智能计算系统
实验教程

智能计算系统课程团队

中国科学院计算技术研究所

目 录

第 1 章　智能编程语言

智能编程语言是连接智能编程框架和智能计算硬件的桥梁。本章将通过具体实验阐述智能编程语言的开发，优化和集成技巧。

具体而言，第 1.1节介绍如何用智能编程语言 BCL 实现用户自定义的高性能库算子（即 PowerDifference），并将其集成到 TensorFlow 框架中。希望读者可以通过本实验了解如何将 BCL 与智能编程框架集成，以将前述训练好的风格迁移网络模型编译为 DLP 硬件指令，满足智能计算系统的可扩展和高性能需求。第 1.3节介绍如何使用智能编程语言的 python 接口来开发 1.1节所介绍的高性能库算子（PowerDifference），使读者可以掌握智能编程语言算子的高效开发方法。第 1.2节介绍 BCL 的一些高级特性，使读者在实现功能的基础上进一步掌握充分发挥 DLP 硬件潜力的编程技巧。

1.1　智能编程语言算子开发与集成实验（BCL 开发实验）

1.1.1　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。

1.1.2　背景介绍

本节重点介绍面向智能编程语言开发所需的编译工具链，包括编译器、调试器及集成开发环境等。

1.1.2.1　编译器（CNCC）

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。

CNCC 的结构如图 1.1所示，开发者使用 BCL 开发自己的 DLP 端的源代码：首先通过 CNCC 前端编译为汇编代码，然后汇编代码由 CNAS 汇编器生成 DLP 上运行的二进制机器码。在实际操作中可以直接使用 CNCC 将 BCL 代码生成可执行文件，中间的步骤将由编译器自动完成。

在使用 CNCC 编译 BCL 文件时，有多个编译选项供开发者使用，如表 1.1所示。

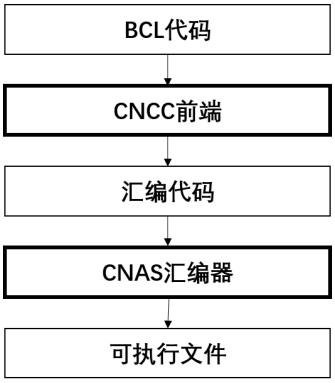

图 1.1　CNCC 结构示意图

表 1.1　CNCC 编译选项

常用选项	说明
-E	编译器只执行预处理的步骤，生成预处理文件
-S	编译器只执行预处理、编译的步骤，生成汇编文件
-c	编译器只执行预处理、编译、汇编的步骤，生成 ELF 格式的汇编文件
-o	将输出写入到指定的文件
-x	为输入的文件指定编程语言，如 BCL 等
–target=	指定生成 DLP 端的目标文件的格式，该文件和目标 Host 平台的目标文件一起链接成目标 Host 端的可执行程序，所以其值为目标 Host 端二进制文件格式，eg x86_64, armv7a 等
–bang-mlu-arch=	为输入的 BCL 程序指定 DLP 的架构
–bang-stack-on-ldram	栈是否放在 LDRAM 上，默认放在 NRAM 上，如果该选项开启，栈会放在 LDRAM 上
–cnas-path=	指定汇编器的路径

1.1.2.2　调试器（CNGDB）

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。下面是使用 CNCC 编译一个 BCL 文件的示例：

```
cncc kernel.mlu -o kernel.o --bang-mlu-arch=MLU270 -g -O0
```

这里以 BCL 写的快速排序程序为例, 演示如何使用 CNGDB 调试程序。图 1.2展示了 CNGDB 调试 recursion.mlu 程序的基本流程，总的来说主要包含如下几个步骤：断点插入、程序执行、变量打印、单步调试和多核切换等。

```
#1.在 CNCC 编译时开启 -g 选项，首先将 recursion.mlu 文件编译为带有调试信息的二进制文件:
cncc recursion.mlu -o recursion.o --bang-mlu-arch=MLU270 -g -O0

#2.然后继续编译得到可运行二进制文件:
g++ recursion.o main.cpp -o quick_sort -lcnrt -I${DLP_INC} -L${DLP_LIB}

#3.在有DLP板卡的机器上使用CNGDB打开quick_sort程序:
cngdb quick_sort

#4. 用 break 命令，在第 x 行添加断点:
(cn-gdb) b recursion.mlu :x

#5.用run命令，执行程序至断点处，此时程序执行至kernel函数的x行处(x行还未执行)
(cn-gdb) r

#6. 用 print 命令，分别查看第一次调用 x 行函数时的三个实参:
(cn-gdb) p input1
(cn-gdb) p input2
(cn-gdb) p input3

#7(a). 如果使用 continue 命令，程序会从当前断点处继续执行直到结束。如果不希望程序结束，可以继续添加断点:
(cn-gdb) c
#7(b). 如果希望进入被调用的某函数内部，可以直接使用 step 命令，达到单步调试的效果:
(cn-gdb) s

#8. 可以使用info args命令和info locals命令查看函数参数以及函数局部变量:
(cn-gdb) info args

#9.如果需要对不同核进行调试，通过切换焦点获取对应core的控制权:
(cngdb) cngdb focus Device 0 cluster 0 core 2
```

图 1.2　CNGDB 调试示例

1.1.2.3　集成开发环境（CNStudio）

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。

当前 CNStudio 插件只提供离线安装包，不支持在线安装，安装包的具体下载地址请参考本课程网站。其中，VSCode 版本要求 1.28.0 及以上版本。具体安装流程如图1.3所示。通过上述步骤找到对应的离线安装包，完成 CNStudio 插件的安装。

安装完毕后，在左侧插件安装界面的搜索框中输入 “@installed” 即可查询全部插件，若显示如图 1.4所示的插件则说明 CNStudio 安装成功。注意：CNStudio 的高亮颜色与 VScode 背景颜色会有冲突，可通过组合快捷键 (Ctrl+k) (Ctrl+t) 更改浅色主题。

在创建工程时（以新建一个 DLP 文件夹为例），由于每个 project 都包含三种类型的文件，需要在 DLP 文件夹中新建 DLP 端程序所需的 dlp.mlu（Device 端程序源文件后缀名为 “*.mlu”。安装 CNStudio 插件后，vscode 会自动识别 mlu 文件），Host 端程序所需的 main.cpp，以及头文件 kernel.h。可通过 VScode 工具栏中 “File” → “Save Workspace As…”，将打开的 DLP 工程保存为 workspace，方便下次直接打开工程文件。

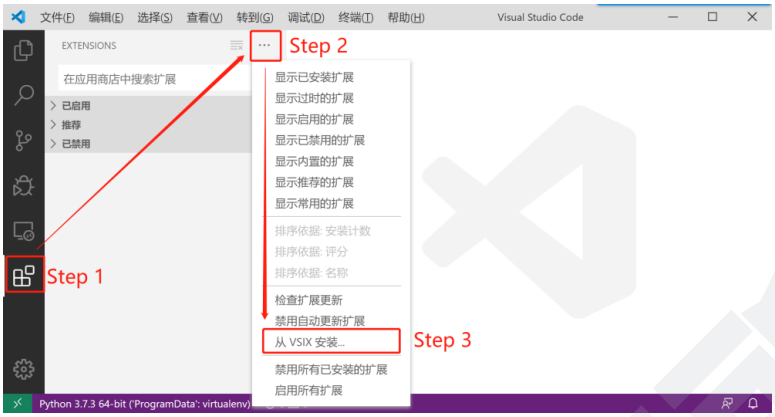

图 1.3　CNStudio 安装流程图

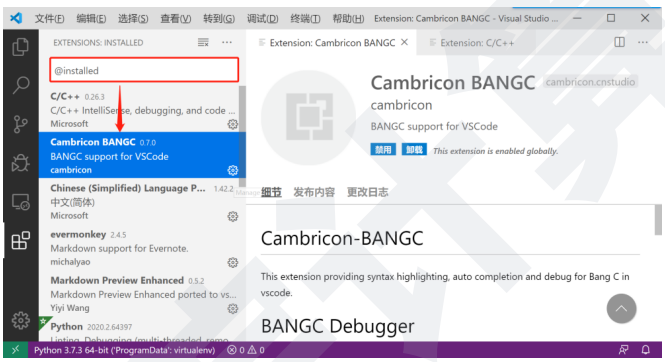

图 1.4　CNStudio 安装完成

1.1.2.4　BCL 算子库（CNPlugin）

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，如代码示例1.1所示，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。

代码示例 1.1　CNML PluginOp 相关的主要 API

```
CNML_DLL_API cnmlStatus_t cnmlCreatePluginOp(cnmlBaseOp_t *op,
const char *name,
void *kernel,
cnrtKernelParamsBuffer_t params,
cnmlTensor_t *input_tensors,
int input_num,
cnmlTensor_t *output_tensors,
int output_num,
cnmlTensor_t *statics_tensor,
int static_num);

CNML_DLL_API cnmlStatus_t cnmlComputePluginOpForward_V4(cnmlBaseOp_t op,
cnmlTensor_t input_tensors[],
void *inputs[],
```

```
int input_num,
cnmlTensor_t output_tensors[],
void *outputs[],
int output_num,
cnrtQueue_t queue,
void *extra);
```

在以上 CNML 的基础之上，如图1.5所示，CNPlugin 中的每个算子都包含了 Host 端代码（plugin_yolov3_detection_output_op.cc 文件）和 Device 端 BCL 代码（plugin_yolov3_detection_output_kernel_v 文件）。其中 Host 端代码主要完成了算子参数处理，CNML PluginOp API 封装和 BCL Kernel 调用等工作。Decive 端代码包含了 BCL 源码，实现了主要的计算逻辑。

```
Cambricon-CNPlugin-MLU270
├── build
├── build_aarch64.sh
├── build_cnplugin.sh
├── CMakeLists.txt
├── common
│   ├── include
│   │   ├── cnplugin.h
├── pluginops
│   ├── PluginYolov3DetectionOutputOp
│   │   ├── plugin_yolov3_detection_output_kernel_v2.h
│   │   ├── plugin_yolov3_detection_output_kernel_v2.mlu
│   │   └── plugin_yolov3_detection_output_op.cc
├── README.md
└── samplecode
```

图 1.5　CNPlugin 的主要目录结构示意

在 CNPlugin 中如果函数参数不多可以选择直接传参的方式，如果参数比较多则建议 OpParam 结构体来完成传参。具体包括了结构体内容，CreatePluginOpParam 函数和 cnmlDestroyPluginOpParam 函数。代码示例1.2是一个 Addpad 算子的 OpParam 示例。

代码示例 1.2　CNPlugin OpParam 示例

```

struct cnmlPluginAddpadOpParam {
int batch_size;
int src_h;
int src_w;
int dst_h;
int dst_w;
int type_uint8;
int type_yuv;
};

cnmlStatus_t cnmlCreatePluginAddpadOpParam(cnmlPluginAddpadOpParam_t *param_ptr,
int batch_size,
int src_h,
int src_w,
int dst_h,
int dst_w,
int type_uint8,
int type_yuv) {
```

```
20 *param_ptr = new cnmlPluginAddpadOpParam();
21 (*param_ptr)->batch_size = batch_size;
22 (*param_ptr)->src_h = src_h;
23 (*param_ptr)->src_w = src_w;
24 (*param_ptr)->dst_h = dst_h;
25 (*param_ptr)->dst_w = dst_w;
26 (*param_ptr)->type_uint8 = type_uint8;
27 (*param_ptr)->type_yuv = type_yuv;
28 return CNML_STATUS_SUCCESS;
29 }
30
31 cnmlStatus_t cnmlDestroyPluginAddpadOpParam(cnmlPluginAddpadOpParam_t param) {
32 delete param;
33 return CNML_STATUS_SUCCESS;
34 }
```

1.1.3 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。

1.1.4 实验内容

本节实验基于第**??**节中的高性能库算子实验，在前者基础上进一步把 PowerDifference 算子用智能编程语言实现，通过高性能库 PluginOp 接口扩展算子，并和高性能库原有算子一起集成到编程框架 TensorFlow 中，此后将风格迁移模型在扩展后的 TensorFlow 上运行，最后将其性能结果和第**??**节中的性能结果进行对比。实验内容和流程如图1.6所示，主要包括：

