第 5 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如何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智能计算硬件指令，实现高效运行的目标。具体而言，第 5.1节则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，以满足实际应用对智能计算系统的可扩展和高性能需求。第 5.2节进一步介绍智能编程语言 BCL 的一些高级特性，在实现功能的基础上进一步优化性能充分发挥 DLP 的硬件潜力。

5.1　智能编程语言算子开发与集成实验

5.1.1　实验目的

本节实验通过使用智能编程语言进行算子开发，对高性能库算子进行扩展，并最终集成到编程框架中，掌握对高性能库及编程框架进行扩展的能力，使读者可以在 DLP 硬件上自由设计并优化满足特定应用场景的新算子，满足日新月异智能算法的实际需求。

具体而言，在前述章节中我们用 DLP 硬件的高性能库替换了风格迁移网络中大部分 CPU 算子，但仍有需要在 CPU 上执行的 PowerDifference 算子。由于 PowerDifference 算子在 CPU 上执行，导致该算子计算的前后存在 DLP 和 CPU 之间的数据交互，即 DLP 到 CPU 的数据拷出以及 CPU 到 DLP 的数据拷入，增加了额外的访存带宽和功耗，影响了整体的执行效率。因此，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。

5.1.2　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 VS-Code 集成开发环境等。

5.1.2.1　编译器（CNCC）

编译器用于将使用智能编程语言编写的程序编译成 DLP 底层指令。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 5.1所示，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 MLISA 汇编，然后 MLISA 汇编由 CNAS 汇编器生成 DLP 上运行的二进制机器码。

在使用 CNCC 编译 BCL 文件时，有多个编译选项供开发者使用，如下表 5.1所示。

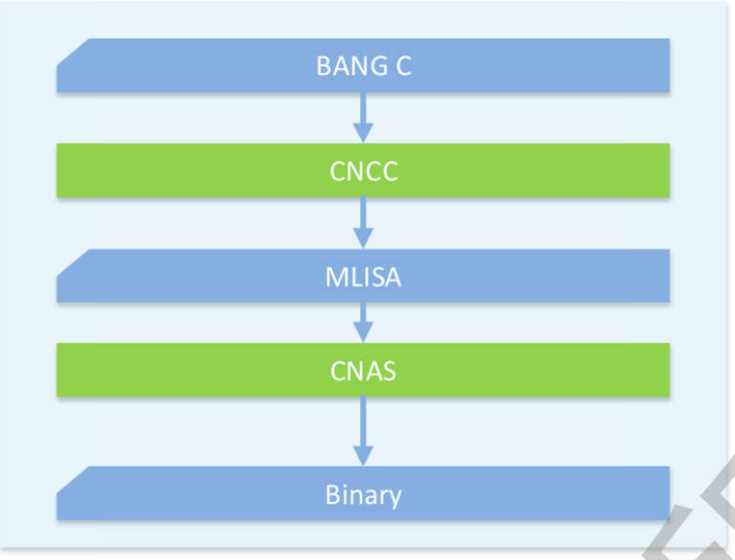

图 5.1　CNCC

表 5.1　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 MLISA 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，e.g.:bang 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 BANG 程序指定 DLP 的架构，e.g.: MLU270 等
–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

5.1.2.2　调试器（CNGDB）

调试器是面向智能编程语言程序的调试工具，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 面向异构编程模型调试的问题，提升了应用程序开发的效率。

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，例如:

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

这里以 BCL 写的快排程序为例, 演示如何使用 CNGDB 调试 BANGC 程序。图 5.2展示了 CNGDB 调试 recursion.mlu BCL 程序的基本流程，总的来说主要包含 6 个步骤：

- 在 xxx.mlu 文件 x 行插入断点:break xxx.mlu : x (break 可以缩写为 b)
- 执行程序:run(run 可以缩写为 r)

