智能编程语言算子开发与集成实验

1953729 吴浩泽

实验简介与说明

该实验是寒武纪陈云霁老师所著的《智能计算系统》自带的实验部分

该实验通过使用智能编程语言（BANGC）进行算子开发，对高性能库（CNML）算子进行扩展，并最终集成到编程框架（TensorFlow）中。这个实验的主要目的就是让大家熟悉使用 BANGC 语言进行 MLU 的算子开发，让大家认识到智能硬件对于目前一些特定任务，比如各种神经网络，卓越的加速作用。

实验目的

本实验通过智能编程语言实现 PowerDifference 算子，掌握使用智能编程语言进行算子开发，扩展高性能库算子，并最终集成到 TensorFlow 框架中的方法和流程，使得完整的风格迁移网络可以在 DLP 硬件上高效执行。

实验工作量：代码量约 150 行，实验时间约 10 小时。

背景介绍

编译器

CNCC 是将使用智能编程语言（BCL）编写的程序编译成 DLP 底层指令的编译器。为了填补高层智能编程语言和底层 DLP 硬件指令间的鸿沟，DLP 的编译器通过复杂寄存器分配、自动软件流水、全局指令调度等技术实现编译优化，以提升生成的二进制指令性能。开发者使用 BCL 开发自己的 DLP 端的源代码：首先通过前端 CNCC 编译为汇编代码，然后汇编代码由 CNAS 汇编器生成 DLP 上运行的二进制机器码。

调试器

CNGDB 是面向智能编程语言所编写程序的调试工具，能够支持搭载 DLP 硬件的异构平台调试，即同时支持 Host 端 C/C++ 代码和 Device 端 BCL 的调试，同时两者调试过程的切换对于用户而言也是透明的。此外，针对多核 DLP 架构的特点，调试器能同时支持单核和多核应用程序的调试。CNGDB 解决了异构编程模型调试的问题，提升了应用程序开发的效率。

为了使用 CNGDB 进行调试，在使用 CNCC 编译 BCL 文件时，需要使用-g 选项，在-O0 优化级别中来获取含有调试信息的二进制文件。

集成开发环境

CNStudio 是一款针对于 BCL 语言可在 Visual Studio Code 使用的编程插件，为了使 BCL 语言在编写过程中更加方便快捷，CNStudio 基于 VSCode 编译器强大的功能和简便的可视化操作提供包括语法高亮，自动补全和程序调试等功能。

BCL 算子库

CNPlugin 是一款包含了一系列 BCL 算子的高性能计算库。通过 CNPlugin 算子库机制可以帮助 BCL，CNML 与框架之间协同工作，有机融合。CNPlugin 在 CNML 层提供一个接口，将 BCL 语言生成的算子与 CNML 的执行逻辑统一起来。

为了将 BCL kernel 函数与 CNML 结合运行，CNML 提供了一套相关 API 来达到这个目的，通过这种 API 运行的算子被称为 PluginOp。

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：硬件平台基于前述的 DLP 云平台环境。
- 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。

实验内容

1. 算子实现：采用智能编程语言 BCL 实现 PowerDifference 算子并完成相应测试。首先，使用 BCL 的内置向量函数实现计算 Kernel，并利用 CNRT 接口直接调用 Kernel 运行并测试功能正确性；

2. 框架集成：通过高性能库 PluginOp 的接口对 PowerDifference 算子进行封装，使其调用方式和高性能库原有算子一致，将封装后的算子集成到 TensorFlow 框架中并进行测试，保证其精度和性能正确；