1. **BCL 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. **框架算子集成：**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. **模型在线推理：**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. **模型离线推理：**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，以及第**??**节中的推理性能进行对比。

图1.6中虚线框的部分是需要同学补充完善的实验文件。每一步实验操作需要修改的具体对应文件内容请参考下一节“实验步骤”。

1.1.5 实验步骤

如前所述，本实验的详细步骤包括：算子实现、框架集成、在线推理和离线推理等。

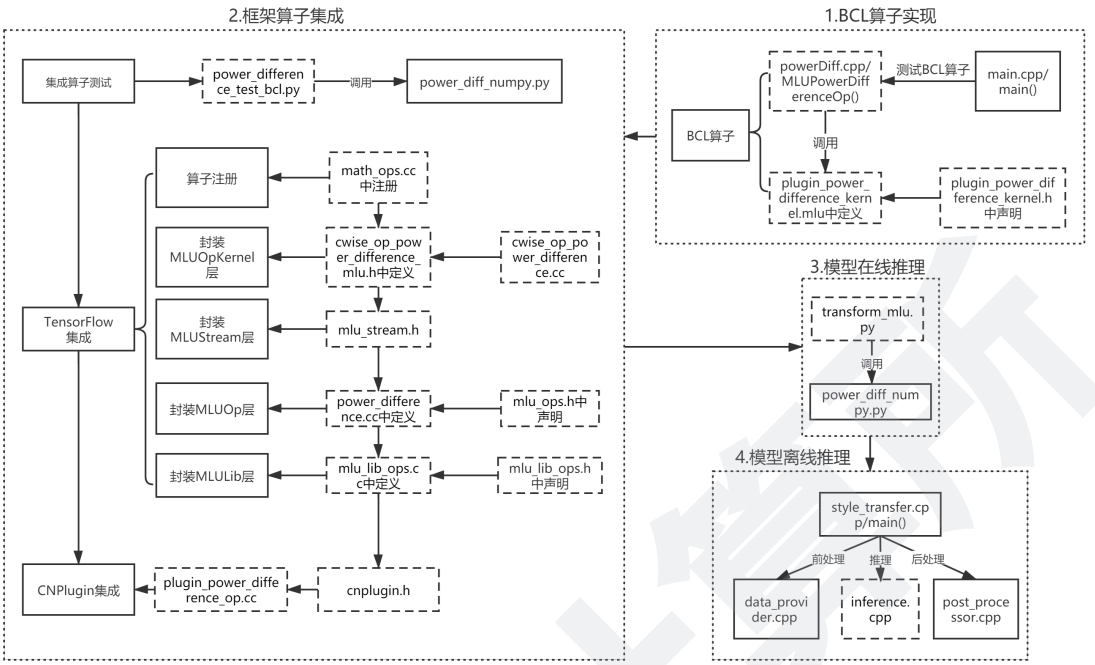

图 1.6　具体实验内容

1.1.5.1　BCL 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）**

本节实验的第一步是使用 BCL 实现 PowerDifference 算子。具体的算子公式请参考 **??**小节的内容。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) 具体实现如代码示例1.3所示。

代码示例 1.3　基于智能编程语言 BCL 的 Power Difference 实现

```
1   //filename: plugin_power_difference_kernel.mlu
2   //定义常量
3   #define ONELINE 256
4
5   __mlu_entry__ void PowerDifferenceKernel(half* input1, half* input2, int pow, half* output, int
     len)
6   {
7   int quotient = len / ONELINE;
8   __bang_printf("%d %d\n", pow, len);
9   int rem = len % ONELINE;
10
11  //声明NRAM空间
12  __nram__ half input1_nram[ONELINE];
13  __nram__ half input2_nram[ONELINE];
14
15  if ( rem != 0)
16  {
17  quotient +=1;
```

```
    }
    for (int i = 0; i < quotient; i++)
    {
    //内存拷贝：从GDRAM的（input1 + i * ONELINE）位置开始，拷贝ONELINE * sizeof(half)大小的数据到
     input1_nram空间中。
    __memcpy(_____________________________);
    __memcpy(_____________________________);
    //按元素减法操作，将input1_nram和input2_nram的对应元素进行相减并储存在input1_nram中。
    __bang_sub(_____________________________);
    //NRAM中两个数据块的数据拷贝操作
    __memcpy(_____________________________);
    for (int j = 0; j < pow - 1; j++)
    {
    //按元素相乘操作
    __bang_mul(_____________________________);
    }
    //内存拷贝：从NRAM中将计算的结果拷出至GDRAM中。
    __memcpy(_____________________________);
    }
    }

```

在上述代码中，为了充分发挥算子性能，使用了向量计算函数来完成运算。为了使用向量计算函数必须要满足两个前提。第一是调用计算时数据的输入和输出存放位置必须在 NRAM 上。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。读者在完成实验时需注意该部分数据的处理逻辑。

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。如代码示例1.4所示，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。

代码示例 1.4　调用运行时库函数

```
    //filename: powerDiff.cpp
    #include <stdlib.h>
    #include "cnrt.h"
    #include "cnrt_data.h"
    #include "stdio.h"
    #ifdef __cplusplus
    extern "C" {
    #endif
    void PowerDifferenceKernel(half* input1, half* input2, int32_t pow, half* output, int32_t len);
    #ifdef __cplusplus
    }
    #endif
```

```
void PowerDifferenceKernel(half* input1, half* input2, int32_t pow, half* output, int32_t len);

int MLUPowerDifferenceOp(float* input1,float* input2, int pow, float*output, int dims_a) {
//some definition
//prepare data
if (CNRT_RET_SUCCESS != cnrtMalloc((void**)&mlu_input1, dims_a * sizeof(half))) {
printf("cnrtMalloc Failed!\n");
exit(-1);
}
...
//kernel parameters
cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &mlu_input1, sizeof(half*));
...
//启动Kernel
cnrtInvokeKernel_V2(_____________________________);
//将计算结果拷回Host
cnrtMemcpy(_____________,CNRT_MEM_TRANS_DIR_DEV2Host);
cnrtConvertHalfToFloatArray(_____________________);
//free data
cnrtDestroy();
...
return 0;
}
```

3. **Main 程序编写（main.cpp）**

如代码示例1.5所示，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。

代码示例 1.5　Main 程序

```
//filename: main.cpp
#include <math.h>
#include <time.h>
#include "stdio.h"
#include <stdlib.h>
#include <sys/time.h>

#define DATA_COUNT 32768
#define POW_COUNT //some num
int MLUPowerDifferenceOp(float* input1,float* input2, int pow, float*output, int dims_a);

int main() {
//define
...
FILE* f_input_x = fopen("./data/in_x.txt", "r");
FILE* f_input_y = fopen("./data/in_y.txt", "r");
FILE* f_output_data = fopen("./data/out.txt", "r");
struct timeval tpend, tpstart;
//load data
...
//compute
gettimeofday(&tpstart, NULL);
MLUPowerDifferenceOp(input_x,input_y,POW_COUNT,output_data,DATA_COUNT);
gettimeofday(&tpend, NULL);
time_use = 1000000 * (tpend.tv_sec - tpstart.tv_sec)+ tpend.tv_usec - tpstart.tv_usec;
for(int i = 0; i < DATA_COUNT;++i)
{
```

```
    err +=fabs(output_data_cpu[i] - output_data[i]) ;
    cpu_sum +=fabs(output_data_cpu[i]);
    }
    printf("err rate = %0.4f%%\n", err*100.0/cpu_sum);
    return 0;
    }

```

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。具体的编译命令如下所示：

```
cncc -c --bang-mlu-arch=MLU200 plugin_power_difference_kernel.mlu -o powerdiffkernel.o
g++ -c main.cpp
g++ -c powerDiff.cpp -I/usr/local/neuware/include
g++ powerdiffkernel.o main.o powerDiff.o -o power_diff_test -L /usr/local/neuware/lib64 -lcnrt
```

其中，首先调用编译器 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。

1.1.5.2 框架算子集成

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。以下将以自底向上的顺序详述各步骤的内容。

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。

代码示例 1.6　PluginOp 接口封装

```
//filename: plugin_power_difference_op.cc

//cnmlCreatePluginPowerDifferenceOp
cnmlStatus_t cnmlCreatePluginPowerDifferenceOp(
```

```
    cnmlBaseOp_t *op,
    cnmlTensor_t* input_tensors,
    int power,
    cnmlTensor_t* output_tensors,
    int len
    ) {
    void** InterfacePtr;
    InterfacePtr = reinterpret_cast<void**>(&PowerDifferenceKernel);
    // Passing param
    cnrtKernelParamsBuffer_t params;
    cnrtGetKernelParamsBuffer(&params);
    cnrtKernelParamsBufferMarkInput(params);    // input 0
    cnrtKernelParamsBufferMarkInput(params);    // input 1
    cnrtKernelParamsBufferAddParam(params, &power, sizeof(int));
    cnrtKernelParamsBufferMarkOutput(params);   // output 0
    cnrtKernelParamsBufferAddParam(params, &len, sizeof(int));
    cnmlCreatePluginOp(op,
    "PowerDifference",
    InterfacePtr,
    params,
    input_tensors,
    2,
    output_tensors,
    1,
    nullptr,
    0);
    cnrtDestroyKernelParamsBuffer(params);
    return CNML_STATUS_SUCCESS;
    }
    // cnmlComputePluginPowerDifferenceOpForward
    cnmlStatus_t cnmlComputePluginPowerDifferenceOpForward(
    cnmlBaseOp_t op,
    void **inputs,
    void **outputs,
    cnrtQueue_t queue
    ) {
    cnmlComputePluginOpForward_V4(op,
    nullptr,
    inputs,
    2,
    nullptr,
    outputs,
    1,
    queue,
    nullptr);
    return CNML_STATUS_SUCCESS;
    }
```

2. **DLP 算子的框架集成**

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 或 CNPlugin 算子进行了多个层次的封装，如图 1.7所示，自底向上包含以下几个层次：

- 封装 MLULib 层：对 CNML 和 CNPlugin 接口的直接封装供 MLUOps 调用，只包含极少的 TensorFlow 数据结构。
- 封装 MLUOp 层：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；

• 封装 MLUStream 层：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；

• 封装 MLUOpKernel 层：定义并注册最终运行的算子类 MLUOpKernel，继承 TensorFlow 中的 OpKernel 类，作为与 TensorFlow 算子层的接口；

• 算子注册：完成最终的算子注册供上层应用调用。

上述五个层次自底向上连接了 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。集成的整体流程如图 1.7所示。

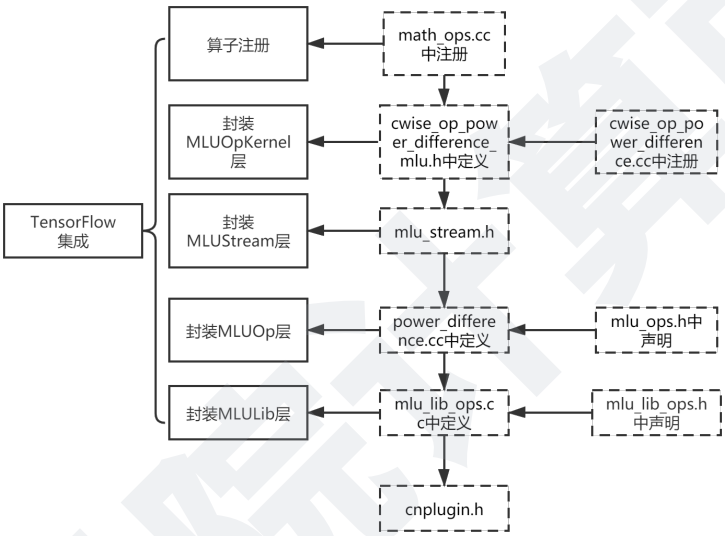

图 1.7　TensorFlow 集成流程图

• **封装 MLULib 层**

定义 MLULib 层接口主要是将前述已通过 PluginOp 接口封装好的接口如 cnmlCreatePluginPowerDifferenceOp 和 cnmlComputePluginPowerDifferenceOpForward 与 TensorFlow 中的 MLULib 层接口进行绑定，实现 MLULib 层的 CreatePowerDifferenceOp 和 ComputePowerDifferenceOp。该部分代码位于 tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h、tensorflow/stream_executor/m 和 tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h 中，如代码示例1.7所示。