- 继续执行至下一个断点:continue(continue 可以缩写为 c)
- 删除断点 1:delete1(delete 可以缩写为 d)
- 打印变量 xx:printxx(print 可以缩写为 p)
- 打印内存:x/个数 + 字长 (x 是 examine 的缩写)

```
#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 5.2 CNGDB 调试示例

5.1.2.3 集成开发环境（CNStudio）

CNStudio 是一款针对于 BANG C 语言可在 Visual Studio Code 使用的编程插件，为了使 BANGC 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以下功能：

1. 语法高亮
2. 自动补全
3. 程序调试

当前 CNStudio 插件只提供离线安装包，不支持在线安装，安装包名称为：Software-Bangc-xxx.vsix 文件。vscode 版本要求 1.28.0 及以上版本，安装流程如下：通过上述三步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，若有“Cambricon BangC”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。

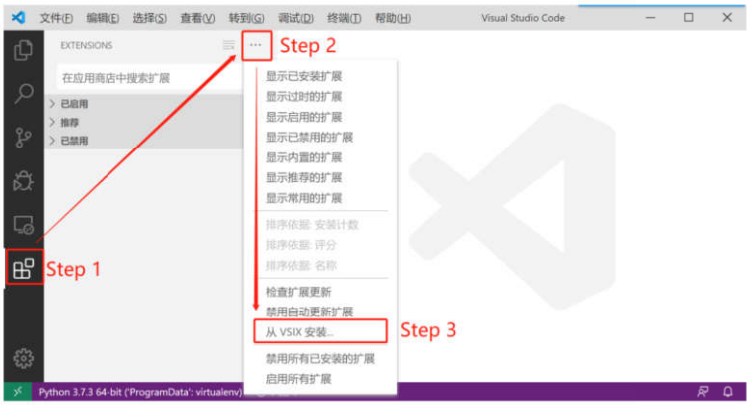

图 5.3　CNStudio 安装流程图

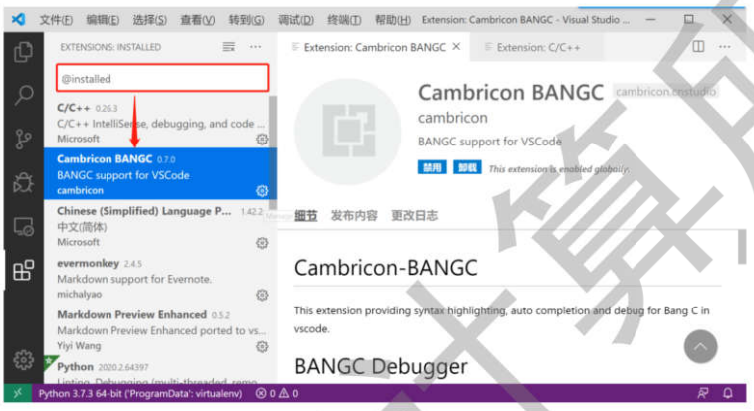

图 5.4　CNStudio 安装完成

在创建工程时（以新建一个 DLP 文件夹为例），每一个 project 都包含三种类型的文件，所以需要在 DLP 文件夹中新建 DLP 端程序所需的 dlp.mlu（BANG C 程序源　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　程保存为 workspace，方便下次直接打开工程文件。

5.1.3　实验环境

本章实验同样基于前述的 DLP 云平台环境，其已包含 CNCC、CNGDB、CNPerf 等基本工具。

5.1.4　实验内容

本节实验基于第 7.1 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 7.1 节中的性能结果进行对比。具体实验内容包括：

1. **算子实现：**采用智能编程语言 BCL 实现 PowerDifference 算子；

2. **算子测试：**对 PowerDifference 算子本身进行测试，保证其功能正确；

3. **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 编程框架中；

4. **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；

5. **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，并将其结果和在线推理，以及第 7.1 节中的推理性能进行对比。

实验工作量：6 学时

5.1.5 实验步骤

5.1.5.1 算子实现

基于智能编程语言的 PowerDifference 具体实现如图 5.5所示。

```
//定义常量
#define ONELINE 256

__mlu_entry__ void PowerDifferenceKernel(half* input1, half* input2, int pow, half*
    output, int len)
{
  int quotient = len / ONELINE;
  __bang_printf("%d %d\n", pow, len);
  int rem = len % ONELINE;

  //声明NRAM空间
  __nram__ half input1_nram[ONELINE];
  __nram__ half input2_nram[ONELINE];

  if ( rem != 0)
  {
    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(input1_nram, input1 + i * ONELINE, ONELINE * sizeof(half), GDRAM2NRAM);
    __memcpy(input2_nram, input2 + i * ONELINE, ONELINE * sizeof(half), GDRAM2NRAM);
    //按元素相加操作，将input1_nram和input2_nram的对应元素进行求和并储存在input1_nram
     中。
    __bang_sub(input1_nram, input1_nram, input2_nram, ONELINE);
    //NRAM中两个数据块的数据拷贝操作
    __memcpy(input2_nram, input1_nram + i * ONELINE, ONELINE * sizeof(half), NRAM2NRAM);
    for (int j = 0; j < pow - 1; j++)
    {
      //按元素相乘操作
      __bang_mul(input1_nram, input1_nram, input2_nram, ONELINE);
    }
    //内存拷贝：从NRAM中将计算的结果拷出至GDRAM中。
    __memcpy(output +i * ONELINE, input1_nram, ONELINE * sizeof(half), NRAM2GDRAM);
  }
}
```

图 5.5 基于智能编程语言 BCL 的 Power Difference 实现

5.1.5.2 算子测试

完成 BCL 算子的编写后，需要进一步编写相应测试程序验证代码实现的正确性。具体包括：编写运行时程序，编写 Main 程序和编译运行二进制等步骤。

1. **编写运行时程序（powerDiff.cpp）**

该程序通过运行时库 CNRT 的接口调用 BCL 算子的具体实现（PowerDifferenceKernel），对应的程序代码如图 5.6所示。其中，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 来完成算子函数的调用，最后完成计算后获取输出结果并销毁相应资源。

