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的开发、优化和集成方法。

具体而言，第 5.1节介绍如何使用智能编程语言 BANG C 实现用户自定义的高性能库算子 Sigmoid，并将其集成到 PyTorch 框架中。第 5.2节进一步介绍如何使用智能编程语言进行向量加法性能优化以充分发挥 MLU 硬件潜力。

5.1　智能编程语言算子开发与集成实验（BANG C 开发实验）

5.1.1　实验目的

掌握使用智能编程语言 BANG C 进行算子开发、编译扩展高性能库算子，并集成到 PyTorch 框架中的方法和流程。能够用 BANG C 实现 Sigmoid 算子，并集成进 PyTorch 的推断网络高效地运行在 MLU 硬件上。

实验工作量：代码量约 150 行，实验时间约 10 小时。

5.1.2　背景介绍

5.1.2.1　BANG C 简介

BANG C 语言采用异构混合编程和编译。一个完整的程序包括主机端（Host）程序和设备端（Device）程序，主机端程序和设备端程序可以写在同一份文件中。混合异构程序使用 CNCC 编译器进行编译，编译器会自动拆分主机端和设备端代码分别编译，最后链接成一个可执行程序。主机端程序主要调用运行时库接口执行内存申请、释放、拷贝，Kernel 的控制执行；设备端程序使用 BANG C 特定的语法规则执行计算部分和并行任务。用户可以在主机端输入数据，做一定处理后，通过一个 Kernel 启动函数将相应输入数据传给设备端，设备端进行计算后，再将计算结果拷回主机端。

5.1.2.2　编译器（CNCC）

CNCC 编译器将使用智能编程语言（BANG C）编写的程序编译成 MLU 架构指令。为了填补高层智能编程语言和底层 MLU 硬件指令间的鸿沟，MLU 的编译器通过寄存器分配、地址空间推断、全局指令调度等技术进行编译优化，以提升生最终二进制程序的性能。

CNCC 的编译过程如图5.1所示，开发者使用 BANG C 开发出的异构混合程序首先通过 CNCC 分离为主机端和设备端代码，然后分别编译为对应架构的中间表示 *.ll 文件，经过优化后编译出汇编 *.s 文件，设备端汇编代码由 CNAS 汇编器生成 MLU 架构的二进制对象 *.o 文件，然后通过设备端链接器链接为多架构混合 *.cnfatbin 文件，最后设备端二进制会被链接进主机端二进制生成最终的二进制可执行文件或动态库。在实际使用中，CNCC 编译器将自动完成上述过程，直接将 BANG C 代码编译、汇编、链接生成二进制机器码。