代码示例 1.7　封装 MLULib 层

```
1        ##tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h
2        tensorflow::Status CreatePowerDifferenceOp(MLUBaseOp** op, MLUTensor* input1, MLUTensor*
     input2, int input3, MLUTensor* output, int len);
3        tensorflow::Status ComputePowerDifferenceOp(MLUBaseOp* op, MLUCnrtQueue* queue, void* input1,
     void* input2, void* output);
4
```

```
    ##tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc
    tensorflow::Status CreatePowerDifferenceOp(MLUBaseOp** op, MLUTensor* input1, MLUTensor*
input2, int input3, MLUTensor* output, int len) {
    MLUTensor* inputs_ptr[2] = {input1, input2};
    MLUTensor* outputs_ptr[1] = {output};
    CNML_RETURN_STATUS(cnmlCreatePluginPowerDifferenceOp(op, inputs_ptr, input3, outputs_ptr, len)
);
    }

    tensorflow::Status ComputePowerDifferenceOp(MLUBaseOp* op, MLUCnrtQueue* queue, void* input1,
void* input2, void* output) {
    void* inputs_ptr[2] = {input1, input2};
    void* outputs_ptr[1] = {output};
    CNML_RETURN_STATUS(cnmlComputePluginPowerDifferenceOpForward(op, inputs_ptr, outputs_ptr,
queue));
    }

    ##tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h 中添加声明
    DECLARE_OP_CLASS(MLUPowerDifference);

```

- **封装 MLUOp 层**

定义 MLUOp 层接口主要是在 MLUOp 层实现算子类的 Create 和 Compute 等方法。该部分代码位于 tensorflow/stream_executor/mlu/mlu_api/ops/power_difference.cc 文件中，如代码示例1.8所示。其中 CreateMLUOp 和 Compute 等方法将调用前面在 MLULib 层实现好的 CreatePowerDifferenceOp 和 ComputePowerDifferenceOp 等方法。

代码示例 1.8　封装 MLUOp 层

```
    //filename: tensorflow/stream_executor/mlu/mlu_api/ops/power_difference.cc
    Status MLUPowerDifference::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");
    MLUBaseOp *power_difference_op_ptr = nullptr;
    MLUTensor *input1 = inputs.at(0);
    MLUTensor *input2 = inputs.at(1);
    int power_c = *((int*)param);
    MLUTensor *output = outputs.at(0);
    ...
    TF_STATUS_CHECK(lib::CreatePowerDifferenceOp(_________________));
    ...
    }
    Status MLUPowerDifference::Compute(const std::vector<void *> &inputs, const std::vector<void
*> &outputs, cnrtQueue_t queue) {
    ...
    TF_STATUS_CHECK(lib::ComputePowerDifferenceOp(_________________));
    ...
    }

```

- **封装 MLUStream 层**

定义 MLUStream 层接口主要是在 MLUStream 层(tensorflow/stream_executor/mlu/mlu_stream.h)添加算子类声明，如代码示例1.9所示。其与 MLUOpKernel 类接口关联，负责 MLU 算子的实例化。在运行时这层代码会自动将算子与运行时队列进行绑定并下发执行。

代码示例 1.9　封装 MLUStream 层

```
//filename: tensorflow/stream_executor/mlu/mlu_stream.h
Status PowerDifference(OpKernelContext* ctx,
Tensor* input1, Tensor* input2, Tensor* output, int input3) {
return CommonOpImpl<ops::MLUPowerDifference>(ctx,
{input1, input2}, {output}, static_cast<void*>(&input3));
}

```

- **封装 MLUOpKernel 层**

定义 MLUOpKernel 层接口主要是在 MLUOpKernel 层定义 MLUPowerDifferenceOp，在其中通过 stream 机制调用 MLUStream 层具体的 PowerDifference 函数。该部分代码位于 tensorflow/core/kernels/cwise_op_power_difference_mlu.h，如代码示例1.10所示。

代码示例 1.10　封装 MLUOpKernel 层

```
//filename: tensorflow/core/kernels/cwise_op_power_difference_mlu.h
class MLUPowerDifferenceOp : public MLUOpKernel {
public:
explicit MLUPowerDifferenceOp(OpKernelConstruction* ctx) :
MLUOpKernel(ctx) {}
void ComputeOnMLU(OpKernelContext* ctx) override {
//输入数据处理与条件判断
--------------------------------------------------
//Stream调用PowerDifference接口
OP_REQUIRES_OK(ctx, stream->PowerDifference(________));

```

- **算子注册**

参考第**??**节中注册的 CPU 算子，在 tensorflow/core/kernels/cwise_op_power_difference.cc 文件中添加如图 **??**所示的 DLP 算子 Kernel 的注册信息（REGISTER_KERNEL_BUILDER）。此外，DLP 算子会与 CPU 算子共享在 tensorflow/core/ops/math_ops.cc 中的算子注册方法（代码示例**??**所示的 REGISTER_OP），这样用户可以使用相同的 Python API（power_difference）调用自定义算子，在编程上无需感知底层硬件的差异。

代码示例 1.11　算子注册

```
//filename: tensorflow/core/kernels/cwise_op_power_difference.cc
#define REGISTER_MLU(T)                \
REGISTER_KERNEL_BUILDER(               \
Name("PowerDifference")          \
.Device(Device_MLU)          \
.TypeConstraint<T>("T"),   \
MLUPowerDifferenceOp<T>);
TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);

```

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。完整的单算子 Python 测试代码如代码示例1.12所示。

代码示例 1.12　采用 Python API 对集成的单算子进行测试

```
#power_difference_test_bcl.py
import numpy as np
import os
import time
#使用以下环境变量控制单算子的执行方式
os.environ['MLU_VISIBLE_DeviceS']="0"
os.environ['TF_CPP_MIN_MLU_LOG_LEVEL']="1"
import tensorflow as tf
np.set_printoptions(suppress=True)

def power_difference_op(input_x,input_y,input_pow):
with tf.Session() as sess:
x = tf.placeholder(tf.float32, name='x')
y = tf.placeholder(tf.float32, name='y')
pow_ = tf.placeholder(tf.float32, name='pow')
z = tf.power_difference(x,y,pow_)
return sess.run(z, feed_dict = {x: input_x, y: input_y, pow_: input_pow})

def main():
start = time.time()
input_x = np.loadtxt("./data/in_x.txt",delimiter=',')
input_y = np.loadtxt("./data/in_y.txt")
input_pow = np.loadtxt("./data/in_z.txt")
output = np.loadtxt("./data/out.txt")
end = time.time()
print("load data cost "+ str((end-start)*1000) + "ms" )
start = time.time()
res = power_difference_op(input_x, input_y, input_pow)
end = time.time()
print("comput op cost "+ str((end-start)*1000) + "ms" )
err = sum(abs(res - output))/sum(output)
print("err rate= "+ str(err*100))

if __name__ == '__main__':
main()

```

1.1.5.3 模型在线推理

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，只需在第 **??**节的实验基础上，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（第 **??**节）和部分 CNML 版本（第 **??**节）都有显著的提升。

1.1.5.4 模型离线推理

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. main 函数

main 函数主要用于串联整体流程，如代码示例1.13所示。

代码示例 1.13　DLP 离线部署 main 函数

```
//filename: src/style_transfer.cpp
#include "style_transfer.h"
#include <math.h>
#include <time.h>
#include "stdio.h"
#include <stdlib.h>
#include <sys/time.h>

int main(int argc, char** argv){
// parse args
std::string file_list = "/path/to/images/" + std::string(argv[1]) + ".jpg";
std::string offline_model = "/path/to/models/offline_models/" + std::string(argv[2]) + ".cambricon
 ";

//creat data
DataTransfer* DataT =(DataTransfer*) new DataTransfer();
DataT->image_name = argv[1];
DataT->model_name = argv[2];
//process image
DataProvider *image = new DataProvider(file_list);
image->run(DataT);

//running inference
Inference *infer = new Inference(offline_model);
infer->run(DataT);

//postprocess image
PostProcessor *post_process = new PostProcessor();
post_process->run(DataT);

delete DataT;
DataT = NULL;
delete image;
image = NULL;
delete infer;
infer = NULL;
delete post_process;
post_process = NULL;
}

```

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，如代码示例1.14所示。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。

代码示例 1.14　DLP 离线部署数据前处理

```
//filename: src/data_provider.cpp
#include "data_provider.h"

namespace StyleTransfer{
DataProvider :: DataProvider(std::string file_list_){
...
```

```
    set_mean();
    }
    void DataProvider :: set_mean(){
    float mean_value[3] = {
    0.0,
    0.0,
    0.0,
    };
    cv::Mat mean(256, 256, CV_32FC3, cv::Scalar(mean_value[0], mean_value[1], mean_value[2]));
    mean_ = mean;
    }
    bool DataProvider :: get_image_file(){
    image_list.push_back(file_list);
    return true;
    }
    cv::Mat DataProvider :: convert_color_space(std::string file_path){
    cv::Mat sample;
    cv::Mat img = cv::imread(file_path, -1);
    ...
    return sample;
    }
    cv::Mat DataProvider :: resize_image(const cv::Mat& source){
    cv::Mat sample_resized;
    cv::Mat sample;
    ...
    return sample_resized;
    }
    cv::Mat DataProvider :: convert_float(cv::Mat img){
    cv::Mat float_img;
    ...
    return float_img;
    }
    cv::Mat DataProvider :: subtract_mean(cv::Mat float_image){...}
    void DataProvider :: split_image(DataTransfer* DataT){...}
    DataProvider :: ~DataProvider(){}

    void DataProvider :: run(DataTransfer* DataT){
    for(int i = 0; i < batch_size; i++){
    get_image_file();
    std::string img_path= image_list[i];
    cv::Mat img_colored = convert_color_space(img_path);
    cv::Mat img_resized = resize_image(img_colored);
    cv::Mat img_floated = convert_float(img_resized);
    DataT->image_processed.push_back(img_floated);
    }
    split_image(DataT);
    }
    }

```

3. CNRT 离线推理

离线推理部分主要是使用 CNRT API 运行离线模型，如代码示例1.15所示。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。

代码示例 1.15　DLP 离线部署推理

```
//filename: src/inference.cpp
#include "inference.h"
#include "cnrt.h"
...
namespace StyleTransfer{
Inference :: Inference(std::string offline_model){
offline_model_ = offline_model;
}
void Inference :: run(DataTransfer* DataT){
// load model
// load extract function
// prepare data on cpu
// allocate I/O data memory on DLP
// prepare input buffer
// prepare output buffer
// setup runtime ctx
// bind Device
// compute offline
// free memory spac
}
} // namespace StyleTransfer

```

4. 后处理

这部分主要完成将计算结果保存成图片，如代码示例1.16所示。

代码示例 1.16　DLP 离线部署后处理

```
//filename: src/post_processor.cpp
#include "post_processor.h"

namespace StyleTransfer{

PostProcessor :: PostProcessor(){
std::cout << "PostProcessor constructor" << std::endl;
}

void PostProcessor :: save_image(DataTransfer* DataT){

std::vector<cv::Mat> mRGB(3);
for(int i = 0; i < 3; i++){
cv::Mat img(256, 256, CV_32FC1, DataT->output_data + 256 * 256 * i);
mRGB[i] = img;
}
cv::Mat im(256, 256, CV_8UC3);
cv::merge(mRGB,im);

```

```
    std::string file_name = DataT->image_name + std::string("_") + DataT->model_name + ".jpg";
    cv::imwrite(file_name, im);
    std::cout << "style transfer result file: " << file_name << std::endl;
    }

    PostProcessor ::~PostProcessor(){
    std::cout << "PostProcessor destructor" << std::endl;
    }

    void PostProcessor :: run(DataTransfer* DataT){
    save_image(DataT);
    }

    } // namespace StyleTransfer