```
#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*));
    ...
    cnrtInvokeKernel_V2()
    //get output data
    cnrtMemcpy(...,CNRT_MEM_TRANS_DIR_DEV2Host)
    cnrtConvertHalfToFloatArray(...);
    //free data
    cnrtDestroy();
    ...
    return 0;
}
```

图 5.6　调用运行时库函数的程序示例代码

2. **编写 Main 程序（main.cpp）**

上述 MLUPowerDifferenceOp 函数需要通过 Main 程序进一步调用，Main 程序的具体

代码如图 5.7所示。其中，核心代码是调用 MLUPowerDifferenceOp 函数对 in_x.txt 和 in_y.txt 中的输入数据进行计算，并将结果输出到文件 out.txt 中，同时统计其计算时间，以及和 CPU 的运算结果相对比的错误率。

```
#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;
}
```

图 5.7　Main 程序示例代码

3. 编译运行二进制（power_diff_test）

完成上述代码的编写后，需要对程序进行编译测试。具体的编译命令如图 5.8所示。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。

```
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
```

图 5.8　测试程序的编译

5.1.5.3 框架集成

为了将前述已完成测试的 PowerDifference 算子集成至 TensorFlow 框架中，需要进行包括：PluginOp 接口封装，DLP 算子集成和算子测试等三方面工作。

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（如算子融合）的机制。因此，在完成 PowerDifference 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 5.9给出了 PluginOp 接口封装的部分示例代码，主要包括声明算子参数初始化函数 CreateOpParam、声明算子参数销毁函数 DestroyOpParam、声明算子构建接口 Create、声明单算子运行接口 Compute 四类函数的具体实现。

   - 声明算子参数初始化函数 CreateOpParam：在本接口中完成 static tensor 的创建、CPU 相关的赋值，再执行 cnmlBindConstData 绑定常量数据。
   - 声明算子参数销毁函数 DestroyOpParam：先销毁 statictensor 和影响 CPU 的指针，然后销毁为 cnmlPluginXXXOpParam 分配的地址空间。
   - 声明算子构建接口 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。
   - 声明单算子运行接口 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。

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：

   - MLUOpKernel：继承 TensorFlow 中的 OpKernel 类，作为与 TensorFlow 算子层的接口；
   - MLUStream：与 MLUOpKernel 类接口关联，负责 MLU 算子的实例化并管理算子缓存；
   - 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；
   - MLULib：对 CNML 和 CNRT 接口的直接封装，只包含极少的 TensorFlow 数据结构。

```
//Create and Destroy OpParam
cnmlStatus_t cnmlCreatePluginPowerDifferenceOpParam(...){}
cnmlStatus_t cnmlDestroyPluginPowerDifferenceOpParam(...){}
//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;
}
```

图 5.9　PluginOp 接口封装的示例代码

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/运行时库，因此在 TensorFlow 中集成 DLP 算子涉及上面各个层次。集成的整体流程如图 **??**所示，主要包括五个步骤：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。

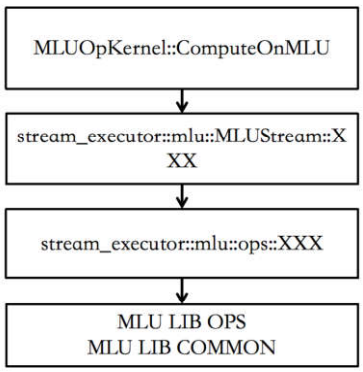

图 5.10　TF 集成流程图

- **算子注册**

  参考第 5 章中注册的 CPU 算子，在 tensorflow/core/kernels/cwise_op_power_difference.cc 文件中添加如图 5.11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（图 4.38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。

```
#define REGISTER_MLU(T)                          \
  REGISTER_KERNEL_BUILDER(                       \
      Name("PowerDifference")                    \
          .Device(Device_MLU)                    \
          .TypeConstraint<T>("T"),               \
      MLUPowerDifferenceOp<T>);
TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);
```

图 5.11　DLP 算子集成（1）：算子注册

- **定义 MLULib 层接口**

  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。具体如图 5.12所示。

- **定义 MLUOp 层接口**

  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。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。具体代码如图 5.13所示。

- **定义 MLUStream 层接口**

  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。其与 MLUOpKernel 类接口关联，负责 MLU 算子的实例化并管理 MLUOP 缓存。具体代码如图 | reffig:tf-combine-specific-4 所示。

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

##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);
```

