit + k_up * byteft) * work_m’ < NRAM_BUF_SIZE，所以输入输出均能完整的载入到 NRAM 空间上。
- 进行 compute 计算，计算每个 [work_m’, work_n, k] 的 MatMul。
- 多重循环中按照输出结果应有的地址偏移将 output 拷出到 GDRAM 上。

7.3.5.4 BatchMatMulV2 BANGC 代码集成

类似于前两个实验的算子添加过程，集成主要分为 CNPlugin 的集成和 TensorFlow 框架的集成。

中科院计算所

1. CNPlugin 集成：分别完成 cnmlCreatePluginBatchMatMulV2OpParam();cnmlDestroyPluginBatchMatM
   以及 cnmlStatus_t cnmlComputePluginBatchMatMulV2OpForward() 构造函数。传入 BCL 计算程序所需的参数：$scale_0$,pos_0,$scale_1$,pos_1(代表算子两个输入的量化参数)；dim_0,dim_1(代表两个输入的第 0 个和第 1 个维度的值)；m,n,k(代表第一个输入的最后两个维度为 [m,k]，第一个输入的最后两个维度为 [n,k])。

   此外还需在 cnplugin.h 头文件中增加相应的接口，如下图 7.67所示：

2. TensorFlow 集成：
   - 算子注册:
   - 定义 MLULib 层接口，用以连接 CNPlugin 的封装接口：
   - 定义 MLUOp 层接口，实现算子类的 Create 和 Compute 方法。
   - 定义 MLUStream 层接口
   - 定义 MLUOpKernel 层接口: 在 tensorflow/core/kernels/batch_matmul_v2_op_mlu.h 文件中完成以下函数，并调用 MLUStream 层的 BatchMatMulV2 函数。

分别完成 CNPlugin 和 TF 的集成后，即可运行正常的推理代码，完成 SQuAD 1.1 验证数据集的测试。完整的推理执行命令如图 7.72

7.3.5.5 BERT 大算子

为了进一步比较 BCL 的优越性，DLP 还实现了完整 BERT BCL 大算子的推理。其在 pb 模型中仅使用一个 CNPlugin 中的集成算子代替其它所有小算子，因此避免了多余的分段。同时由于开发者可以通过 BCL 自由控制硬件计算资源和内存的使用，进一步提高了片上内存的使用效率，极大的提升了整体端到端的性能。

使用者在运行 BERT 大算子模型时，仍然使用 DLP 移植后的开源代码，仅需将模型来源指定为相应的 pb 模型。

在生成大算子模型时，与第 7.3.5.2章节不同的是，其在量化配置文件中选择大算子作为输出节点。配置文件如下图所示：

运行推理代码后，会发现此种方式下性能有着大幅度的提升。

7.3.6 实验评估

相比其它两个综合实验，BERT 实验困难度最高，因此难度系数分为 1.2。学生最终得分为 1.2 乘以相应的得分。

- 60 分标准：完成 BatchMatMulV2 BCL 算子的实现与 cnplugin 的集成，可以跑通单算子测试程序。
- 70 分标准：在 60 分的基础上，执行单算子测试脚本时，错误率在 1% 以内。

中科院计算所

- 80 分标准：在 70 分的基础上，完成 TensorFlow 算子集成，执行完整的 BERT 模型推理验证脚本，f1 值大于 80、exact_math 值大于 70，单 Batch 单次推理平均延时小于 100ms。
- 90 分标准：在 80 分的基础上，执行测试脚本时，f1 值大于 82、exact_math 值大于 73，单 Batch 单次推理平均延时小于 60ms。
- 100 分标准：在 90 分的基础上，执行测试脚本时，f1 值大于 82.5、exact_match 值大于 74，单 Batch 单次推理平均延时小于 50ms。

7.3.7 实验思考

1. 本章提供的样例代码中通过使用卷积来做矩阵运算，可否使用矩阵乘等基本 BCL 操作来完成？2. 使用上述不同方式完成 BCL 算子实现有何不同？哪种效率更高？3.BatchMatMulV2 BCL 算子的两个输入最后两个维度为何为 [m,k]、[n,k] 形式？而不是 [m,k]、[k,n] 形式，这对 TensorFlow 框架集成 BCL 算子时调用 CNML 有何影响。4. 与大算子 BERT 相比，有何方法可以进一步提升小算子 BERT 网络的性能？