```

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，如代码示例1.17所示。

代码示例 1.17　DLP 离线部署代码编译的 CMakeLists.txt 示例

```
//filename: CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(style_transfer)

set(CMAKE_BUILD_TYPE "Debug")
set(CMAKE_CXX_FLAGS_DEBUG "-std=c++11 -g -Wall ${CMAKE_CXX_FLAGS_DEBUG}")
set(CMAKE_EXE_LINKER_FLAGS "-lpthread -fPIC ${CMAKE_EXE_LINKER_FLAGS}")

find_package(OpenCV REQUIRED COMPONENTS core imgproc highgui)
include_directories(${OPENCV_INCLUDE_DIR})
include_directories($ENV{NEUWARE}/include/)
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include)

#link_directories(${CMAKE_CURRENT_SOURCE_DIR}/lib)
link_directories($ENV{X86_LIB_PATH})
link_directories($ENV{NEUWARE}/lib64/)
link_libraries("libcnrt.so")
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_CURRENT_SOURCE_DIR}/bin)

add_executable(style_transfer src/style_transfer.cpp
src/data_provider.cpp
src/inference.cpp
src/post_processor.cpp)

target_link_libraries(style_transfer ${OpenCV_LIBS})

```

1.1.6　实验评估

本次实验中主要考虑基于智能编程语言的算子实现与验证、与框架的集成以及完整的模型推理。模型推理的性能和精度应同时作为主要的参考指标。因此，本实验的评估标准设定如下：

- 60 分标准：完成 PowerDifferenece 算子实现以及基于 CNRT 的测试，在测试数据中精度误差在 1% 以内，延时在 100ms 以内；

• 80 分标准：在 60 分基础上，完成 BCL 算子与 TensorFlow 框架的集成，使用 Python 在 Device 端测试大规模数据时，精度误差在 10% 以内，平均延时在 150ms 以内。

• 90 分标准：在 80 分基础上，使用 DLP 推理完整 pb 模型时，输出精度正常的风格迁移图片，输出正确的离线模型。

• 100 分标准：在 90 分基础上，完成离线推理程序的编写，执行离线推理时风格迁移图片精度正常。

1.1.7 实验思考

1. PowerDifference 算子实现本身性能提升有哪些方法?
2. 融合方式为何可以带来性能的提升?
3. 离线方式为何可以带来性能的提升?
4. 如何更好地利用 DLP 的多核架构来提升性能?

1.2 智能编程语言性能优化实验

1.2.1 实验目的

掌握使用智能编程语言优化算法性能的原理，掌握智能编程语言的调试和调优方法，能够使用智能编程语言在 DLP 上加速矩阵乘的计算。

实验工作量：约 70 行代码，6 小时。

1.2.2 背景介绍

1.2.2.1 BCL 编程模型

BCL 编程模型的主要内涵在智能计算系统教材中已经全面阐述过了，在这里简单回顾一下和本实验相关的概念。

如图1.8所示,在本实验中同学们需要手动完成数据在 Global Memory (GDRAM), SRAM, NRAM，WRAM 之间的调度。

在 DLP 中，每个 cluster 中包含 4 个计算核，1 个 SRAM。需要注意的是，在每个 Cluster 中和 SRAM 相配合的还有 1 个 Memory Core 的部件，专门用于管理片上总线和 SRAM。与图中普通的计算 Core 相比，Memory Core 本身不承担计算任务，但是可以独立运行，完成数据从 GDRAM 到 SRAM 的拷贝。所以，在实际运行时 Memory Core 可以承担一部分数据搬运的任务，减轻计算 Core 的负担。在具体编程时，只要调用 BCL 的 memcpy 函数，如果拷贝方向是 GDRAM2SRAM，那么在编译器编译代码时就会自动将相关指令放到 Memory core 上去执行。

在编程时可以在程序中指定运行一次任务调用的计算资源数量。特别的我们称一次执行只调用一个计算核的任务为 **BLOCK** 任务。一次执行只调用一个 Cluster 的任务为 **UNION1** 任务，对应调用两个 Cluster 与四个 Cluster 分别为 UNION2 和 UNION4。

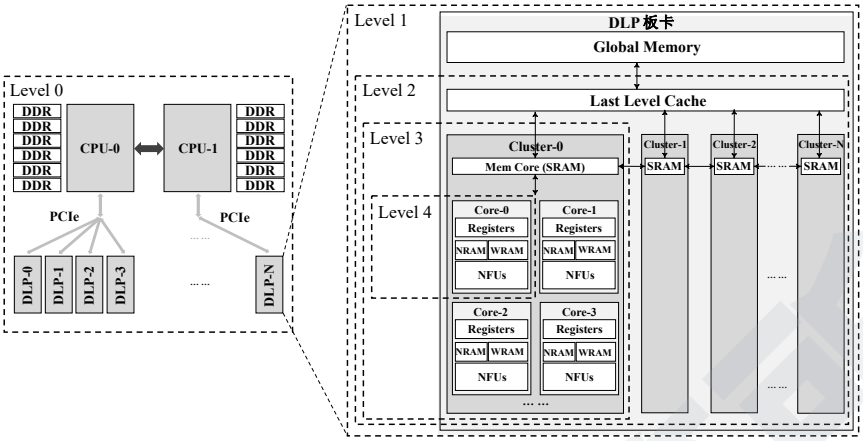

图 1.8　BCL 编程模型

1.2.2.2　DLP 并行编程

在 BCL 中使用一系列的并行内嵌变量来帮助使用者充分发挥 DLP 的并行特征。

Core 变量：

coreDim 表示一个 Cluster 包含的 Core 个数，MLU270 上等于 4

coreId 表示每个 Core 在 Cluster 内的逻辑 ID，MLU270 上的取值范围为 [0-3]

Cluster 变量：

clusterDim 表示启动 Kernel 时指定的任务调度型表示的 Cluster 个数，例如 UNION4 时等于 4

clusterId 表示 clusterDim 内某个 Cluster 的逻辑 ID，例如 UNION4 时其取值范围是 [0-3]

Task 变量：

taskDimX/taskDimY/taskDimZ 表示 1 个 Task 在 [XYZ] 方向上的任务规模，其值等于 Host 端所指定的任务规模

taskDim 表示任务线性化后的任务规模信息，线性化公式如下：

$$taskDim = taskDimX * taskDimY * taskDimZ$$

taskIdX/taskIdY/taskIdZ 表示程序运行时所分配的逻辑规模在 [XYZ] 方向上的任务 ID

taskId 表示程序运行时所分配的任务 ID，其值为对逻辑规模降维后的任务 ID：

$$taskId = taskIdZ * taskDimY * taskDimX + taskIdY * taskDimX + taskIdX$$

如图1.2是一个实际的内嵌变量取值示例。当程序调用 8 个计算核（UNION2）的时候每个核上的并行变量就会得到如图所示的取值。这里 taskDimX,Y,Z 设为 {8,1,1}。

表 1.2　DLP 并行内嵌变量示例

taskId	taskIdX	taskIdY	taskIdZ	clusterDim	coreDim	coreId	clusterID	taskDimX	taskDimY	taskDimZ	taskDim
0	0	0	0	2	4	0	0	8	1	1	8
1	1	0	0	2	4	1	0	8	1	1	8
2	2	0	0	2	4	2	0	8	1	1	8
3	3	0	0	2	4	3	0	8	1	1	8
4	4	0	0	2	4	0	1	8	1	1	8
5	5	0	0	2	4	1	1	8	1	1	8
6	6	0	0	2	4	2	1	8	1	1	8
7	7	0	0	2	4	3	1	8	1	1	8

1.2.2.3　Notifier 机制

在本实验中不再要求框架集成等系统开发内容，主要思路集中在智能编程语言的使用和优化。为了详细地统计程序在 DLP 上的运行时间，在此我们介绍 Notifier 机制。

Notifier 可以看作一种特殊类型的任务，它可以像 Kernel 任务一样放入 Queue 中执行。无论是 Notifier 还是 Kernel，内部队列始终遵循 FIFO 调度原则。与 Kernel 任务相比，Notifier 任务不需要执行实际的硬件操作，只占用很少的执行时间（几乎可以忽略不计）。可以使用 Notifier 任务来统计 Kernel 计算任务的硬件执行时间。代码示例1.18是对 ROIPoolingKernel 的执行时间进行统计的示例。

代码示例 1.18　Notifier 机制

```
cnrtNotifier_t notifier_start, notifier_end;
cnrtCreateNotifier(&notifier_start);
cnrtCreateNotifier(&notifier_end);
cnrtPlaceNotifier(notifier_start, pQueue);
ret = cnrtInvokeKernel_V3(reinterpret_cast<void *>(&ROIPoolingKernel), init_param, dim, params, c,
    pQueue, NULL);
cnrtPlaceNotifier(notifier_end, pQueue);
ret = cnrtSyncQueue(pQueue);
cnrtNotifierElapsedTime(notifier_start, notifier_end, &timeTotal);
printf("Hardware Total Time: %.3f ms\n", timeTotal / 1000.0);
```

1.2.3　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语言及编译器，高性能库 CNML 和运行时库 CNRT 仅作参考学习。

1.2.4　实验内容

本节实验内容主要为矩阵乘法的性能优化。需要完成如图1.9的矩阵乘法 Host 端和 Device 端的代码并正确编译运行。本实验中的 Device 端代码只接受特定规模的输入。

在计算结果正确的基础上，需要进一步使用多种手段完成性能优化，提升计算效率。主要的优化手段包括：

• 充分利用 DLP 的片上存储架构，包括 NRAM 和 WRAM 等特性实现高效数据调度。

• 在片上数据调度的基础上使用 __bang_conv 张量操作函数实现高效矩阵乘运算。

• 在充分利用单核计算架构（架构特点包括了 NRAM、WRAM 和张量计算指令等）的基础上，借助 DLP 本身的多核架构实现任务并行。

• 在朴素的多核并行的基础上，更进一步使用 Cluster 架构，使用片上 SRAM 达到减轻核间访存带宽竞争的目的。

• 充分使用片上 Memory Core 架构实现访存和计算流水化作业。

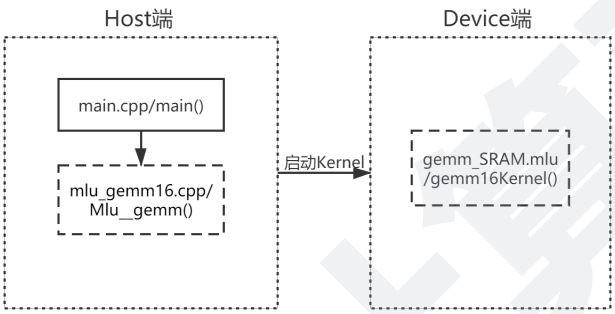

图 1.9　实验内容

本节实验涉及到的文件比较简单，图1.9所示，主要为 Host 端的 cpp 文件和 Device 端的 mlu 文件。

1.2.5　实验步骤

结合前述优化手段，本节实验的具体步骤包括：Host 端程序实现、标量操作实现、张量操作实现、多核并行实现、SRAM 使用、访存和计算流水等。

1.2.5.1　Host 端程序

由于智能编程语言采用异构编程，一个完整的程序包括 Host 端和 Device 端。Host 端和 Device 端分别进行编程和编译，最后链接成一个可执行程序。Host 端使用 C/C++ 语言进行编写，调用 CNRT 接口执行控制部分和串行任务；Device 端使用 BCL 特定的语法规则执行计算部分和并行任务。用户可以在 Host 端输入数据，做一定处理后，通过一个 Kernel 启动函数将相应输入数据传给 Device 端，Device 端进行计算后，再将计算结果拷回 Host 端。

在整个矩阵乘程序执行过程中，用户先输入参数 m, k, n 代表要计算的左右矩阵分别为 $m * k$ 和 $k * n$ 大小，随后 Host 端对这两个矩阵进行随机赋值，将输入矩阵及大小相应的参数传入 Device 端进行矩阵运算，然后将运算结果传回 Host 端，在 Host 端打印矩阵乘的硬件执行时间。Host 端代码位于 main.cpp 和 mlu_gemm16.cpp。其关键代码如 矩阵乘 Host 端代码示例所示：

