为 OneFlow 添加新的前端语言

结项报告

项目 ID: 210130141
项目导师: 蔡晟航
申请人: 周泽楷
导师邮箱: caishenghang@oneflow.org
申请人邮箱: 946079208@qq.com

2021 年 9 月 30 日

目 录

1 项目信息

1.1 项目名称

为 OneFlow 添加新的前端语言

1.2 方案描述

1.2.1 方案概述

利用 JNI 技术为 OneFlow 实现 Java 前端语言，提供模型部署相关的功能。

1.2.2 目标

为 OneFlow 添加一门新的前端语言，支持模型部署的功能。项目输出一个动态链接库 liboneflow_java.so，用来给 Java 前端链接调用；一个 jar 包 oneflow-api.jar，给 Java 提供接口；一个调用例子和文档，帮助用户上手使用。

1.2.3 详细实施方案

按照功能，将代码划分为四个部分。

- 第一部分，调用 OneFlow core，对 core 进行适当封装。提供接口，方便调用，提高代码复用性。
- 第二部分，负责做交互，一方面从 Java 获取数据，向 Java 发送数据，另一方面调用第一部分的接口，推送数据和拉取数据。
- 第三部分，Java 接口，负责做一些相对较高层的处理，调用第二部分暴露的 native 方法，将这些细粒度的方法组合成一个个相对完整的功能，进一步封装。
- 第四部分，CMake 代码，提供项目构建的方法，解决软件包依赖，项目间依赖等问题。最终目标是，用户可以几行命令将整个软件构建起来。

1.3 时间规划

主办方给出了三个月的开发时间，从 7 月 1 日开始，9 月 30 日结束。之前将开发任务分解为 11 周进行，每周一个小目标，如表 1 所示。剩下两周，根据实际执行情况灵活调配。

表 1: 计划表

开始时间	结束时间	目标
7.1	7.7	Maven 项目管理和接口设计
7.8	7.14	env, session 初始化
7.15	7.21	实现计算图的加载，加载图结构
7.22	7.28	编译计算图
7.29	8.4	实现 C++ JobInstance
8.5	8.11	加载图权重，启动 session
8.12	8.18	Java 实现一个 NDArray
8.19	8.25	实现模型输入
8.26	9.1	实现模型推理，获取输出，实现关闭 session
9.2	9.8	完善测试用例和重构
9.9	9.15	完善 Java 代码和文档，编写一个调用例子。
9.16	9.22	
9.23	9.30	

2 项目总结

2.1 项目产出

2.1.1 Java 前端

Java 前端支持 OneFlow 推理的功能，可以加载模型，进行前向传播。

Java 前端的代码分为两个部分，一个是 native 代码，一个是 Java 代码。native 代码包括 CMake 和 C++ 代码，编译生成一个动态链接库。Java 代码打包成 oneflow-api.jar，用户可以引入并使用，可以导入到本地 Maven 仓库。

2.1.2 OneFlow Java API 教程

地址:https://www.yuque.com/docs/share/9e5691a9-a46f-4aae-ab6d-b63c2e21a38c

这篇教程由三部分组成:

第一部分：模型。在这个部分中，我们将使用 OneFlow 在 mnist 数据集上训练一个手写体识别的模型，并且将这个模型保存下来；你也可以跳过这个步骤，直接下载模型。

第二部分：环境准备。准备 Java 的环境，下载 Jar 包，将 Jar 包安装到本地 Maven 仓库，使用 IDEA 新建 Maven 项目。

第三部分：介绍 OneFlow 的 Java API，并且使用这个 API 搭建一个识别手写体图片的命令行工具。

2.1.3 使用例程

例程代码在仓库中的 oneflow-example 下面。

2.1.4 项目经验分享

微信文章：https://mp.weixin.qq.com/s/-2Am97QOC4BpNzJLVcbhdg
博客园：https://www.cnblogs.com/zzk0/p/15191920.html
这篇项目经验分享的文章，简单介绍了这个项目的任务、目标、意义。梳理了 OneFlow 各个流程，详细记录了实施的过程，记录探索、设计、重构的各个阶段。

2.1.5 源码分析文章

为了对 OneFlow 源码有更加深刻的理解，了解内部的设计和构造，读了读 OneFlow 的设计，写了几篇源码分析笔记。

- 初始化环境：https://www.cnblogs.com/zzk0/p/15212161.html
- Python 端构图：https://www.cnblogs.com/zzk0/p/15213127.html
- 从 Op 到 Job：https://www.cnblogs.com/zzk0/p/15216185.html
- 启动 Session：https://www.cnblogs.com/zzk0/p/15217438.html
- 从 Job 到 Plan：https://www.cnblogs.com/zzk0/p/15222259.html
- 启动 Runtime：https://www.cnblogs.com/zzk0/p/15226851.html
- 计算数据的来源：https://www.cnblogs.com/zzk0/p/15230583.html
- softmax 的分析：https://www.cnblogs.com/zzk0/p/15173022.html
- Python 端构图的另一篇分析：https://www.cnblogs.com/zzk0/p/15009227.html