3. 在线推理：通过 TensorFlow 框架的接口，在内部高性能库 CNML 和运行时库 CNRT 的配合下，完成对风格迁移模型的在线推理，并生成离线模型；

4. 离线推理：采用运行时库 CNRT 的接口编写应用程序，完成离线推理，并将其结果和第三步中的在线推理进行对比。

实验步骤

算子实现

为了实现 PowerDifference 算子，需要完成 Kernel 程序编写、运行时程序编写、Main 程序编写和编译运行等步骤。

1. **Kernel 程序编写（plugin_power_difference_kernel.mlu）**

实验的主要内容需要完成__mlu_entry__ 函数供 CNRT 或 CNML 调用。这样可供调用的__mlu_entry__ 函数称为一个 Kernel。基于智能编程语言的 PowerDifference

(plugin_power_difference_kernel.mlu) 具体实现在实验代码中

这个部分需要使用智能编程语言 BCL 实现 PowerDifference 算子，即需要补全 /opt/AICSE-demo-student/demo/style_transfer_bcl/src/bangc/PluginPowerDifferenceOp/plugin_power_difference_kernel.h 和 plugin_power_difference_kernel.mlu 文件

向量乘法的代码在 BANGC 语言中对应的代码是

__bang_mul(output, input_1, input_2, LEN)

同时需要注意算子的参数顺序和类型，因为这个会影响到后续使用时导入参数的顺序问题。__mlu_entry__ void PowerDifferenceKernel(A, B, C, D, E)

这里可以看同文件路径下的 powerDiff.cpp 中，这个是后续需要使用的单算子测试程序，里面对于参数的导入顺序分别是输入 mlu_input1，输入 mlu_input2，次方数 pow，输出结果 mlu_output 以及向量长度 dims_a。这里推荐使用给出的参数顺序和数据类型，减少后续修改的代码数量。

```
cnrtKernelParamsBufferAddParam(params, &mlu_input1, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &mlu_input2, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &pow, sizeof(int));
cnrtKernelParamsBufferAddParam(params, &mlu_output, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &dims_a, sizeof(int));
```

这要求我们必须在计算前先使用 memcpy 将数据从 GDRAM 拷贝到 NRAM 上。在计算完成后也要将结果从 NRAM 拷贝到 GDRAM 上。第二是向量操作的输入规模必须对齐到 64 的整数倍。在这里程序将数据对齐到 256。

由于 NRAM 大小的限制，不能一次性将所有数据全部拷贝到 NRAM 上执行，因此需要对原输入规模进行分块。这里分块的规模在满足 NRAM 大小和函数对齐要求的前提下由用户指定，这里设置为 256（ONELINE）。分块的重点在于余数段的处理。由于通常情况下输入不一定是 256 的倍数，所以最后会有一部分长度小于 256，大于 0 的余数段。

2. **运行时程序编写（powerDiff.cpp）**

运行时程序通过利用运行时库 CNRT 的接口调用 BCL 算子来实现。首先声明被调用的算子实现函数，然后在 MLUPowerDifferenceOp 中通过一系列 CNRT 接口的调用完成，包括：使用 cnrtKernelParamsBuffer 来设置 PowerDifference 算子的输入参数，通过 cnrtInvokeKernel 来调用算子 Kernel 函数（PowerDifferenceKernel），最后完成计算后获取输出结果并销毁相应资源。

3. **Main 程序编写（main.cpp）**

Main 程序首先读取文件 in_x.txt 和 in_y.txt 中的数据加载到内存中，然后调用上一步定义的 MLUPowerDifferenceOp 函数对输入数据进行计算，并将结果输出到文件 out.txt 中。其中会统计计算时间，并得到和 CPU 运算结果相对比的错误率

4. **编译运行（power_diff_test）**

完成上述代码的编写后，需要编译运行该程序。调用编译器 CNCC 将算子实现函数编译成为 powerdiffkernel.o 文件，然后通过 Host 的 g++ 编译器，将其和 powerDiff.cpp，main.cpp 等文件一起编译链接成最终的 power_diff_test 可执行程序。