代码示例 1.19　矩阵乘 Host 端代码示例

```
1
```

```
//filename: mlu_gemm16.cpp
#include <float.h>
#include <math.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <vector>
#include "cnrt.h"
#include "gemm16Kernel.h"

#define PAD_UP(x, m) ((x + m - 1) / m * m)
#define MP_SELECT 16
#define MP1 ((MP_SELECT & 1))
#define MP4 ((MP_SELECT & 4))
#define MP8 ((MP_SELECT & 8))
#define MP16 ((MP_SELECT & 16))
#define MP32 ((MP_SELECT & 32))
int Mlu_gemm(int8_t *A, int8_t *B, float *C, int32_t M, int32_t N, int32_t K,
    int16_t pos1, int16_t pos2, float scale1, float scale2,float &return_time) {
  struct timeval start;
  struct timeval end;
  float time_use;
  int N_align = N;
  cnrtRet_t ret;
  gettimeofday(&start, NULL);

  cnrtQueue_t pQueue;
  CNRT_CHECK(cnrtCreateQueue(&pQueue));

  cnrtDim3_t dim;
  cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;    // CNRT_FUNC_TYPE_BLOCK=1
  dim.x = 1;
  dim.y = 1;
  dim.z = 1;

  if (MP1) {
    dim.x = 1;
    func_type = CNRT_FUNC_TYPE_BLOCK;
  } else if (MP4) {
    dim.x = 4;
    func_type = CNRT_FUNC_TYPE_UNION1;
    //printf("UNION1!\n");
  } else if (MP8) {
    dim.x = 8;
    func_type = CNRT_FUNC_TYPE_UNION2;
  } else if (MP16) {
    dim.x = 16;
    func_type = CNRT_FUNC_TYPE_UNION4;
    //printf("16\n");
  } else if (MP32) {
    dim.x = 32;
    func_type = CNRT_FUNC_TYPE_UNION8;
  } else {
    printf("MP select is wrong! val = %d, use default setting,mp=1\n",
           MP_SELECT);
     return -1;
  }

  gettimeofday(&end, NULL);
  time_use =
      ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
```

```
64          1000.0;
65      //printf("cnrt init time use %f ms\n", time_use);
66
67      float *h_f32b = (float *)malloc(K * sizeof(float));
68      half *h_c = (half *)malloc(M * N_align * sizeof(half));
69
70      half *d_c = NULL;
71      int8_t *d_a = NULL;
72      int8_t *d_w = NULL;
73      int16_t pos = pos1 + pos2;
74
75      gettimeofday(&start, NULL);
76
77  // 分配空间
78      CNRT_CHECK(cnrtMalloc((void **)&d_c, sizeof(half) * M * N_align));
79      CNRT_CHECK(cnrtMalloc((void **)&d_a, sizeof(int8_t) * M * K));
80      CNRT_CHECK(cnrtMalloc((void **)&d_w, sizeof(int8_t) * K * N_align));
81
82  // 将矩阵A和B的内容赋值给新分配的空间
83      CNRT_CHECK(cnrtMemcpy(d_a, A, sizeof(int8_t) * M * K, CNRT_MEM_TRANS_DIR_HOST2DEV));
84      CNRT_CHECK(cnrtMemcpy(d_w, B, sizeof(int8_t) * K * N_align,CNRT_MEM_TRANS_DIR_HOST2DEV));
85
86      gettimeofday(&end, NULL);
87      time_use =
88          ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
89          1000.0;
90      //printf("malloc &copyin time use %f ms\n", time_use);
91
92      cnrtKernelParamsBuffer_t params;
93      CNRT_CHECK(cnrtGetKernelParamsBuffer(&params));      // Gets a parameter buffer for
            cnrtInvokeKernel_V2 or cnrtInvokeKernel_V3.
94      CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &d_c, sizeof(half *)));    // Adds a parameter to a
            specific parameter buffer.
95      CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &d_a, sizeof(int8_t *)));
96      CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &d_w, sizeof(int8_t *)));
97      CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &M, sizeof(uint32_t)));
98      CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &K, sizeof(uint32_t)));
99      CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &N_align, sizeof(uint32_t)));
100     CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &pos, sizeof(uint16_t)));
101
102     cnrtKernelInitParam_t init_param;
103     CNRT_CHECK(cnrtCreateKernelInitParam(&init_param));
104     CNRT_CHECK(cnrtInitKernelMemory((const void *)gemm16Kernel, init_param));
105
106     cnrtNotifier_t notifier_start;    // A pointer which points to the struct describing notifier.
107     cnrtNotifier_t notifier_end;
108     CNRT_CHECK(cnrtCreateNotifier(&notifier_start));
109     CNRT_CHECK(cnrtCreateNotifier(&notifier_end));
110     float timeTotal = 0.0;
111
112     //printf("start invoke  : \n");
113     gettimeofday(&start, NULL);
114
115     CNRT_CHECK(cnrtPlaceNotifier(notifier_start, pQueue));    // Places a notifier in specified queue
116     CNRT_CHECK(
117         cnrtInvokeKernel_V3((void *)&gemm16Kernel, init_param, dim, params, func_type, pQueue, NULL));
            // Invokes a kernel written in Bang with given params on MLU
118     CNRT_CHECK(cnrtPlaceNotifier(notifier_end, pQueue));      // Places a notifier in specified queue
119
120     CNRT_CHECK(cnrtSyncQueue(pQueue));    // Function should be blocked until all precedent tasks in the
            queue are completed.  同步Queue
121     gettimeofday(&end, NULL);
```

```
122   time_use =
123       ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
124       1000.0;
125 //   //printf("invoke  time use %f ms\n", time_use);
126 //   cnrtNotifierElapsedTime(notifier_start, notifier_end, &timeTotal);
127 // get the duration time between notifer_start and notifer_end.
128   // cnrtNotifierDuration(notifier_start, notifier_end, &timeTotal);     // Gets duration time of two
      makers
129   CNRT_CHECK(cnrtNotifierDuration(notifier_start, notifier_end, &timeTotal));
130   return_time = timeTotal / 1000.0;
131   //printf("hardware total Time: %.3f ms\n", return_time);
132   gettimeofday(&start, NULL);
133
134   CNRT_CHECK(cnrtMemcpy(h_c, d_c, sizeof(half) * M * N_align,
135                         CNRT_MEM_TRANS_DIR_DEV2HOST));
136   for (int j = 0; j < M; j++) {
137     for (int i = 0; i < N; i++) {
138       CNRT_CHECK(cnrtConvertHalfToFloat(&C[j * N + i], h_c[j * N_align + i]));
139       C[j * N + i] = C[j * N + i]/(scale1 * scale2);
140     }
141   }
142   gettimeofday(&end, NULL);
143   time_use =
144       ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
145       1000.0;
146   //printf("copyout &convert time use %f ms\n", time_use);
147
148 // free
149   CNRT_CHECK(cnrtFree(d_c));
150   CNRT_CHECK(cnrtFree(d_a));
151   CNRT_CHECK(cnrtFree(d_w));
152
153   CNRT_CHECK(cnrtDestroyQueue(pQueue));
154   CNRT_CHECK(cnrtDestroyKernelParamsBuffer(params));
155   CNRT_CHECK(cnrtDestroyNotifier(&notifier_start));
156   CNRT_CHECK(cnrtDestroyNotifier(&notifier_end));
157   free(h_f32b);
158   free(h_c);
159   //free(h_w);
160   //free(h_w_reshape);
161   return 0;
162 }
```

在优化过程中，Host 端的代码基本不变，我们重点关注 Device 端代码的优化过程。

1.2.5.2 标量操作实现

该步骤直接在 NRAM 上使用循环和标量操作进行计算。每个 DLP 计算核都有自己的 NRAM，和 GDRAM 相比，虽然空间较小，但是具有更高的读写带宽和更低的访问延迟。因此，该步骤中将输入的左右矩阵全部从 GDRAM 拷入 NRAM 中，在 NRAM 中进行计算，然后再拷回 GDRAM。为了方便读者理解，在这个例子中我们假设输入的左右矩阵规模都为 256*256，以保证输入矩阵可以一次性拷入 NRAM。一旦输入矩阵规模超过 NRAM 的空间大小时，则需要对 NRAM 复用进行多次拷入和拷出。如1.20代码示例所示，该程序中无须对输入矩阵做任何处理，直接使用矩阵乘公式进行计算。由于并没有利用到 DLP 硬件架构的优势，计算时间较长。该代码命名为 gemm_GDRAM.mlu。

代码示例 1.20　标量操作实现矩阵乘法的示例代码

```
1
2  //filename: gemm_GDRAM.mlu
3  #include "mlu.h"
4  __mlu_entry__ void gemm16Kernel(half *outputDDR, half *input1DDR, half *input2DDR,
5                                  uint32_t m, uint32_t k, uint32_t n) {
6      half ret;
7      __nram__ half input1NRAM[256*256];
8      __nram__ half input2NRAM[256*256];
9      __nram__ half outputNRAM[256*256];
10     __memcpy(input1NRAM, input1DDR, m * k * sizeof(half), GDRAM2NRAM);   //从 GDRAM拷入NRAM
11     __memcpy(input2NRAM, input2DDR, k * n * sizeof(half), GDRAM2NRAM);
12
13     for (uint32_t i = 0; i < m; i++) {
14         for (uint32_t j = 0; j < n; j++) {
15             ret = 0;
16             for (uint32_t t = 0; t < k; t++) {
17                 ret += input1NRAM[i*k+t] * input2NRAM[t*n+j];
18             }
19             outputNRAM[i*n+j] = ret;
20         }
21     }
22     __memcpy(outputDDR, outputNRAM, m * n * sizeof(half), NRAM2GDRAM);   //将计算结果拷回GDRAM
23 }
```

1.2.5.3 张量操作实现

在上一步基础上，该步骤中使用 BCL 提供的向量计算指令完成矩阵乘计算。向量计算指令可以更好地发挥 DLP 硬件性能优势，提升计算效率。

这里先介绍后续步骤中要解决的矩阵乘法规模。为方便读者理解，假设左矩阵规模大小为 256*256，右矩阵规模大小为 256*N（由于指令计算的对齐要求，N 必须可被 256 整除，如 327680）。如图 1.10所示：

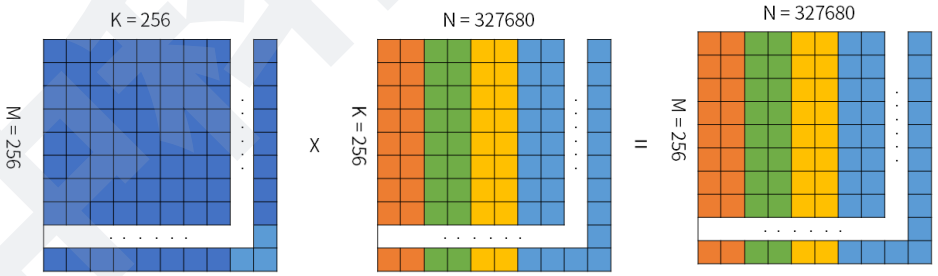

图 1.10　待解决的矩阵乘法规模

针对该问题规模，可以将输入左矩阵一次性拷入 NRAM。针对 DLP 架构特点，在执行卷积指令操作时，需要将输入的右矩阵拷入 WRAM 中，并且在向 WRAM 拷入前需要对数据进行量化处理并摆放成特定的数据格式，然后再使用 __bang_conv 指令进行计算。由于右矩阵规模较大，在代码中需将右矩阵分批次拷入 WRAM 进行计算。

Device 端关键代码如代码示例1.21所示（文件名为 gemm_CONV.mlu）。其中，all_round 表示计算的循环次数，和右矩阵规模大小相关; dst_stride 和 src_stride 代表调整右矩阵数据