中科院计算所

```python
from tensorflow.core.framework import graph_pb2
from tensorflow.python.framework import importer
from tensorflow.python.framework import ops
...
flags.DEFINE_bool("export_frozen_graph", False, "Whether to export SavedModel.")
...
def save_dlp_frozen_graph(bert_config, seq_len, init_checkpoint):
  tf_config = tf.ConfigProto()
  with tf.Session(config=tf_config) as tf_sess:
    input_ids = tf.placeholder(tf.int32, [None, seq_len], 'input_ids')
    input_mask = tf.placeholder(tf.int32, [None, seq_len], 'input_mask')
    segment_ids = tf.placeholder(tf.int32, [None, seq_len], 'segment_ids')

    # construct origin graph
    (start_logits, end_logits) = create_model(
        bert_config=bert_config,
        is_training=False,
        input_ids=input_ids,
        input_mask=input_mask,
        segment_ids=segment_ids,
        use_one_hot_embeddings=False)

    output_node_names = ['unstack']
    tvars = tf.trainable_variables()
    initialized_variable_names = {}
    if init_checkpoint:
      (assignment_map, initialized_variable_names
      ) = modeling.get_assignment_map_from_checkpoint(tvars, init_checkpoint)

      tf.train.init_from_checkpoint(init_checkpoint, assignment_map)
      tf.logging.info("**** Load Variables Done ****")
      tf_sess.run(tf.global_variables_initializer())
    else:
      raise ValueError("No init_checkpoint!")

    tf.logging.info("**** Trainable Variables ****")
    for var in tvars:
      init_string = ""
      if var.name in initialized_variable_names:
        init_string = ", *INIT_FROM_CKPT*"
      tf.logging.info(" name = %s, shape = %s%s", var.name, var.shape, init_string)

    frozen_graph = tf.graph_util.convert_variables_to_constants(tf_sess,
        tf_sess.graph.as_graph_def(), output_node_names)

    export_file = os.path.join(FLAGS.output_dir, "frozen_model.pb")
    with tf.gfile.GFile(export_file, "wb") as f:
      f.write(frozen_graph.SerializeToString())
    tf.logging.info("**** Save Frozen Graph Done ****")
...
def main(_):
  ...
  config = tf.ConfigProto(allow_soft_placement=True,
                          inter_op_parallelism_threads=1,
                          intra_op_parallelism_threads=1,
                          log_device_placement=False)
  config.mlu_options.core_version = "MLU270"
  config.mlu_options.precision = "int16"
  config.mlu_options.save_offline_model = False
  config.mlu_options.core_num = 16
  config.mlu_options.optype_black_list = "OneHot"
  run_config = tf.contrib.tpu.RunConfig(...
      session_config=config)
```

图 7.60 BERT DLP 移植

```
[config]
quantization_type = int16
device_mode = clean
int_op_list = FC

[model]
activation_quantization_alg = naive
input_tensor_names = input_ids:0, input_mask:0, segment_ids:0
weight_quantization_alg = naive
seq_length = 128
original_models_path = /path/to/frozen/pb
output_tensor_names = unstack:0,unstack:1
save_model_path = /path/to/output/frozen_model_int16.pb

[data]
do_lower_case = False
doc_stride = 128
max_query_length = 64
batch_size = 8
num_runs = 1
vocab_file = bert/uncased_L-12_H-768_A-12/vocab.txt
data_list = bert/squad/dev-v1.1.json
```

图 7.61　BERT 量化

```
for (int cur_dim0 = 0; cur_dim0 < dim_0; cur_dim0++) {
    for (int cur_dim1 = 0; cur_dim1 < dim_1; cur_dim1++) {
    ……
    }
}
```

图 7.62　两重循环

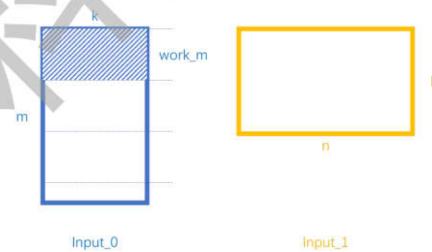

图 7.63　INPUT0 拆分

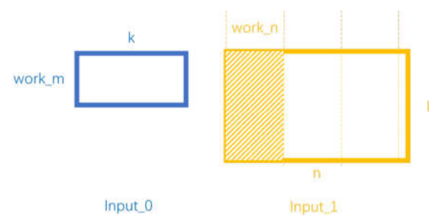

图 7.64　INPUT1 拆分

中科院计算所

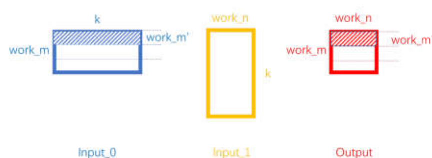

图 7.65 NRAM 拆分

```
struct cnmlPluginBatchMatMulV2OpParam {...};

typedef cnmlPluginBatchMatMulV2OpParam *cnmlPluginBatchMatMulV2OpParam_t;

cnmlStatus_t cnmlCreatePluginBatchMatMulV2OpParam(...);

cnmlStatus_t cnmlDestroyPluginBatchMatMulV2OpParam(...);

cnmlStatus_t cnmlCreatePluginBatchMatMulV2Op(...);

cnmlStatus_t cnmlComputePluginBatchMatMulV2OpForward(...);
```