图 5.12　DLP 算子 TF 集成（2）：定义 MLULib 层接口

```
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(...));
  ...
}
```

图 5.13　DLP 算子集成（3）：定义 MLUOp 层接口

```
  Status PowerDifference(OpKernelContext* ctx,
      Tensor* input1, Tensor* input2, Tensor* output, int input3) {
    return CommonOpImpl<ops::MLUPowerDifference>(ctx,
        {input1, input2}, {output}, static_cast<void*>(&input3));
  }
```

图 5.14　DLP 算子集成（4）：定义 MLUStream 层接口

- **定义 MLUOpKernel 层接口**

  定义 MLUOpKernel 层接口主要是在 MLUOpKernel 层定义 MLUPowerDifferenceOp，在其中通过 streama 机制调用 MLUStream 层具体的 PowerDifference 函数。具体代码如图 5.15所示。

```
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(...));
```

图 5.15　DLP 算子集成（5）：定义 MLUOpKernel 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。完整的单算子 Python 测试代码如图 5.16所示。

5.1.5.4　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 模型可以不分段地跑在 DLP 上，且性能相较于纯 CPU 版本（第 4.4节）和部分 CNML 版本（第 **??**节）都有显著的性能优势。

5.1.5.5　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深度学习框架和深度学习算子库，仅与运行时库 CNRT 相关，因此会减少不必要的开销，提升执行效率。

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，整个推理主要包含着三个步骤：输入数据前处理，离线推理以及后处理。下面将分别介绍这三个部分的实现代码。

1. 主函数：主函数主要用于串联上述的三个步骤的作用，如下图 5.17。

2. 数据前处理:

3. 推理：

```python
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()
```

图 5.16　采用 Python API 对集成的单算子进行测试

4. 后处理：

5. 编译：

5.1.6　实验评估

本次实验中主要考虑基于智能编程语言的算子的实现与验证、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 PowerDifference BCL 算子的实现以及基于 CNRT 的测试，对于实验平台提供的测试数据可以做到精度正常;
- 80 分标准：完成 BCL 算子与 TensorFlow 框架的集成，对于 Python 单算子测试实验精度正常;
- 90 分标准：使用 DLP 推理完整 pb 模型的精度正常且性能达到标准;
- 100 分标准：编写离线推理应用，精度正常且性能达到标准。

```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;
}
```

图 5.17　DLP 离线部署主函数

5.1.7　实验思考

1. BCL 算子如何使用多核进行优化？
2. PowerDifference 算子实现本身性能有哪些提升方法？
3. 融合方式为何可以带来性能的提升？
4. 离线方式为何可以带来性能的提升？
5. 如何更好地利用 DLP 的多核架构来提升性能？

```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);
}
}
```

图 5.18　DLP 离线部署数据前处理

```
#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
```

图 5.19　DLP 离线部署推理

```
#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
```

图 5.20　DLP 离线部署后处理

```
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})
```

图 5.21　DLP 离线部署编译-CMakeLists.txt

5.2 智能编程语言性能优化实验

5.2.1 实验目的

对于一般情况而言，智能编程语言可以简单的快速开发算子从而达到计算功能扩展实现深度学习需求的目的。但是对于一些对算力要求较高的瓶颈算子则需要兼顾算法本身结构特点和 DLP 架构优势进一步的细致优化。因此，本实验的主要目的是通过矩阵乘这样一个在现实中非常常见的计算需求经过一系列的优化手段，最终帮助读者加深对智能计算系统和智能编程语言的理解。

实验工作量：6 学时

5.2.2 背景介绍

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开发和使用。为了更清晰的把握程序的芯片运行时间，在此我们介绍 Notifier 机制。

Notifier 可以看做是一种 CNRT 中特殊类型的任务，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。

```
1
2  cnrtNotifier_t notifier_start, notifier_end;
3  cnrtCreateNotifier(&notifier_start);
4  cnrtCreateNotifier(&notifier_end);
5  cnrtPlaceNotifier(notifier_start, pQueue);
6  ret = cnrtInvokeKernel_V3(reinterpret_cast<void *>(&ROIPoolingKernel), init_param, dim,
       params, c, pQueue, NULL);
7  cnrtPlaceNotifier(notifier_end, pQueue);
8  ret = cnrtSyncQueue(pQueue);
9  cnrtNotifierElapsedTime(notifier_start, notifier_end, &timeTotal);
10 printf("Hardware Total Time: %.3f ms\n", timeTotal / 1000.0);
```