2.2 方案进度

根据原定方案，在规定的时间内顺利完成了任务要求的输出。

2.3 遇到的问题及解决方案

2.3.1 数组多次复制的问题

问题描述: Java 将一个数组对象传给 C++，使用 jni.h 中提供的方法从数组对象拿出数组，会发生**一次复制**。OneFlow 启动 Job 是异步的，一旦函数结束，数组就会被销毁掉，此时还需要**再复制一次**。之后还要将数据推送到 Blob 里面，又**复制了一次**。数组复制了多次，如何解决这个问题呢?

- 解决方法: 使用 DirectBuffer 即可！在 Java 中，有堆内存和堆外内存之分，堆内存由 JVM 进行管理，会受到 GC 的影响，因此 JNI 调用的时候，为了确保数组不受 GC 的影响，需要进行数据复制。堆外内存，是在 JVM 同个进程中，但是不受 GC 影响，内容是稳定的，在 JNI 调用的时候，不会发生复制，并且传递过去的是一个指针。用了 DirectBuffer 之后，只需要一次数组复制就好了，将数据推送到 Blob 里面。

2.3.2 Jar 包构建

问题描述: 前期为了方便开发和调试，直接新建了一个 Maven 项目，后来发现，除了我之外别人都不太好构建。此外，前期还将 OneFlow 内部的.proto 文件编译成了.java 文件，放入到了 Maven 项目中，如果 OneFlow 内部的.proto 文件更新了，需要手动更新。这些都不太方便。

- 解决方法（*[1]）: 使用 CMake 直接构建 Jar 包，使用 CMake 来编译 proto 文件，并且直接打包提供给 Java 前端用。分为三步走，第一步构建出 proto-java.jar。第二步编译.proto 为.java，并且打包成 oneflow-proto.jar，这一步依赖前一步的 jar 包。第三步，编译 Java 前端的代码，这一步依赖前两步的 Jar 包。

2.3.3 protobuf 的 bug

问题描述: 在项目申报的时候，发现了 protobuf 的一个 bug。将 proto 文件编译成 java 文件时，proto 中定义的一个 message[2]，它的一个属性命名为 input，生成 java 文件后会和 java 的局部变量产生冲突。

- 解决方法: 前期没有找到问题在哪里，所以前期的解决办法时手动修改 message 定义的属性名字，这增大了在其他人那里构建的难度。后

[1] 重构的项目里面不再需要编译 OneFlow 内部的.proto 文件了，重构之后只需要第三步即可。
[2] oneflow/core/framework/user_op_conf.proto

来用新版本试了一下，已经解决了。在本机上用 apt 安装的 3.0.0 版本存在这个问题，OneFlow 目前依赖的 3.9.0 没有这个问题。所以更新版本就好了。

2.4 项目完成质量

完成基本的功能，提供的 Java 接口导师评价“挺清爽的”。

C++ 代码能够做到规范化，参考 OneFlow 其他代码进行规范化。Java 部分的代码命名规范，不过逻辑写的不够清楚，缺少适当的设计和抽象。

整体代码质量有待提高，需要学习最佳实践，然后将最佳实践应用到这些代码上面。

2.5 与导师沟通及反馈的情况

沟通很顺利，反馈超及时。导师很负责任，组织开会、题目讲解、进度检查，感谢

3 代码示例

图 1 是 Java API 的示例代码。下面简单讲讲如何使用。

Java 的 API 主要由两个方面组成，一个是用来表示存储数据的数据结构 Tensor，一个是进行模型加载、前向传播的 Session。为了使用 Java API，需要执行如下步骤：