摆放格式过程中的步长；total_times 表示调整右矩阵数据格式需要的次数，因为实验使用的 DLP 上有 64 个卷积计算单元，需要将原本顺序摆放的数据按照 64 个为一组间隔摆放。

代码示例 1.21　张量操作实现矩阵乘法的示例代码

```

//filename: gemm_CONV.mlu
#include "mlu.h"
#define ROUND 256
__mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
                                uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
    __nram__ int8_t input1NRAM[256*256];
    __nram__ int8_t input2NRAM[256*256];
    __nram__ int8_t input2NRAM_tmp[256*256];
    __wram__ int8_t input2WRAM[256*256];
    __nram__ half outputNRAM[256*256];
    __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
                            //在这里将左矩阵一次性拷入NRAM

    int all_round = n / ROUND;
    int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
    int32_t src_stride = k * sizeof(int8_t);
    int32_t size = k * sizeof(int8_t);
    int32_t total_times = ROUND / 64;
    //__bang_printf("taskDim=%d,clusterId=%d,coreId=%d\n",taskDim,clusterId,coreId);
    for(int i = 0; i < all_round; i++) {
        __memcpy(_____________________________________);
        for (int j = 0; j < total_times; j++) {      //这里将数据摆放成bang_conv可以使用的格式
            __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
                                    size, NRAM2NRAM, dst_stride, src_stride, 64);
        }
        __memcpy(_____________________________________);
        __bang_conv(outputNRAM, input1NRAM, input2WRAM, k, m, 1, 1, 1, 1, 1, ROUND, pos);
        for (int j = 0; j < m; j++) {    //要对每轮计算的结果进行拼接
            __memcpy(_____________________________________);
        }
    }
}
```

1.2.5.4　多核并行实现

上述步骤中只调用了 DLP 的一个计算核进行计算，考虑到所使用的 DLP 有 16 个计算核，可以进一步采用 16 个计算核进行并行运算。其基本思想是根据输入矩阵规模的大小，将输入矩阵拆分成多份并分配给不同的计算核进行计算，最后再对计算结果进行合并。通过将原本由 1 个计算核承担的计算任务分配给 16 个核，大大提升了计算速度。

在这里我们对 n 进行分块并行计算。分块单位为长度 256 的数据块。每个计算核每次从 GDRAM 上拷贝数据的时候都根据自己的 CoreId 来确定目标数据的内存地址，并且只将自己负责的数据块拷入 NRAM。Device 端关键代码如代码示例1.22所示（文件名为 gemm_PARALL.mlu）。

代码示例 1.22　多核并行实现矩阵乘法的示例代码

```

//filename: gemm_PARALL.mlu
#include "mlu.h"
#define ROUND 256
```

```
__mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
    uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
    __nram__ int8_t input1NRAM[256*256];
    __nram__ int8_t input2NRAM[256*256];
    __nram__ int8_t input2NRAM_tmp[256*256];
    __wram__ int8_t input2WRAM[256*256];
    __nram__ half outputNRAM[256*256];
    __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
                            //在这里将左矩阵一次性拷入NRAM

    int all_round = n / ( taskDim * ROUND);    //因为现在使用16个核同时运算，所以每个核循环的次数也相应
     减少
    int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
    int32_t src_stride = k * sizeof(int8_t);
    int32_t size = k * sizeof(int8_t);
    int32_t total_times = ROUND / 64;

    //__bang_printf("taskDim=%d,taskId=%d\n",taskDim, taskId);
    for(int i = 0; i < all_round; i++) {
        __memcpy(_____________________________________);    //只涉及这个核需要的数据

        for (int j = 0; j < total_times; j++) {
            __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
                        size, NRAM2NRAM, dst_stride, src_stride, 64);
        }
        __memcpy(_____________________________________);
        __bang_conv(_____________________________________);
        for (int j = 0; j < m; j++) {                                    //向GDRAM回写的时候也要注意每个
     核的位置不同
            __memcpy(_____________________________________);
        }
    }
}
```

1.2.5.5 SRAM 的使用

在第上一步中，因为使用了 4 个 Cluster 的 16 个计算核进行并行计算，而相同 Cluster 上的 4 个计算核在从 GDRAM 上拷贝数据到各自的 NRAM/WRAM 时，会争抢该 Cluster 到 GDRAM 的带宽，导致数据读取速度降低。考虑搭配每个 Cluster 有一个共享的 SRAM，我们将数据先从 GDRAM 拷贝到 SRAM，再从 SRAM 分发到 NRAM/WRAM 中，避免了带宽竞争问题，提高了数据读取速度。

特别注意的是，因为从 GDRAM 拷入数据到 SRAM 和从 SRAM 拷入数据到 NRAM 这两个操作是在两种不同功能的核上执行（即普通计算核和 Memory 核），所以这两个操作可以并行执行。为了保证数据一致性，需要在数据从 GDRAM 拷入到 SRAM 之后，从 SRAM 拷入到 NRAM 之前设置同步操作，即 BCL 内置的 __sync_cluster() 函数。其原理如 1.11图所示。

整个执行过程如下图所示:

Device 端关键代码如代码示例1.23所示（文件名为 gemm_SRAM.mlu）。其中 clusterId 表示此时执行任务的是哪个 Cluster，范围为 [0,3]。和原本的多核并行程序相比较，带有 SRAM 的版本打破了原来各个计算核之间在时序上的独立性。由此特别需要注意在每次将数据从 GDRAM 拷贝到 SRAM 和从 SRAM 拷贝到 NRAM 都需要执行同步来保证时序上的数据一

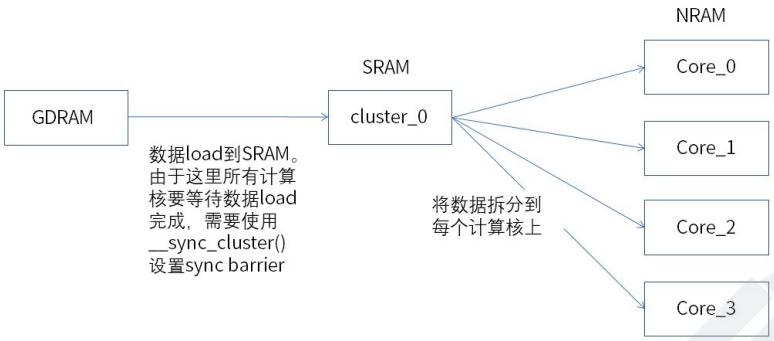

图 1.11　SRAM 的数据同步

致性。其整个执行过程中的数据调度时序如图1.12所示。

代码示例 1.23　使用 SRAM 的示例代码

```

//filename: gemm_SRAM.mlu
#include "mlu.h"
#define ROUND 256
__mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
    uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
    __nram__ int8_t input1NRAM[256*256];
    __nram__ int8_t input2NRAM[256*256];
    __nram__ int8_t input2NRAM_tmp[256*256];
    __wram__ int8_t input2WRAM[256*256];
    __nram__ half outputNRAM[256*256];
    __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
                        //在这里将左矩阵一次性拷入NRAM
    int all_round = n / ( taskDim * ROUND);    //因为现在使用16个核同时运算，所以每个核循环的次数也相应
     减少
    int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
    int32_t src_stride = k * sizeof(int8_t);
    int32_t size = k * sizeof(int8_t);
    int32_t total_times = ROUND / 64;
    __mlu_shared__ int8_t input2SRAM[256*1024];
    //_bang_printf("taskDim=%d,clusterId=%d,coreId=%d\n",taskDim,clusterId,coreId);
    for(int i = 0; i < all_round; i++)
    {
        // copy GDRAM2SRAM
        __memcpy(_____________________________);        //  只将右矩阵拷入SRAM中
        __sync_cluster();    //设置sync barrier
        // copy SRAM2NRAM
        __memcpy(_____________________________);

        // 将数据摆好对应的格式
        for (int j = 0; j < total_times; j++) {
            __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
                                    size, NRAM2NRAM, dst_stride, src_stride, 64);
        }
        // copy NRAM2WRAM
        __memcpy(_____________________________);
        // compute
        __bang_conv(_________________________);
        // copy NRAM2GDRAM
```

```
39          for (int j = 0; j < m; j++) {                    //向GDRAM回写的时候也要注意每个
        核的位置不同
40              __memcpy(_____________________);
41          }
42          __sync_cluster();    //设置sync barrier
43      }
44  }
```

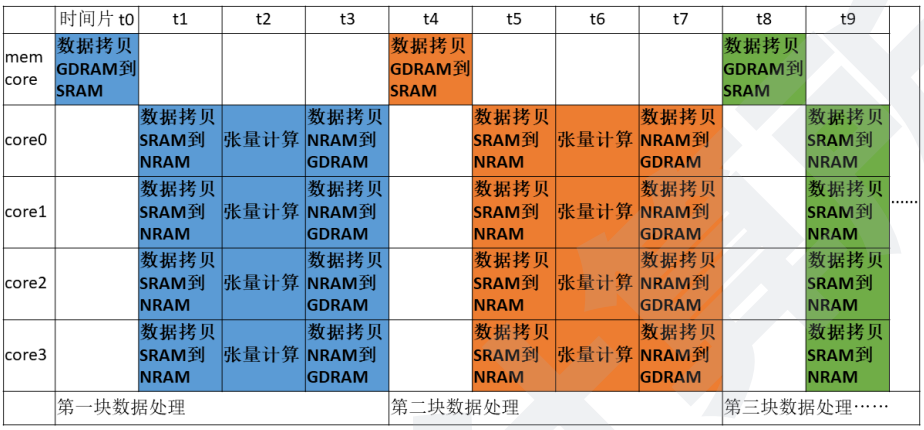

图 1.12　SRAM 的数据调度时序

1.2.5.6　访存与计算流水

由于所使用的 DLP 上有专门用于片上总线和 SRAM 管理的 SRAM 核，可以在前面步骤的基础上进一步实现访存与计算的流水。具体而言，针对 4 个并行执行的 Cluster，希望每个 Cluster 中的 SRAM 核和其他 4 个计算核构成流水线的计算模式。其中 SRAM 核只负责将数据从 GDRAM 拷入 SRAM，其余计算核则负责从 SRAM 拷入数据、完成矩阵乘法计算、将数据拷回 GDRAM。

Device 端关键代码如代码示例1.24所示（文件名为 gemm_PIPELINE.mlu）。其中，我们设置了在 SRAM 上的两个变量 input2SRAM1，input2SRAM2。初始时，SRAM 核从 GDRAM 上拷入数据到 input2SRAM1，当数据拷入完成后，4 个计算核开始工作，它们将自己需要的数据从 input2SRAM1 拷入进行计算。在计算核工作的同时，SRAM 核不会停止工作，它会将下一次需要计算的数据从 GDRAM 拷入 input2SRAM2，供给 4 个计算核在下一次使用，减少了计算核下一次的等待时间，input2SRAM1 和 input2SRAM2 交替读写重复上述过程直至所有数据计算完成。具体的数据调度时序如图 访存与计算流水的数据调度时序所示，从中可以发现耗时很长的从 GDRAM 到 SRAM 的数据拷贝被“隐藏”了。和之前步骤相比在相同的时间内，每次搬运了更多的 GDRAM 数据到片上并完成了计算。这也是计算机系统优化时常用到的数据流控制技巧，即“乒乓操作”。