图 5.22 Notifier 机制代码示例

5.2.3 实验环境

本章实验基于前述的 DLP 云平台环境，与上节内容一致。其中有 DLP 的硬件平台和相应的软件环境。

5.2.4 实验内容

本节实验内容主要为矩阵乘算子的性能优化。具体实验内容主要包括：

- 完成 Host 端数据准备，拷贝等功能。
- 使用循环和标量操作实现一个简单的 NRAM 小规模矩阵乘。
- 使用张量操作函数 __bang_conv 实现矩阵乘高速运算。

- 借助 DLP 本身的多核架构实现任务分解和并行化。
- 使用片上 SRAM 达到减轻核间带宽争抢的目的。
- 充分使用片上结构实现数据搬运和计算的流水化作业。

5.2.5 实验步骤

5.2.5.1 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、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，会调用 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，做一定的处理后，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 demo 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，将输入矩阵以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处理时间。Host 端关键代码如 5.23所示：

在优化的过程中，Host 端的代码基本不变，我们重点关注 Device 端代码的开发和优化过程。

5.2.5.2 标量操作实现

这一步是直接在 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，虽然相比于 GDRAM 空间小，但是可以获得更高的读写带宽和更低的访问时延。所以在此种方式中，我们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。需要注意的是，为了展示方便，在这个例子中我们是假设输入的左右矩阵规模都为 256*256，来保证输入的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。在这个程序中无须对输入的矩阵作任何处理，使用矩阵乘公式直接计算，完全没有利用到 DLP 硬件架构的优势，所以整个计算时间很长。

DLP 端代码如 5.24所示：

5.2.5.3 张量操作实现

这里是在上一步的基础上，使用 BCL 提供的向量计算指令完成矩阵乘的计算。采用向量计算指令可以更好地发挥 DLP 硬件性能，减少计算时间。

这里先介绍接下来几步要解决的矩阵乘中的矩阵规模大小问题。为了方便展示和读者理解，这里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）。问题规模如图 5.25所示：

所以在此问题中可以将输入左矩阵一次性拷入 NRAM。同时由于 DLP 架构的特点，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和摆放成特定要求的数据摆放格式，然后使用 __bang_conv 指令进行计算。另外由于右矩阵规模较大，我们在代码中将右矩阵分批次拷入 WRAM 进行计算。

Device 端关键代码如 5.26所示，其中 all_round 表示计算的循环次数，这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，因为此实验环境中的 DLP 上有 64 个卷积计算单元，所以需要将原本顺序摆放的数据按照 64 个为一组间隔摆放。

5.2.5.4 多核并行计算

在上面几步的计算中只调用了 DLP 的一个计算核进行计算，而在这里可以进一步采用 16 个计算核进行并行运算，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。这就将原本由 1 个计算核承担的计算量分摊给了 16 个计算核，大大提高了计算速度。

Device 端代码如 5.27所示。在实现过程中会利用到与并行相关的内置变量。这里大概介绍一下下面代码中使用到的内置变量：taskDim 表示任务规模；taskId 表示程序运行时所分配的任务 ID；在里的方法中 taskDim=16，taskId 范围为 [0,15]。

5.2.5.5 SRAM 的使用

这里在上一步的基础上引入 SRAM，在 DLP 中，每个 cluster 中的 4 个计算核共享一个 SRAM。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，而同 1 个 cluster 上的 4 个计算核在从 GDRAM 上拷贝数据到各自的 NRAM/WRAM 时，会争抢这个 cluster 到 GDRAM 的带宽，从而导致数据读取速度降低。所以我们将数据先从 GDRAM 拷贝到 SRAM，再从 SRAM 分发到 NRAM/WRAM 中，避免了调度争抢问题，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（这个会在下一步的方法中解释），所以这两个操作是并行的关系。为了避免数据冲突，故要设置一个同步，保证数据从 GDRAM 拷入到 SRAM 之后，才能执行从 SRAM 拷入到 NRAM 的过程，在 BCL 中可以使用内置的 __sync_cluster() 函数完成同步功能。如 5.28图所示：

整个执行过程如下图所示:

Device 端关键代码如下，其中 clusterId 表示此时执行任务的是哪个 cluster，范围为 [0,3]：

5.2.5.6 数据搬运与计算的流水

对于 DLP 上的每个 cluster，除了 4 个一般的计算核之外，还有专门用以管理片上总线和 SRAM 的 SRAM 核。这就是上面提到的两种不同功能的核。这同时也为使用流水线的优化手段创造了条件。

所以这里最后一步在上面的基础上，实现了 4 个 cluster 并行，且每个 cluster 中的 SRAM 核和其他 4 个计算核构成流水线的计算模式。在每个 cluster 中，SRAM 核只负责将数据从

GDRAM 拷入 SRAM，其余的每个计算核则负责从 SRAM 拷入数据、矩阵乘计算、将数据拷回 GDRAM。

在此优化方法中，我们设置了在 SRAM 上的两个变量 input2SRAM1，inputSRAM2。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部分从 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 inputSRAM2，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，inputSRAM1 和 inputSRAM2 交替读写重复上述过程直至所有数据计算完成。