图 7.66 BERT-CNPlugin 接口

```
在tensorflow/core/kernels/batch_matmul_op_impl.h中增加注册
#if CAMBRICON_MLU
#define REGISTER_BATCH_MATMUL_MLU(T)                    \
  REGISTER_KERNEL_BUILDER(                              \
      Name("MLUBatchMatMulV2").Device(DEVICE_MLU).TypeConstraint<T>("T"),        \
      MLUBatchMatMulV2<T>)
#endif // CAMBRICON_MLU

在tensorflow/core/ops/math_ops.cc中增加注册，并声明该算子只可用于DLP中。
REGISTER_OP("MLUBatchMatMulV2")
    .Input("x: T")
    .Input("y: T")
    .Output("output: T")
    .Attr("T: {half, float}")
    .Attr("input1_position: int = 0")
    .Attr("input1_scale: float = 1.0")
    .Attr("input2_position: int = 0")
    .Attr("input2_scale: float = 1.0")
    .Attr("adj_x: bool = false")
    .Attr("adj_y: bool = false")
    .SetShapeFn(shape_inference::BatchMatMulV2Shape)
    .Doc(R"doc(
MLUBatchMatMulV2 (DLP only)
)doc");
```

图 7.67 算子注册

```
###tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc
tensorflow::Status CreateBatchMatMulV2Op(MLUBaseOp** op,
            cnmlPluginBatchMatMulV2OpParam_t param,
            MLUTensor* input1, MLUTensor* input2,
            MLUTensor* output) {
  MLUTensor* inputs[2];
  MLUTensor* outputs[1];
  inputs[0] = input1;
  inputs[1] = input2;
  outputs[0] = output;
  CNML_RETURN_STATUS(cnmlCreatePluginBatchMatMulV2Op(op, param, inputs, outputs));
}

tensorflow::Status ComputeBatchMatMulV2Op(MLUBaseOp* op,
            MLUCnrtQueue* queue,
            void** inputs, int input_num,
            void** outputs, int output_num) {
  CNML_RETURN_STATUS(cnmlComputePluginBatchMatMulV2OpForward(
      op, inputs, input_num, outputs, output_num, queue));
}

###tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h
tensorflow::Status CreateBatchMatMulV2Op(MLUBaseOp** op,
            cnmlPluginBatchMatMulV2OpParam_t param,
            MLUTensor* input1, MLUTensor* input2,
            MLUTensor* output);
tensorflow::Status ComputeBatchMatMulV2Op(MLUBaseOp* op,
            MLUCnrtQueue* queue,
            void** inputs, int inputs_num,
            void** outputs, int outputs_num);

###tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h
struct MLUBatchMatMulV2OpParam {
  float scale_0_;
  int pos_0_;
  float scale_1_;
  int pos_1_;
  int dim_0_;
  int dim_1_;
  int m_;
  int n_;
  int k_;
  MLUBatchMatMulV2OpParam(float scale_0, int pos_0,
                          float scale_1, int pos_1,
                          int dim_0, int dim_1,
                          int m, int n, int k)
    : scale_0_(scale_0), pos_0_(pos_0), scale_1_(scale_1), pos_1_(pos_1),
      dim_0_(dim_0), dim_1_(dim_1), m_(m), n_(n), k_(k) {}
};
...
DECLARE_OP_CLASS(MLUBatchMatMulV2);
```