代码示例 1.24　访存与计算流水的示例代码

```
1
2  //filename: gemm_PIPELINE.mlu
3  #include "mlu.h"
4  #define ROUND 256
5  __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
6      uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
7      __nram__ int8_t input1NRAM[256*256];
8      __nram__ int8_t input2NRAM[256*256];
9      __nram__ int8_t input2NRAM_tmp[256*256];
10     __wram__ int8_t input2WRAM[256*256];
11     __nram__ half outputNRAM[256*256];
12     __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
13                                 //在这里将左矩阵一次性拷入NRAM
14     int all_round = n / ( taskDim * ROUND);    //因为现在使用16个核同时运算，所以每个核循环的次数也相应
       减少
15     int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
16     int32_t src_stride = k * sizeof(int8_t);
17     int32_t size = k * sizeof(int8_t);
18     int32_t total_times = ROUND / 64;
19     __mlu_shared__ int8_t input2SRAM1[256*1024];
20     __mlu_shared__ int8_t input2SRAM2[256*1024];
21     __mlu_shared__ int8_t * input2SRAM_read;
22     __mlu_shared__ int8_t * input2SRAM_write;
23     input2SRAM_write=input2SRAM1;
24     // copy GDRAM2SRAM
25     __memcpy(input2SRAM_write, input2DDR + ROUND * (clusterId * 4) * k,
26              k * ROUND * 4 * sizeof(int8_t), GDRAM2SRAM);       //  只将右矩阵拷入SRAM中
27     __sync_cluster();    //设置sync barrier
28     //_bang_printf("taskDim=%d,clusterId=%d,coreId=%d\n",taskDim,clusterId,coreId);
29     for(int i = 0; i < all_round-1; i++)
30     {
31         if (i % 2 == 0)
32         {
33             input2SRAM_read=input2SRAM1;
34             input2SRAM_write=input2SRAM2;
35         } else
36         {
37             input2SRAM_read=input2SRAM2;
38             input2SRAM_write=input2SRAM1;
39         }
40         // copy GDRAM2SRAM
41         __memcpy(_____________________________________);       //  只将右矩阵拷入SRAM中
42         // copy SRAM2NRAM
43         __memcpy(_____________________________________);
44
45         // 将数据摆好对应的格式
46         for (int j = 0; j < total_times; j++) {
47             __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
48                                     size, NRAM2NRAM, dst_stride, src_stride, 64);
49         }
50         // copy NRAM2WRAM
51         __memcpy(input2WRAM, input2NRAM, ROUND*k*sizeof(int8_t), NRAM2WRAM);
52         // compute
53         __bang_conv(_________________________________);
54         // copy NRAM2GDRAM
55         for (int j = 0; j < m; j++) {                                    //向GDRAM回写的时候也要注意每个
       核的位置不同
56             __memcpy(_________________________________);
57         }
58         __sync_cluster();    //设置sync barrier
59     }
60     __memcpy(_____________________________________);
```

```
61
62      // 将数据摆好对应的格式
63      for (int j = 0; j < total_times; j++) {
64          __memcpy(_____________________________________);
65      }
66      // copy NRAM2WRAM
67      __memcpy(_____________________________________);
68      // compute
69      __bang_conv(_____________________________________);
70      // copy NRAM2GDRAM
71      for (int j = 0; j < m; j++) {                          //向GDRAM回写的时候也要注意每个核的
         位置不同
72          __memcpy(_____________________________________);
73      }
74  }
```

	时间片 t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	
mem core	数据拷贝 GDRAM到 SRAM(S1)	数据拷贝 GDRAM到 SRAM(S2)			数据拷贝 GDRAM到 SRAM(S1)			数据拷贝 GDRAM到 SRAM(S2)			
core0		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	
core1		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	……
core2		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	
core3		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRA	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	
	第一块数据处理				第二块数据处理			第三块数据处理……			

图 1.13　访存与计算流水的数据调度时序

1.2.6　实验评估

本实验主要考虑用智能编程语言解决特定问题，实验评估指标主要为程序本身对问题规模的支持范围和性能。本实验设定的评估标准如下：

- 60 分标准：在规模 m=256,k=256,n=327680 下 DLP 计算结果与 CPU 计算结果误差在 10% 以内。
- 80 分标准：在规模 m=256,k=256,n=327680 下 DLP 计算结果与 CPU 计算结果误差在 1% 以内。
- 90 分标准：在规模 m=256,k=256,n=327680 下 DLP 计算结果与 CPU 计算结果误差在 0.1% 以内，耗时小于 20ms。
- 100 分标准：在规模 m=256,k=256,n=327680 下 DLP 计算结果与 CPU 计算结果误差在 0.06% 以内，耗时小于 15ms。

1.2.7 实验思考

1. __bang_conv 指令对数据摆放有特殊要求的原因是什么?
2. 在 SRAM 的使用代码 1.23中，其中有两个 sync，这是为什么?
3. 以上程序目前的瓶颈在哪里？有什么方法可以验证?
4. 访存与计算流水的设计是否还有其他方案？是否有可能从整个软硬件系统的角度进一步提升性能?
5. 如果输入数据长度不是 256 的整数倍，程序该怎么修改以适应这种变化?

1.3 智能编程语言算子开发实验（BPL 开发实验）

1.3.1 实验目的

本实验通过智能编程语言的 python 开发接口 BPL 来实现 PowerDifference 算子，对算子进行单算子正确性验证，扩展高性能库算子，并最终集成到 TensorFlow 框架。通过本实验读者可以掌握使用 BPL 开发智能编程语言算子的方法，简化算子开发流程，增加开发效率。

实验工作量：代码量约 150 行，实验时间约 10 小时。

1.3.2 背景介绍

本节重点介绍智能编程语言的 python 开发接口 BPL，包括 BPL 的简单介绍，编程模型及调试方法等。

1.3.2.1 BPL 简介

在 **??**中，我们介绍了智能编程语言 BANG C Language（BCL），它是一种类 C++ 的编程语言，学习成本较高。为了降低用户学习成本，方便用户快速上手智能编程语言开发，我们在 BCL 的基础上设计了基于 Python 的智能编程语言 BANG Python Language（BPL），用于神经网络算子开发与网络搭建，为 DLP 硬件上的算子开发提供便捷的软件接口，相比于 BCL 的 C++ 编程模式更加快速简单。

BPL 具有面向向量编程，代码可读性高；输出形式广泛，便于二次开发；编程架构清晰，易于用户上手；具有运行时支持，方便算子开发验证等多个优点。其提供了一系列的 Python API 接口，方便用户操作 DLP 硬件的运行。在这些接口中添加了大量的数据类型与数据对齐检查，为用户在编程阶段提供编程指导与建议，帮助用户快速写出可执行代码。相比于 BCL，用户在使用 BPL 时不需要去考虑语言的语法特性，只需要专注于内存的操作与计算信息的描述，开发难度极大降低。同时，BPL 特有的运行时（runtime）模块，可以自动生成 DLP 硬件的 Host 端代码，用户可以快速的进行算子开发调试。与 BCL 相比，BPL 的平均开发代码量减少了约 40%，用户的开发效率可以得到极大提高。

BPL 的软件栈结构如图 1.14所示，开发者使用 BPL 接口描述的算子实现，会生成 BPL 的 IR（intermediate representation），BPL IR 经过 BPL 编译优化之后，可以生成对应的 DLP 端 BCL 代码（.mlu 为后缀的文件）、CPU 端动态链接库文件（.so 为后缀的文件）、可执行

module（Python 文件中的变量）。BPL 生成得到的 BCL 代码可以根据用户具体需求进行二次开发，并集成到 BCL 算子库（CNPlugin）中，同时 BPL 生成得到的可执行 module 可以结合 BPL 的运行时支持直接运行，用来验证算子的正确性。

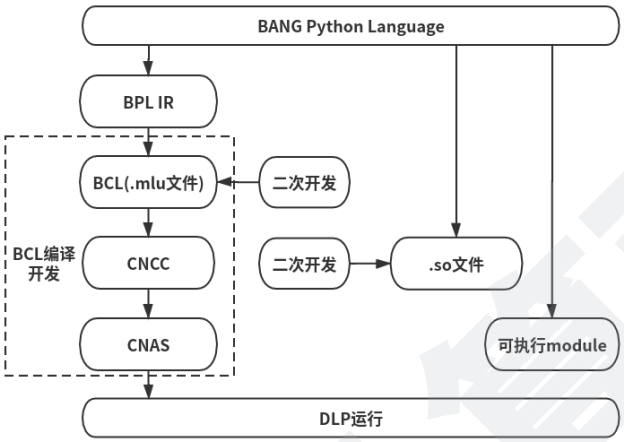

图 1.14　BPL 软件栈示意图

1.3.2.2　BPL 编程模型

BPL 是面向向量编程的，其通过提供一系列的 API 用于算子开发，这些 API 被称为 Tensor Compute Primitive（张量计算原语），简称 TCP。

张量是神经网络中的基本数据单元，在数学上表示为多维数组，TCP 扩展了张量的概念，将一维数组（向量）、二维数组（矩阵）也称为张量。在 TCP 中，一个张量由它的形状 (shape)、数值类型 (dtype)、名字 (name) 和地址空间 (scope) 确定。可以用以下方式定义：

```
tensor=TCP.Tensor(shape, dtype, name, scope)
```

除了张量外，标量也同样是神经网络的一种数据单元，在 TCP 中，标量是神经网络的一种数据单元，在数学上表示为一个单独的数值。它对应的是存储在芯片寄存器里的数值。在 TCP 中，一个标量由它的数据类型 (dtype)，名称 (name) 和初始值 (value) 确定，可以用以下方式定义：

```
scalar=TCP.Scalar(dtyp, name, value)
```

使用 TCP 开发算子的大致流程如图 1.15所示，用户需要首先创建 TCP 容器，接着在 TCP 容器中使用 TCP 接口描述计算，主要包括定义张量、张量搬运、张量计算、并行编程以及条件和循环控制等。计算描述完成后，用户通过编译接口生成可执行 module，该可执行 module 可以保存成 CPU 端的动态库文件和 DLP 端的 mlu 文件。用户可以选择将输入、初始化的输出数据传入可执行 module 中运行，先行检查算子的正确性，并根据运行结果进行调试分析。在调试分析完成后，可以将 DLP 端的 mlu 文件集成到 BCL 算子库（CNPlugin）中。代码 1.25展示了一个 TCP 算子开发的简要流程代码。

代码示例 1.25　TCP 算子开发流程示例

```
//filename: tcp_sample.py
    # 获取numpy的输入输出数据
data_in0 = np.random.uniform(low=-10, high=10, size=SHAPE)
data_in1 = np.random.uniform(low=-10, high=10, size=SHAPE)
data_out = data_in0 + data_in1

length = np.prod(data_in0.shape)

# 创建TCP容器
bp = tcp.TCP()

# 计算分块次数
assert length % task_num == 0
core_wl = 3 * length // task_num
loop_num = np.ceil(core_wl * dtype_sz / NRAM_SIZE)
core_wl //= 3
while core_wl % loop_num != 0:
loop_num += 1
loop_wl = int(core_wl // loop_num)

# 定义输入输出张量
tensor_in0 = bp.Tensor(shape=data_in0.shape, name="INPUT0", dtype=dtype, scope="global")
tensor_in1 = bp.Tensor(shape=data_in1.shape, name="INPUT1", dtype=dtype, scope="global")
tensor_out = bp.Tensor(shape=data_out.shape, name="OUTPUT", dtype=dtype, scope="global")

# 描述多核并行逻辑和分块计算逻辑
task_type = TaskType.get_task_type(task_num)
with bp.for_range(0, task_num, task_num=task_num, task_type=task_type) as task_id:

# 定义NRAM上的中间张量
tensor_in0_n = bp.Tensor(shape=(loop_wl,), name="INPUT0_N", dtype=dtype, scope="nram")
tensor_in1_n = bp.Tensor(shape=(loop_wl,), name="INPUT1_N", dtype=dtype, scope="nram")
tensor_out_n = bp.Tensor(shape=(loop_wl,), name="OUTPUT_N", dtype=dtype, scope="nram")

# 描述分块逻辑
with bp.for_range(0, loop_num) as i:
start = task_id * core_wl + i * loop_wl
stop = start + loop_wl
# 调用TCP接口进行张量搬运与计算
bp.memcpy(tensor_in0_n, tensor_in0[start:stop])
bp.memcpy(tensor_in1_n, tensor_in1[start:stop])
bp.add(tensor_out_n, tensor_in0_n, tensor_in1_n)
bp.memcpy(tensor_out[start:stop], tensor_out_n)


# 编译生成可执行module
fvec_add = bp.BuildBANG(inputs=[tensor_in0, tensor_in1], outputs=[tensor_out], kernel_name="fvec_add")

# 设备端空间申请与张量搬运
ctx = BPL.context(0)
data_in0_dev = BPL.Array(data_in0.astype(dtype), ctx)
data_in1_dev = BPL.Array(data_in1.astype(dtype), ctx)
data_out_dev = BPL.Array(np.zeros(data_out.shape, dtype), ctx)

# 准备一个空目录dirname，保存mlu代码和.so文件
fvec_add.save(dirname)

# 直接启用设备运行，验证结果
fvec_add(data_in0_dev, data_in1_dev, data_out_dev)
BPL.assert_allclose(data_out_dev.asnumpy(), data_out.astype(dtype), rtol=1e-2, atol=1e-2)