从中可以发现耗时很长的从 GDRAM 到 SRAM 的这一步被“藏起来”了。和原来相比，在相同的时间内，我们每次搬运了更多的 GDRAM 数据到片上并且完成了计算。

有一个需要思考的问题是：为什么会使用两个 SRAM 变量对 GDRAM 上的数据进行拷贝呢？因为在上述过程中，计算核在从 SRAM 读取数据的同时，SRAM 核也会从 GDRAM 读数据写入 SRAM。如果只使用一个 SRAM 变量，则很有可能导致计算核应该读取的数据在读取前被写入覆盖。有经验的开发者可能已经发现，这里使用的是一种常用的数据流控制的处理技巧，乒乓操作。

整个过程如 5.31图所示。

Device 端代码如 5.32所示。

5.2.5.7 测试性能

由于循环操作计算矩阵乘性能太差，计算时间太长，没有实际意义，故不在此展示，下图只罗列出后四步在相同规模下的硬件执行时间的比较。

测试环境性能配置参数：思元 270 推理卡，海光 C86 7185 32-core CPU。

m=256 k=256 n=327680 规模	耗时 (ms)	提升幅度
NRAM+conv+ 单核	83.637	
NRAM+conv+16 核	14.142	491.40%
SRAM+conv+16 核	13.026	8.56%
SRAM+conv+16 核 + 流水	12.375	5.26%

5.2.6 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 $m = 256, k = 256, n = 327680$ 下性能与实验效果误差在 10% 以内。
- 80 分标准：进一步通过其他优化手段（加大单次数据调度规模等方法）进一步优化以上问题，计算性能超过基准性能。
- 90 分标准：通过对齐操作使得保证性能的前提下，程序可以接受任意长度的 n 输入。($m = 256, k = 256, n = 1$ 到 327680 的任意正整数)

- 100 分标准：保证性能的情况下，程序可以接受任意长度的 n 和 k 输入。($m = 256, k = 1$ 到 2048 的任意正整数 , $n = 1$ 到 327680 的任意正整数)

5.2.7 实验思考

1. __bang_conv 指令对数据摆放的特殊要求有什么深层次原因？
2. 在 SRAM 的使用代码 5.30中，我们发现代码中有两个 sync，这是为什么？
3. 以上程序目前的瓶颈在哪里？有什么方法可以实证验证？
4. 流水线的设计还有没有其他的方案？有没有可能从全软硬件系统的角度进一步突破？

```
1
2  #include <float.h>
3  #include <math.h>
4  #include <memory.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/time.h>
8  #include <vector>
9  #include "cnrt.h"
10 #include "gemm16Kernel.h"
11
12 #define PAD_UP(x, m) ((x + m - 1) / m * m)
13 #define MP_SELECT 16
14 #define MP1 ((MP_SELECT & 1))
15 #define MP4 ((MP_SELECT & 4))
16 #define MP8 ((MP_SELECT & 8))
17 #define MP16 ((MP_SELECT & 16))
18 #define MP32 ((MP_SELECT & 32))
19 // Mlu_gemm(quantA, quantB, Cmlu, M, N, K, 0xFFFF, 0xFFFF, 1.98438, 1.9, 0.0)
20 //int Mlu_gemm(float *A, const float *B, float *C, int M, int N, int K) {
21 int Mlu_gemm(int8_t *A, int8_t *B, float *C, int32_t M, int32_t N, int32_t K,
22     int16_t pos1, int16_t pos2, float scale1, float scale2,float &return_time) {
23   struct timeval start;
24   struct timeval end;
25   float time_use;
26   int N_align = N;
27 //   int N_align = PAD_UP(N, 16);
28 //   int K_align = PAD_UP(K, 64);
29   cnrtRet_t ret;
30   gettimeofday(&start, NULL);
31
32   cnrtQueue_t pQueue;
33   CNRT_CHECK(cnrtCreateQueue(&pQueue));
34
35   cnrtDim3_t dim;
36   cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;     // CNRT_FUNC_TYPE_BLOCK=1
37   dim.x = 1;
38   dim.y = 1;
39   dim.z = 1;
40
41
42   if (MP1) {
43     dim.x = 1;
44     func_type = CNRT_FUNC_TYPE_BLOCK;
45   } else if (MP4) {
46     dim.x = 4;
47     func_type = CNRT_FUNC_TYPE_UNION1;
48     // printf("UNION1!\n");
49   } else if (MP8) {
50     dim.x = 8;
51     func_type = CNRT_FUNC_TYPE_UNION2;
52   } else if (MP16) {
53     dim.x = 16;
54     func_type = CNRT_FUNC_TYPE_UNION4;
55     // printf("16\n");
56   } else if (MP32) {
57     dim.x = 32;
58     func_type = CNRT_FUNC_TYPE_UNION8;
59   } else {
60     // printf("MP select is wrong! val = %d, use default setting,mp=1\n",
61     //        MP_SELECT);
62     // return -1;
63   }
64
65   gettimeofday(&end, NULL);
66   time_use =
67       ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
```