图 7.68 MLULib 接口

中科院计算所

```
###tensorflow/stream_executor/mlu/mlu_api/ops/batch_matmul_v2.cc
#include "tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h"
#include "tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h"
#include "tensorflow/stream_executor/mlu/mlu_api/tf_mlu_intf.h"

namespace stream_executor {
namespace mlu {
namespace ops {

Status MLUBatchMatMulV2::CreateMLUOp(std::vector<MLUTensor *> &inputs,
            std::vector<MLUTensor *> &outputs,
            void *param) {
  TF_PARAMS_CHECK(inputs.size() > 1, "Missing input");
  TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");

  MLUTensor *in0 = inputs.at(0);
  MLUTensor *in1 = inputs.at(1);
  MLUTensor *output = outputs.at(0);

  MLULOG(3) << "CreateBatchMatMulV2Op, input1: "
            << lib::MLUTensorUtil(in0).DebugString()
            << ", input2: " << lib::MLUTensorUtil(in1).DebugString()
            << ", output: " << lib::MLUTensorUtil(output).DebugString();

  MLUBatchMatMulV2OpParam *op_param = static_cast<MLUBatchMatMulV2OpParam *>(param);
  ...

  TF_STATUS_CHECK(lib::CreateBatchMatMulV2Op(...));

  base_ops_.push_back(batch_matmul_op_ptr);

  return Status::OK();
}

Status MLUBatchMatMulV2::Compute(const std::vector<void *> &inputs,
                                 const std::vector<void *> &outputs,
                                 cnrtQueue_t queue) {
  MLULOG(3) << "ComputeMLUBatchMatMulV2";
  ...
  TF_STATUS_CHECK(lib::ComputeBatchMatMulV2Op(...));

  TF_CNRT_CHECK(cnrtSyncQueue(queue));

  return Status::OK();
}

}  // namespace ops
}  // namespace mlu
}  // namespace stream_executor
```

图 7.69 MLUOp 接口

中科院计算所

```
Status BatchMatMulV2(OpKernelContext* ctx,
  Tensor* input1, Tensor* input2,
  float scale_0, int pos_0, float scale_1,
  int pos_1, int dim_0, int dim_1, int m,
  int n, int k, Tensor* output) {
  ops::MLUBatchMatMulV2OpParam op_param(scale_0, pos_0, scale_1,
      pos_1, dim_0, dim_1, m, n, k);
  return CommonOpImpl<ops::MLUBatchMatMulV2>(ctx, {input1, input2},
      {output}, static_cast<void*>(&op_param));
}
```

图 7.70 MLUStream 接口

```
#ifndef TENSORFLOW_CORE_KERNELS_BATCH_MATMUL_V2_OP_MLU_H_
#define TENSORFLOW_CORE_KERNELS_BATCH_MATMUL_V2_OP_MLU_H_
#if CAMBRICON_MLU
...
#include "tensorflow/stream_executor/mlu/mlu_stream.h"

namespace tensorflow {

template <typename T>
class MLUBatchMatMulV2 : public MLUOpKernel {
 public:
  explicit MLUBatchMatMulV2(OpKernelConstruction* context)
    : MLUOpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("adj_x", &adj_x_));
    OP_REQUIRES_OK(context, context->GetAttr("adj_y", &adj_y_));
    ...
    }

  void ComputeOnMLU(OpKernelContext* ctx) override {
    se::mlu::MLUStream* stream = static_cast<se::mlu::MLUStream*>(
      ctx->op_device_context()->stream()->implementation());
    ...

    OP_REQUIRES_OK(ctx, stream->BatchMatMulV2(ctx, ...));
  };

 private:
  bool adj_x_;
  bool adj_y_;
  ...
};
}  // namespace tensorflow
#endif  // CAMBRICON_MLU
#endif  // TENSORFLOW_CORE_KERNELS_BATCH_MATMUL_V2_OP_MLU_H_
```

图 7.71 MLUOpKernel 层接口

中科院计算所

```
python run_squad.py \
  --vocab_file=${BERT_BASE_DIR}/vocab.txt \
  --bert_config_file=${BERT_BASE_DIR}/bert_config.json \
  --predict_batch_size=8 \
  --max_seq_length=128 \
  --hidden_size=768 \
  --output_dir=${OUTPUT_DIR} \
  --export_frozen_graph=false \
  --do_predict=true \
  --predict_file=${SQUAD_DIR}/dev-v1.1.json
python ${SQUAD_DIR}/evaluate-v1.1.py \
    ${SQUAD_DIR}/dev-v1.1.json \
    ${OUTPUT_DIR}/predictions.json
```

图 7.72　BERT 推理

```
[config]
quantization_type = int16
device_mode = clean
int_op_list = FC, BatchMatMul

[model]
input_nodes = input_ids, input_mask, segment_ids
layer_num = 12
seq_length = 128
original_models_path = /path/to/frozen_model.pb
output_nodes = unstack
quantization_output_nodes = bert_plugin_op
scope_names = attention/self/query/MatMul, attention/self/key/MatMul,
              attention/self/value/MatMul, attention/self/MatMul,
              attention/self/MatMul_1, attention/output/dense/MatMul,
              intermediate/dense/MatMul, output/dense/MatMul
post_process_name = MatMul
save_model_path = /path/to/output/frozen_model_int16.pb

[data]
do_lower_case = False
doc_stride = 128
max_query_length = 64
batch_size = 8
iters = 1
vocab_file = bert/uncased_L-12_H-768_A-12/vocab.txt
data_list = bert/squad/dev-v1.1.json
```

图 7.73　BERT 大算子配置文件

中科院计算所