框架集成

为了将前述 PowerDifference 算子集成至 TensorFlow 框架中，需要完成 PluginOp 接口封装、DLP 算子集成和算子测试等步骤。

1. **PluginOp 接口封装**

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机制。因此，在完成 PowerDifference 算子的开发后，可以利用 CNML PluginOp 相关接口封装出方便用户使用的 CNPlugin 接口（包括 PluginOp 的创建、计算和销毁等接口），使用户自定义算子和高性能库已有算子有一致的编程模式和接口。

PluginOp 接口封装的部分示例代码，主要包括算子构建接口 Create、单算子运行接口 Compute 函数的具体实现。函数定义在 plugin_power_difference_op.cc 中，声明在 cnplugin.h 中。

算子构建接口 Create 函数：通过调用 cnmlCreatePluginOp 传递 BCL 算子函数指针、输入和输出变量指针完成算子创建。创建成功后可以得到 cnmlBaseOp_t 类型的指针。算子的相关参数需要使用 cnrtKernelParamsBuffer_t 的相关数据结构和接口创建。

单算子运行接口 Compute 函数：通过调用 cnmlComputePluginOpForward 利用前面创建的 cnmlBaseOp_t 的指针和输入输出变量指针完成上述计算过程。注意单独的 Compute 函数主要是在非融合模式下使用。

由于本算子的功能本身比较简单，所以参数（例如 power 和 len）采用了在 Create 时直接传递的方式。如果参数比较复杂则建议使用 OpParam 机制，将参数打包定义结构体来完成参数传递。

2. **DLP 算子集成**

为了使 DLP 硬件往 TensorFlow 框架中的集成更加模块化，我们对高性能库 CNML 算子进行了多个层次的封装，自顶向下包含以下几个层次：

- 最终运行的算子类 MLUOpKernel：继承 TensorFlow 中的 OpKernel 类，作为与 TensorFlow 算子层的接口；
- 封装 MLUStream 成员函数：与 MLUOpKernel 类接口关联，负责 MLU 算子的实例化并与运行时队列结合；
- 定义 MLUOps：负责 TensorFlow 算子的 DLP 实现，可以是单算子也可以是内存拼接的算子。完成对底层算子的调用后实现完整 TensorFlow 算子的功能供 MLUStream 部分调用；
- 对 CNPlugin 封装的 MLULib：对 CNML 和 CNRT 接口的直接封装供 MLUOps 调用，只包含极少的 TensorFlow 数据结构。

上述四个层次自顶向下连接了 TensorFlow 内部的 OpKernel 和 DLP 所提供的高性能库及运行时库，因此在 TensorFlow 中集成 DLP 算子涉及上面各层次。集成的整体流程主要包括：算子注册、定义 MLULib 层接口、定义 MLUOps 层接口、定义 MLUStream 层接口以及定义 MLUOpKernel 层接口并注册。

除了以上层次，还需要算子注册，定义 MLULib 层接口，定义 MLUOp 层接口，定义 MLUStream 层接口，定义 MLUOpKernal 层接口

3. **算子测试**

在新增自定义的 PowerDifference 算子与 TensorFlow 框架的集成完后，用户需要使用 Bazel 重新编译 TensorFlow，然后即可使用 Python 侧的 API 对新集成的算子功能进行测试。

由于对用户的 API 是一致的，用户在测试时需要通过环境变量来配置该算子的实现是调用 CPU 还是 DLP 版本。该部分代码位于 power_difference_test_bcl.py。

在线推理

针对完整的 pb 模型推理，在框架层集成了 DLP 算子后，在创建 TensorFlow 的执行图时，会自动将这些算子分配到 DLP 上计算，无需使用者显式指定。具体而言，使用新编译的 TensorFlow 重复执行一次即可。可以看到，新集成了 DLP 上的 PowerDifference 算子后，

整个 pb 模型可以完整地跑在 DLP 上，且性能相较于纯 CPU 版本和部分 CNML 版本都有显著的提升。

离线推理

通过前一小节的在线推理，可以得到不分段实时风格迁移的离线模型。在实际场景中，为了尽可能提高部署的效率，通常会选择离线部署的方式。离线与在线的区别在于其脱离了 TensorFlow 编程框架和高性能库 CNML，仅与运行时库 CNRT 相关，减少了不必要的开销，提升了执行效率。