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

图 5.24　标量操作实现代码示例

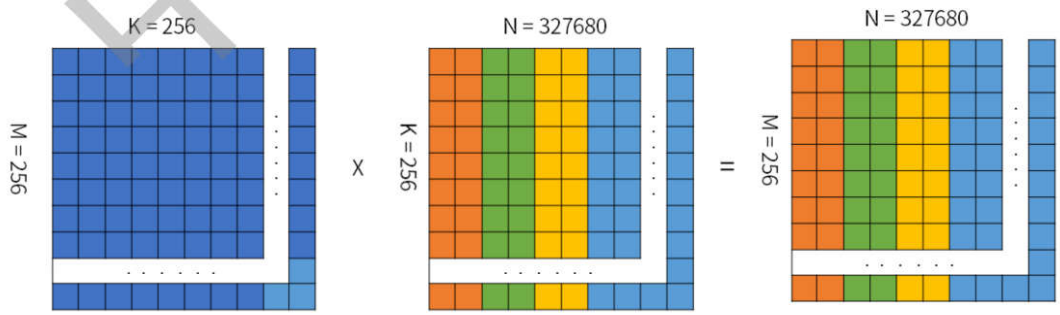

图 5.25　问题中的矩阵规模大小

```
1
2  #include "mlu.h"
3  #define ROUND 256
4  __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
5                                  uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
6      __nram__ int8_t input1NRAM[256*256];
7      __nram__ int8_t input2NRAM[256*256];
8      __nram__ int8_t input2NRAM_tmp[256*256];
9      __wram__ int8_t input2WRAM[256*256];
10     __nram__ half outputNRAM[256*256];
11     __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
12                              //在这里将左矩阵一次性拷入NRAM
13
14     int all_round = n / ROUND;
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     //__bang_printf("taskDim=%d,clusterId=%d,coreId=%d\n",taskDim,clusterId,coreId);
20     for(int i = 0; i < all_round; i++) {
21         __memcpy(input2NRAM_tmp, input2DDR + i * ROUND * k,
22                                     k * ROUND * sizeof(int8_t), GDRAM2NRAM);
23         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
24             __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
25                                         size, NRAM2NRAM, dst_stride, src_stride, 64);
26         }
27         __memcpy(input2WRAM, input2NRAM, ROUND*k*sizeof(int8_t), NRAM2WRAM);
28         __bang_conv(outputNRAM, input1NRAM, input2WRAM, k, m, 1, 1, 1, 1, 1, ROUND, pos)
       ;
29         for (int j = 0; j < m; j++) {    //要对每轮计算的结果进行拼接
30             __memcpy(outputDDR + i * ROUND + j * n, outputNRAM + j * ROUND,
31                                       ROUND * sizeof(half), NRAM2GDRAM);
32         }
33     }
34 }
```

图 5.26　张量计算代码示例

```
1
2  #include "mlu.h"
3  #define ROUND 256
4  __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
5      uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
6      __nram__ int8_t input1NRAM[256*256];
7      __nram__ int8_t input2NRAM[256*256];
8      __nram__ int8_t input2NRAM_tmp[256*256];
9      __wram__ int8_t input2WRAM[256*256];
10     __nram__ half outputNRAM[256*256];
11     __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
12                                  //在这里将左矩阵一次性拷入NRAM
13
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
20     //__bang_printf("taskDim=%d,taskId=%d\n",taskDim, taskId);
21     for(int i = 0; i < all_round; i++) {
22         __memcpy(input2NRAM_tmp, input2DDR + ROUND * (i * taskDim + taskId) * k ,    //只
       涉及这个核需要的数据
23                     k * ROUND * sizeof(int8_t), GDRAM2NRAM);
24         for (int j = 0; j < total_times; j++) {
25             __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
26                         size, NRAM2NRAM, dst_stride, src_stride, 64);
27         }
28         __memcpy(input2WRAM, input2NRAM, ROUND*k*sizeof(int8_t), NRAM2WRAM);
29         __bang_conv(outputNRAM, input1NRAM, input2WRAM, k, m, 1, 1, 1, 1, 1, ROUND, pos)
       ;
30         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
31             __memcpy(outputDDR + (i * taskDim + taskId) * ROUND + j * n,
32                         outputNRAM + j * ROUND, ROUND * sizeof(half), NRAM2GDRAM);
33         }
34     }
35 }
```