- 输入数据，存储在一个 float 数组当中。
- 将输入数据存放到 Tensor 当中。
- 创建输入 op 名字和输入数据向量之间的映射。
- 设置选项，比如设备、模型路径、模型版本。
- Session 初始化，打开 Session
- 执行前向传播
- 获取输出的向量
- 输出的后续处理由用户自行编写，比如获取最大值对应的位置作为推理的结果等操作。

```java
public void example() {
    String jobName = "mlp_inference";

    Option option = new Option();
    option.setDeviceTag("gpu")
            .setControlPort(11245)
            .setSavedModelDir("mnist_test/models")
            .setModelVersion("1");

    InferenceSession inferenceSession = new InferenceSession(option);
    inferenceSession.open();
    float[] image = readImage( filePath: "mnist_test/test_set/00000000_7.png");
    Tensor imageTensor = Tensor.fromBlob(image, new long[]{ 1, 1, 28, 28 });
    Tensor tagTensor = Tensor.fromBlob(new int[]{ 1 }, new long[]{ 1 });
    Map<String, Tensor> tensorMap = new HashMap<>();
    tensorMap.put("Input_14", imageTensor);
    tensorMap.put("Input_15", tagTensor);

    Map<String, Tensor> resultMap = inferenceSession.run(jobName, tensorMap);
    Tensor resTensor = resultMap.get("output");
    float[] resFloatArray = resTensor.getDataAsFloatArray();
    inferenceSession.close();
}
```

图 1: 项目结构

4 Java 和 Python 的性能对比

在这一节中，我们将简单对比 Python InferenceSession 和 Java InferenceSession 的性能。

4.1 实验设置

在 mnist 上训练一个 mlp 识别手写体的模型，一个 lenet 模型，将他们的计算图分别保存下来，分别使用 Python InferenceSession 和 Java InferenceSession 来进行推理。实验将分别在 CPU 和 GPU 设备上进行，设备后面的次数表示，在启动 Session、加载模型之后，进行 N 次“推送数据，前向传播，拉取数据”的时间。

表 2: Java 上的结果 (执行时间单位：ms)

	Java mlp		Java lenet	
	cpu	gpu	cpu	gpu
100	77	101	817	139
1000	631	1105	8249	1384
10000	7045	10661	81520	12534
100000	67903	96775	799904	135966

表 3: Python 上的结果 (执行时间单位：ms)

	Python mlp		Python lenet	
	cpu	gpu	cpu	gpu
100	211	203	1038	215
1000	1087	1750	9590	2094
10000	8969	16886	89650	19760
100000	112985	166767	863381	204650

4.2 实验结果

4.3 实验分析

对比 Java 和 Python，我们可以得出结论，Java 的执行效率会更高一点。

- 在 CPU 上：对于 mlp 模型来说，Java 单次推理平均不到 1 ms，只需要 0.67 ms，Python 中单次推理则需要 1.12 ms。对于 lenet 模型，Java 单次推理平均需要 8 ms，而 Python 需要 8.6 ms。因此，在 CPU 上，简单模型的加速会比较明显，复杂模型加速不太明显。
- 在 GPU 上：对于 mlp 模型，Java 单次推理平均 0.96 ms，Python 单次推理平均 1.66 ms。对于 lenet 模型，Java 单次推理需要 1.35 ms，而 Python 需要 2.04 ms。因此，在 GPU 上，不管模型复杂还是简单，加速都比较明显。
- 接下来分析同一个模型，不同的设备。对于 mlp 这样的简单来说，使用 cpu 并不见得就会比 gpu 要慢；对于 lenet 这样稍微复杂一点的模型来说，使用 gpu 的速度提升明显。

4.4 性能差异的原因

以下纯属个人猜想，没有实验证明。

性能差异的主要原因，应该是来自于“内存拷贝”次数的差异。为什么这么认为呢?

我们先来思考另一个问题，为什么简单模型 mlp 在 gpu 上慢了，而复杂模型 lenet 在 gpu 上加速又那么明显呢? 造成这种现象的本质原因是，简单模型计算开销很小，时间都浪费在了复制内存上了。特别是从 Host 复制到 Device 这种非常耗时的操作，直接导致了简单模型 mlp 在 gpu 上计算会慢。如果性能的瓶颈在计算，加速计算就可以得到很大的提升。

回到前面的问题，Python 和 Java 的性能差异来自于哪里? 我们先看简单模型在 cpu 上执行的结果，显然 Java 快很多，而复杂模型在 cpu 上的执行的结果，两者差异不大。因为简单模型计算不是瓶颈，内存拷贝造成的性能差异会明显；反之，复杂模型计算瓶颈是计算，所以内存拷贝造成的性能差异将会被掩盖。因而，我们可以下结论“内存拷贝”是造成性能差异的主要原因。

5 致谢

感谢开源之夏主办方，举办这样的活动实属不易，极大促进了学生和开源社区之间的互动。

感谢 OneFlow，从 0.4 开始关注 OneFlow，如今的 0.5 在易用性上提升很大。祝愿 OneFlow 1.0 可以获得高性能、易用性、完备性各方面的成功。

感谢我的导师蔡晟航，给我组织开会，讲解题目，提供指导。

感谢我的朋友郑东阳，一同度过了开源之夏，一起吃饭，一起游泳。在我低沉的时候给我鼓励，帮我做分析。

感谢!