```

```
# 载入module，启用设备运行，验证结果
fadd = load_mod(dirname)
fadd(data_in0_dev, data_in1_dev, data_out_dev)
BPL.assert_allclose(data_out_dev.asnumpy(), data_out.astype(dtype), rtol=1e-2, atol=1e-2)
```

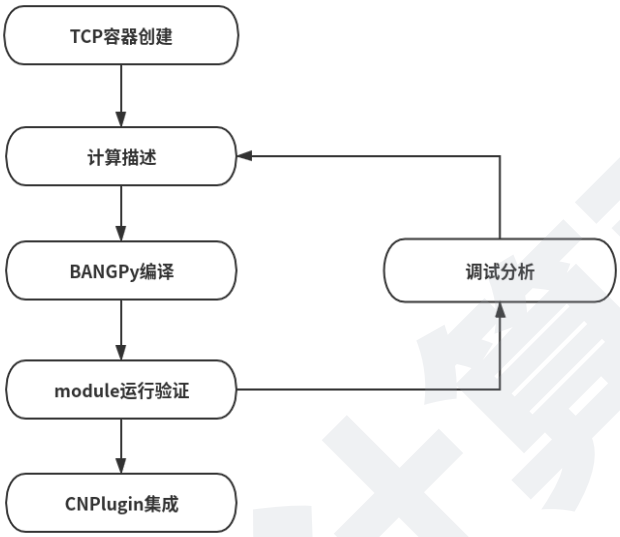

图 1.15　TCP 算子开发流程示意图

1.3.2.3　BPL 开发调试

BPL 提供了多种调试方法及接口来简便用户的学习与开发。在算子开发的计算描述阶段，用户可以直接使用 python 的 print 方法直接打印代码中的标量与张量，获取它们的具体信息以便进行调试：

```
Input:
    tensor = tcp.Tensor(shape=(256,), name="INPUT0", dtype=BPL.float16, scope="global")
    print(tensor)
Output:
    <BPL.Tensor 'INPUT0' shape=(256,) dtype=info(dtype=float16) scope=global buffer=buffer(INPUT0,
      0x18f3990)>
```

在编译阶段，BPL 有对计算原语的检查机制，会检查计算原语中使用的标量与向量是否符合计算原语的各种限制，包括对齐限制与类型限制等，对不满足限制的代码，会抛出报错信息，给出出错原因与修改意见：

```
Input:
    tensor_g = tcp.Tensor(shape=(255,), name="INPUT_G", dtype=BPL.float16,scope="global")
    tensor_n = tcp.Tensor(shape=(256,), name="INPUT_N", dtype=BPL.float16,scope="global")
    tcp.memcpy(tensor_n, tensor_g)
Output:
    ValueError: Error: src_data must have same length with dst_data! Current length is 256,
     dst_data length is 255.
```

使用 BPL 的运行时支持，用户可以直接运行 module 进行单算子运行正确性验证，同时用户可以使用 TCP 的 print 接口，插入 BCL 的 printf 接口到生成的 BCL 代码之中，以便于在

module 运行验证阶段输出打印 DLP 上的中间运算结果，进行调试验证。对于 module 运行验证的方法，请读者参见 BPL 用户手册的 6.1 节内容，这里不再赘述。

1.3.3 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。

1.3.4 实验内容

本节实验基于第 1.1节中的智能编程语言算子开发与集成实验，在前者基础上进一步把 PowerDifference 算子用智能编程语言的 python 接口 BPL 实现并进行单算子正确性验证，通过高性能库 PluginOp 接口扩展算子，并和高性能库原有算子一起集成到编程框架 TensorFlow 中，此后将风格迁移模型在扩展后的 TensorFlow 上运行，最后将其性能结果和第**??**节中的性能结果进行对比。实验内容和流程如图1.16所示，主要包括：

1. **BCL 算子实现与正确性验证：**采用智能编程语言 BCL 的 python 接口 BPL 实现 PowerDifference 算子并完成相应测试。首先，使用 BPL 的张量计算原语实现计算 Kernel，生成可执行 module 与 DLP 端 mlu 文件，并在 python 端执行可执行 module 验证算子正确性；

2. **框架算子集成：**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. **模型在线推理：**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. **模型离线推理：**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，以及第**??**节中的推理性能进行对比。

图1.6中虚线框的部分是需要同学补充完善的实验文件。每一步实验操作需要修改的具体对应文件内容请参考下一节“实验步骤”。

1.3.5 实验步骤

如前所述，本实验的详细步骤包括：算子实现、框架集成、在线推理和离线推理等。

1.3.5.1 用 BPL 实现 BCL 算子

本节实验的第一步是使用 BPL 实现 PowerDifference 算子。具体的算子公式请参考 **??**小节的内容。实验的主要内容需要完成 PowerDifference 算子的计算描述代码，包括定义张量、张量搬运、张量计算、条件和循环控制等。基于 BPL 的 PowerDifference (plugin_power_difference_kernel.py) 具体实现如代码示例 1.26所示。

BPL 的张量计算原语都是针对 NRAM 地址空间上的张量，使用这些原语之前需要使用数据拷贝原语 memcpy 将 GDRAM 上定义的张量拷贝到 NRAM 上，同时相关的张量计算

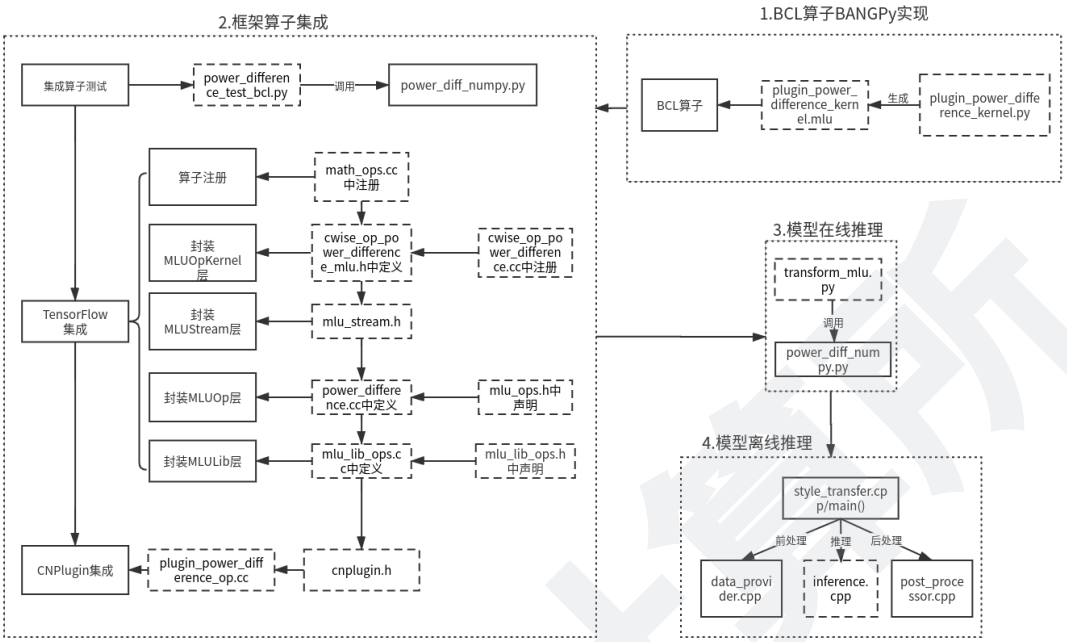

图 1.16　具体实验内容

原语也有一定的对齐约束，请读者参照 BPL 用户手册对张量操作进行对齐处理。

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。读者在完成实验时需注意该部分数据的处理逻辑。

代码示例 1.26　基于智能编程语言 BCL 的 Power Difference 实现

```
//filename: plugin_power_difference_kernel.py
//定义常量
SHAPE = 256

def   power_diff( ):
#TCP容器定义
bp = tcp.TCP()

#DLP内建变量与可变参数声明
task_id = bp.builtin_var("taskId")
len = bp.Var("len")
pow = bp.Var("pow")

#计算分片
quotient = ____________________________

#张量定义
input1 = bp.Tensor(shape=(___________), name="input1", dtype=dtype, scope="global")
input2 = bp.Tensor(shape=(___________), name="input2", dtype=dtype, scope="global")
```

```
output = bp.Tensor(shape=(____________), name="output", dtype=dtype, scope="global")
input1_nram = bp.Tensor(shape=(____________), name="input1_nram", dtype=dtype, scope="nram")
input2_nram = bp.Tensor(shape=(____________), name="input2_nram",dtype=dtype, scope="nram")

#条件与循环控制
with bp.for_range(0, quotient) as i:
#张量搬运
bp.memcpy(_______________________________)
........
#计算描述
........

#BPL编译
f = bp.BuildBANG(inputs=[input1, input2, len, pow], outputs=[output], kernel_name="
    PowerDifferenceKernel")
```

在计算描述完成后，可以直接运行生成的可执行 module 进行单算子正确性验证。BPL 的可执行 module 可以接收 BPL 的 Array 类型数据作为输入输出。读者可以使用 BPL 的 Array 接口将 numpy 的 ndarray 类型数据转换为 Array 类型。module 运行接收后，使用 Array 的 asnumpy 接口将输出数据转换为 numpy 的 ndarray 类型，然后与期望的正确结果进行误差比较。

```
#numpy变量定义
data_in0=np.random.uniform(low=-10, high=10, size=SHAPE)
data_in1=np.random.uniform(low=-10, high=10, size=SHAPE)
data_out=_______________________________

#Array类型转换
ctx=BPL.context(0)
data_in0_dev=BPL.Array(data_in0.astype(dtype), ctx)
data_in1_dev=BPL.Array(data_in1.astype(dtype), ctx)
data_out_dev=BPL.Array(np.zeros(data_out.shape, dtype), ctx)

#modue运行
f(_______________________________)

#误差比较
BPL.assert_allclose(data_out_dev.asnumpy(), data_out.astype(dtype),rtol=1e-2, atol=1e-2)
```

1.3.5.2 框架算子集成

该步骤内容与 1.1.5.2小节相同，请参考 1.1.5.2小节的实现步骤。

1.3.5.3 模型在线推理

该步骤内容与 1.1.5.3小节相同，请参考 1.1.5.3小节的实现步骤。

1.3.5.4 模型离线推理

该步骤内容与 1.1.5.4小节相同，请参考 1.1.5.4小节的实现步骤。

1.3.6 实验评估

本次实验中主要考虑基于 BPL 的算子实现与验证、与框架的集成以及完整的模型推理。模型推理的性能和精度在 1.1节已作为主要的参考指标，本实验的主要参考指标在于 BPL 的算子实现与验证。因此，本实验的评估标准设定如下：

- 60 分标准：使用 BPL 完成 PowerDifferenece 算子实现，正确生成 mlu 文件及可执行 module。
- 80 分标准：在 60 分基础上，使用 BPL 的运行时支持验证 PowerDifferenece 算子的正确性，精度误差在 1% 以内。
- 100 分标准：在 60 分基础上，完成 BCL 算子与 TensorFlow 框架的集成；使用 DLP 推理完整 pb 模型时，输出精度正常的风格迁移图片，输出正确的离线模型；完成离线推理程序的编写，执行离线推理时风格迁移，图片精度正常。

1.3.7 实验思考

1. 尝试使用 BPL 来完成 1.2节的实验，使用 BPL 是否会简化代码开发?
2. 使用 BPL 与使用原生 BCL 有何不同?
3. BPL 相比于原生 BCL 的优点与缺陷分别是什么?
4. BPL 最适合的使用场景有哪些?

参考文献