图 5.27　多核并行计算代码示例

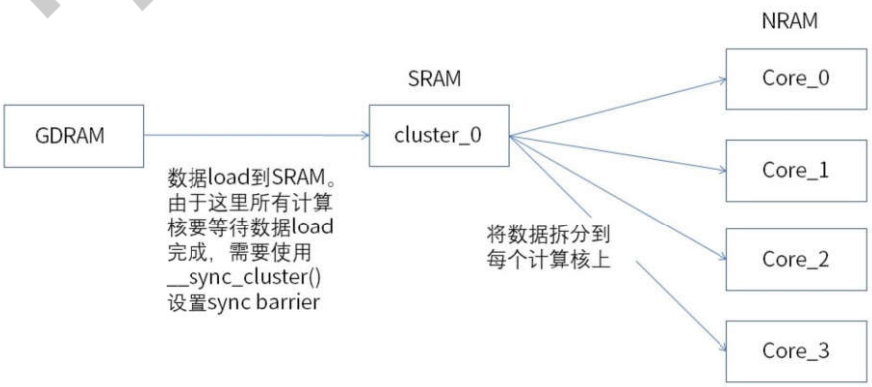

图 5.28　SRAM 的数据同步

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

图 5.29　SRAM 的数据调度时序

```
#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(input2SRAM, input2DDR + ROUND * (i * taskDim + clusterId * 4) * k ,
                 k * ROUND * 4 * sizeof(int8_t), GDRAM2SRAM);       //  只将右矩阵拷入
     SRAM中
        __sync_cluster();    //设置sync barrier
        // copy SRAM2NRAM
        __memcpy(input2NRAM_tmp, input2SRAM + ROUND * coreId * k , k * ROUND * sizeof(
     int8_t), SRAM2NRAM);

        // 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(input2WRAM, input2NRAM, ROUND*k*sizeof(int8_t), NRAM2WRAM);
        // compute
        __bang_conv(outputNRAM, input1NRAM, input2WRAM, k, m, 1, 1, 1, 1, 1, ROUND, pos)
     ;
        // copy NRAM2GDRAM
        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(outputDDR + (i * taskDim + taskId) * ROUND + j * n,
                        outputNRAM + j * ROUND, ROUND * sizeof(half), NRAM2GDRAM);
        }
        __sync_cluster();    //设置sync barrier
    }
}
```

图 5.30　SRAM 使用代码示例

	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 5.31　SRAM 核与计算核的流水数据调度时序

```
#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 input2SRAM1[256*1024];
    __mlu_shared__ int8_t input2SRAM2[256*1024];
    __mlu_shared__ int8_t * input2SRAM_read;
    __mlu_shared__ int8_t * input2SRAM_write;
    input2SRAM_write=input2SRAM1;
    // copy GDRAM2SRAM
    __memcpy(input2SRAM_write, input2DDR + ROUND * (clusterId * 4) * k,
              k * ROUND * 4 * sizeof(int8_t), GDRAM2SRAM);      // 只将右矩阵拷入SRAM中
    __sync_cluster();    //设置sync barrier
    //_bang_printf("taskDim=%d,clusterId=%d,coreId=%d\n",taskDim,clusterId,coreId);
    for(int i = 0; i < all_round-1; i++)
    {
        if (i % 2 == 0)
        {
            input2SRAM_read=input2SRAM1;
            input2SRAM_write=input2SRAM2;
        } else
        {
            input2SRAM_read=input2SRAM2;
            input2SRAM_write=input2SRAM1;
        }
        // copy GDRAM2SRAM
        __memcpy(input2SRAM_write, input2DDR + ROUND * ((i+1) * taskDim + clusterId * 4)
     * k ,
                  k * ROUND * 4 * sizeof(int8_t), GDRAM2SRAM);      //  只将右矩阵拷入
     SRAM中
        // copy SRAM2NRAM
        __memcpy(input2NRAM_tmp, input2SRAM_read + ROUND * coreId * k , k * ROUND *
     sizeof(int8_t), SRAM2NRAM);

        // 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(input2WRAM, input2NRAM, ROUND*k*sizeof(int8_t), NRAM2WRAM);
        // compute
        __bang_conv(outputNRAM, input1NRAM, input2WRAM, k, m, 1, 1, 1, 1, 1, ROUND, pos)
    ;
        // copy NRAM2GDRAM
        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(outputDDR + (i * taskDim + taskId) * ROUND + j * n,
                     outputNRAM + j * ROUND, ROUND * sizeof(half), NRAM2GDRAM);
        }
        __sync_cluster();    //设置sync barrier
    }
    __memcpy(input2NRAM_tmp, input2SRAM_write + ROUND * coreId * k , k * ROUND * sizeof(
     int8_t), SRAM2NRAM);
```