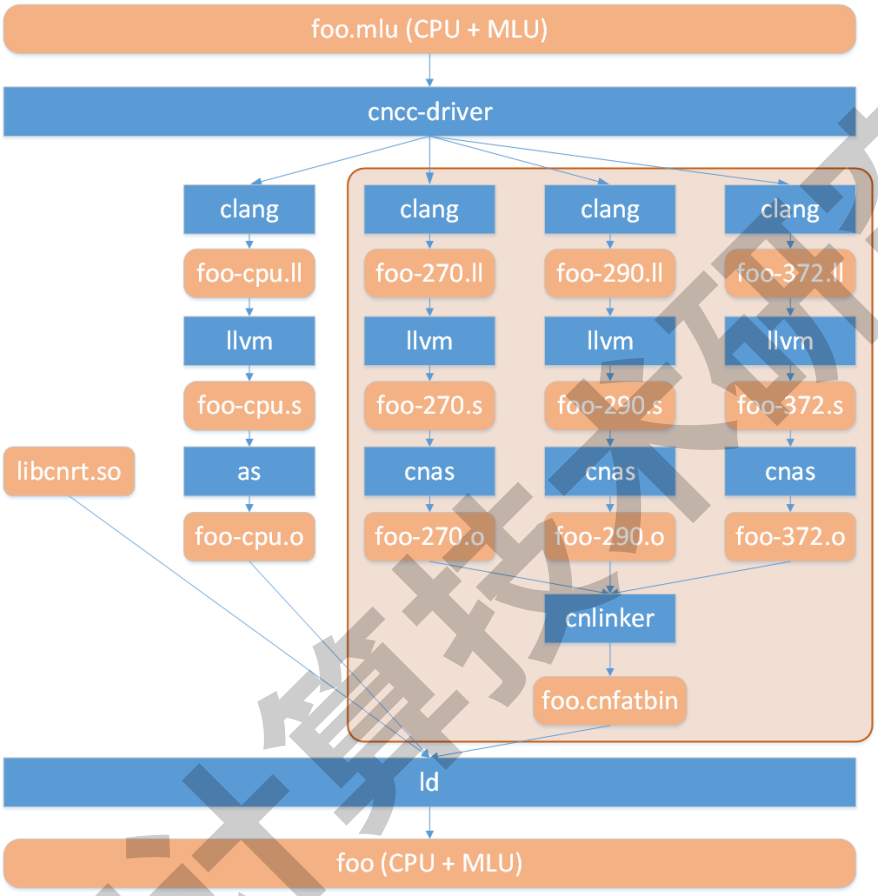

图 5.1　CNCC 编译流程

在使用 CNCC 编译 BANG C 文件时，有多个编译选项供开发者使用，常用选项如表 5.1所示。

5.1.2.3　调试器（CNGDB）

CNGDB 是面向智能编程语言的调试器，支持搭载 MLU 硬件的异构平台调试，即同时支持主机端 C/C++ 代码和设备端 BANG C 代码的调试，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 MLU 架构的特点，调试器可以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。

表 5.1　CNCC 常用编译选项

常用选项	说明
–help	查看 CNCC 帮助信息
-E	编译器只执行预处理步骤，生成预处理文件
-S	编译器只执行预处理、编译步骤，生成汇编文件
-c	编译器只执行预处理、编译、汇编步骤，生成 ELF 格式的汇编文件
-o	将输出写入到指定的文件
-g	在编译时产生调试信息
-O	指定编译优化级别，其中-O0 不做编译优化
–target=	指定可执行文件的目标主机平台架构，不指定时使用当前主机平台架构， 例如 x86_64-linux-gnu，aarch64-linux-gnu 等
–bang-arch=	指定 MLU 的第几代架构，例如 compute_30
–bang-mlu-arch=	指定 MLU 的具体架构号，例如 mtp_372，可以同时指定多个架构进行 fatbin 编译， 例如–bang-mlu-arch=mtp_372，–bang-mlu-arch=mtp_270
–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
–bang-device-only=	面向设备侧编译程序，通常与 -S 选项配合使用，生成 MLISA 汇编代码
-emit-llvm	生成中间表示，通常与 -S 选项配合，生成 LLVM IR 文件
-###	显示编译的子命令行，用来查看异构混合编译的详细流程

如果要使用 CNGDB 进行调试，需要在用 CNCC 编译 BANG C 文件时添加-g 选项、选择-O0 优化级别，如图5.2所示，以编译生成含有调试信息的二进制文件。

```
cncc main.mlu -o a.out --bang-arch=compute_30 -g -O0
```

图 5.2　使用 CNCC 编译生成带调试信息的二进制文件的命令示例

下面以 BANG C 编写的快速排序程序为例, 介绍如何使用 CNGDB 调试程序。快速排序程序的设备端 BANG C 代码文件为 recursion.mlu。图 5.3展示了使用 CNGDB 调试 recursion.mlu 程序的基本流程，主要包含以下几个步骤：断点插入、程序执行、变量打印、单步调试和多核切换等。

5.1.2.4　集成开发环境（CNToolkit 和 CNStudio）

CNToolkit 是基于 BANG 异构计算平台的编译、调试、分析、运行的工具集。CNToolkit 安装包内提供了各个组件的示例代码和用户手册，其中包含 CNStudio 组件。CNStudio 是一款方便在 Visual Studio Code（VSCode）中开发调试 BANG C 语言的编程插件。为了使 BANG C 语言在编写过程中更加方便快捷，CNStudio 基于 VSCode 编辑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。安装包的具体下载地址参考网站（https://developer.cambricon.com）