在编写离线推理工程时，DLP 目前仅支持 C++ 语言。与在线推理相似，离线推理主要包含：输入数据前处理、离线推理及后处理。下面详细介绍具体的实现代码。

1. **主函数**

主函数主要用于串联整体流程，该部分代码位于 src/style_transfer.cpp。

2. **数据前处理**

常见的数据前处理包括减均值、除方差、图像大小 Resize、图像数据类型转换（例如 Float 和 INT 转换）、RGB 转 BGR 转换等等。具体需要哪些预处理需要与原神经网络模型对齐。以 Resize 操作为例，可以调用 OpenCV 中的 Resize 函数 cv::resize(sample, sample_resized, cv::Size(256,256))；该函数参数分别对应输入、输出和 Resize 的目标大小等。该部分代码位于 src/data_provider.cpp 中。

3. **离线推理**

离线推理部分主要是使用 CNRT API 运行离线模型。其主要流程包括以下步骤：

第一步将磁盘上的离线模型文件载入并抽取出 CNRT Function。一个离线模型文件中可以存储多个 Function，但是多数情况下离线模型文件中只有一个 Function，这取决于离线模型生成时框架层的设置。本实验中由于所有算子都可以在 DLP 上运行，经过 CNML 算子间融合处理之后只有一个 Function。

第二步要准备 Host 与 Device 的输入输出内存空间和数据。由于 DLP 的异构计算特征，需要先在 Host 端准备好数据后再将其拷贝到 Device 端，所以在此之前也要先分别在 Device 端和 Host 端分配相应内存空间。其中需要注意的是数据类型（例如 INT 或 Float）和存储格式（例如 NCHW 或 NHWC）在 Host 端和 Device 端之间可能会不同，所以在做数据拷贝前要先完成相应的转换。

第三步主要和 DLP 设备本身相关。包括设置运行时上下文、绑定设备、将计算任务下发到队列等。

第四步将计算结果拷回 Host 端并完成相关的数据转换。

最后一步将上面申请的所有内存和资源释放。上述代码位于 src/inference.cpp 中。

4. **后处理**

这部分主要完成将计算结果保存成图片，具体代码位于 src/post_processor.cpp 中。

5. **编译运行**

这里借助 CMake 工具完成对整个项目的编译管理，具体代码在 CMakeList.txt 中。

```
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 108.517169952ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 30.1489830017ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 30.0080776215ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 30.3950309753ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 31.0678482056ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 33.9748859406ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 34.970998764ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 34.9991321564ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 35.8290672302ms
CNML: 7.2.1 c8ada41
CNML: 7.2.1 c8ada41
comput BCL op cost 36.8809700012ms
comput op cost 211.896181107ms
err rate= 0.0754365216257073
(virtualenv_mlu) root@localhost:/home/AICSE-demo-student/demo/style_transfer_bcl/src/online_mlu# cd ../online_cpu/
(virtualenv_mlu) root@localhost:/home/AICSE-demo-student/demo/style_transfer_bcl/src/online_cpu# python power_difference_test_cpu.py
CNML: 7.2.1 c8ada41
CNRT: 4.2.1 fa5e44c
CNML: 7.2.1 c8ada41
comput C++ op cost 66.1339759827ms
comput numpy op cost 222.080945969ms
err rate= 6.1142082184421e-06
(virtualenv_mlu) root@localhost:/home/AICSE-demo-student/demo/style_transfer_bcl/src/online_cpu#
```

使用 CPU 时，算子消耗的时间是 66.134 毫秒。使用 MLU 时，BCL 算子的时间是 36.881 毫秒，错误率是 0.075%，低于 0.1%

```
1 comput BCL op cost 294.717073441ms
2 comput op cost 225.753068924ms
3 err rate= 5.8923983536261504e-06
4
```