CNStudio 插件只支持离线安装，安装 CNToolkit 后的插件位置为 /usr/local/neuware/data/cnstudio/cnstudio.vsix。参考用户文档下载并安装 CNToolkit 后，按照图5.4所示的安装流程即可完成 CNStudio 插件的安装。

```
#1.在 CNCC 编译时开启 -g 选项，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=mtp_372 -g -O0

#2.编译得到可运行二进制文件:
g++ recursion.o main.cpp -o quick_sort -lcnrt -I${MLU_INC} -L${MLU_LIB}

#3.在有MLU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 5.3　CNGDB 调试示例

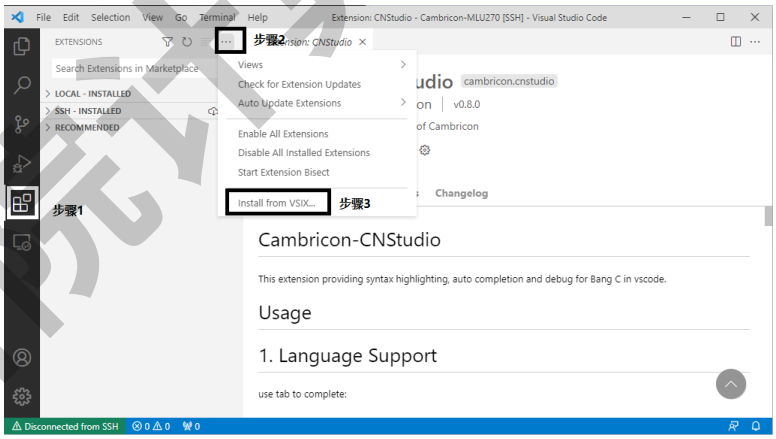

图 5.4　CNStudio 安装流程图

CNStudio 插件安装完毕后，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，若显示图 5.5所示的插件则说明 CNStudio 安装成功。如果 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 MLU 文件夹为例），每个 project 都包含三种类型的文件：设备端使用 BANG C 编写的 Kernel 程序源文件 *.mlu（安装 CNStudio 插件后，VSCode 会

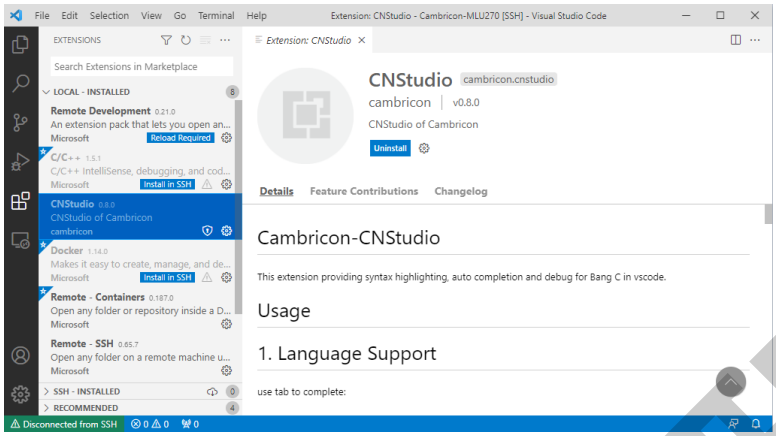

图 5.5　CNStudio 安装完成

自动识别后缀名为 mlu 的文件），主机端的 C++ 程序 main.cpp，以及头文件 kernel.h。通过 VSCode 工具栏中“File” → “Save Workspace As…”，将打开的 MLU 工程保存起来，方便下次直接打开工程文件。

5.1.2.5　高性能算子库 (CNNL 和 MLU-OPS)

CNNL 高性能算子库是基于 BANG C 开发的算子库集合，CNNL 为 PyTorch、TensorFlow、PaddlePaddle 等开源框架提供了运行在 MLU 硬件的完备算子集合，用户无需用 BANG C 开发即可通过 CNNL 运行主流网络的训练和推理并获得最优性能。MLU-OPS 是 CNNL 算子库的开源版本，提供基于 MLU 使用 C 接口或者 Python 接口开发高性能算子的示例代码。MLU-OPS 旨在通过提供示例代码，供开发者参考使用，可用于开发自定义算子，实现对应模型的计算。项目地址 https://github.com/Cambricon/mlu-ops 。

如代码示例图5.6所示，一个高性能算子库的头文件 mlu_ops.h 需要提供 Tensor 描述符 mluOpTensorDescriptor_t，矩阵乘运算描述符 mluOpMatMulDescriptor_t 等数据结构，还需要提供了 mluOpAbs 和 mluOpMatMul 等算子运算接口，以及配套的数据结构和运行时相关的接口如 mluOpCreateTensorSetDescriptor、mluOpCreate、mluOpGetQueue 等。

MLU-OPS 的每个算子都包含主机端代码（mlu-ops/bangc-ops/kernels/abs/abs.cpp 文件）和异构混合 BANG C 代码 (mlu-ops/bangc-ops/kernels/abs/abs_block.mlu 文件)，如图5.7所示。其中主机端代码主要完成算子参数处理、MLU-OPS 接口封装和 Kernel 并行规模策略计算等工作。设备端代码包含 BANG C 源码和 Kernel 的异构 <<<>>>核函数调用，实现主要的计算逻辑。

更多关于智能编程语言的介绍，详见《智能计算系统》教材[1] 第 8 章。

5.1.2.6　PyTorch 框架

MLU 版本的 PyTorch 借助 PyTorch 自身提供的设备扩展接口，将 MLU 后端中所包含的算子操作动态注册到 PyTorch 中，MLU 后端可处理 MLU 上的张量和算子的运算。PyTorch 会基于 CNNL 库在 MLU 后端实现一些常用算子，并完成一些数据拷贝。

```
typedef struct mluOpTensorStruct *mluOpTensorDescriptor_t;

typedef struct mluOpMatMulStruct *mluOpMatMulDescriptor_t;

mluOpStatus_t mluOpAbs(mluOpHandle_t handle,
                       const mluOpTensorDescriptor_t x_desc,
                       const void *x,
                       const mluOpTensorDescriptor_t y_desc,
                       void *y);

mluOpStatus_t mluOpMatMul(mluOpHandle_t handle,
                          const bool is_trans_a,
                          const bool is_trans_b,
                          const void *alpha,
                          const mluOpTensorDescriptor_t a_desc,
                          const void *a,
                          const mluOpTensorDescriptor_t b_desc,
                          const void *b,
                          const void *beta,
                          const mluOpTensorDescriptor_t c_desc,
                          void *c);

mluOpStatus_t mluOpCreateTensorDescriptor(mluOpTensorDescriptor_t *desc);

mluOpStatus_t mluOpCreate(mluOpHandle_t *handle);

mluOpStatus_t mluOpGetQueue(mluOpHandle_t handle, mluQueue_t *queue);
```

图 5.6　MLU-OPS 相关的主要接口

```
mlu-ops/bangc-ops/
├── cmake
├── core
├── kernels
│   ├── abs
│   │   ├── abs.cpp
│   │   ├── abs.h
│   │   └── abs_block.mlu
│   ├── ...
├── scripts
├── test
├── CMakeLists.txt
├── README.md
├── build.sh
├── kernel_depends.toml
└── mlu_ops.h
```

图 5.7　MLU-OPS 的主要目录结构示意

MLU 版本的 PyTorch 兼容原生 PyTorch 的 Python 编程接口和原生 PyTorch 网络模型，支持以在线逐层方式进行训练和以 JIT 融合方式进行推理。网络权重可以从 pth 格式文件读取，已支持的分类和检测网络结构由 Torchvision 管理，可以从 Torchvision 中读取。对于训练任务，支持 float32 及定点量化模型。

为了能在 Torch 模块方便使用 MLU 设备，MLU 版的 PyTorch 在后端进行了以下扩展：

1. 通过 Torch 模块可调用 MLU 后端支持的网络运算。

2. 对 MLU 暂不支持的算子，并且该算子在 MLU 后端库中已添加注册，支持该类算子自动切换到 CPU 上运行。

3.Torch 模块中与 MLU 相关的接口的语义与 CPU 和 GPU 的接口语义保持一致。

4. 支持 CPU 和 MLU 之间的无缝切换。

5.1.3　实验环境

硬件平台：MLU 云平台环境。

软件环境：编程框架 PyTorch、CNNL 高性能算子库、BANG 异构计算平台的开发工具包 CNToolkit。

5.1.4　实验内容

本节实验在第4.4节自定义 PyTorch CPU 算子实验的基础上，进一步用智能编程语言 BANG C 来实现自定义算子 Sigmoid 的计算逻辑（Kernel 函数），通过 PyTorch 的自定义算子扩展机制（MLUExtension 参考 `CUDAExtension`[①]），将自定义算子 Sigmoid 集成到编程框架 PyTorch 中，最后与第4.4节的实现进行精度对比。实验流程主要包括：

(1) BANG C 自定义算子的 Kernel 实现：采用智能编程语言 BANG C 实现自定义算子 Sigmoid 的计算逻辑并进行正确性测试，包括使用 BANG C 的内置向量函数实现 Kernel 函数，通过主机端 C++ 代码调用 Kernel 函数运行并测试功能正确性；

[①]CUDAExtension 的定义参考：https://github.com/pytorch/pytorch/blob/v2.1.0/torch/utils/cpp_extension.py#L975

(2) 框架算子集成：通过 PyTorch 的自定义算子扩展机制对 Sigmoid 算子进行封装，使其调用方式和高性能库原有 MLU 算子一致，然后将封装后的 MLU 算子[①]集成到 PyTorch 框架中并进行测试，保证其精度和功能正确；

(3) MLU 算子和 CPU 算子对比测试：调用 PyTorch 框架的 CPU Sigmoid 算子，和 MLU Sigmoid 自定义算子做精度对比测试。

5.1.5 实验步骤

本节首先介绍使用 BANG C 语言开发和集成自定义算子的主要原理和流程，然后分步骤介绍实现一个 sigmoid 自定义算子的主程序与核函数、通过 pybind11 封装自定义算子接口、使用 setuptools 对自定义算子编译和集成、自定义算子和框架原生算子如何对比测试。

首先，PyTorch 编程框架的设计理念为 “Python First”，所以集成自定义算子的基本思想就是使用 Python 语言的胶水能力将自定义算子嵌入到 PyTorch 框架的算子调用流程中。PyTorch 框架自定义算子的主要流程如下：

(1) 实现主机端的算子主程序和设备端的算子核函数，其中主程序确定算子的输入输出张量并定义算子接口，核函数实现算子的计算；

(2) 在 PyTorch 框架的 `torch.utils.cpp_extension` 模块中添加 `MLUExtension` 函数定义，添加方式类似 `CUDAExtension`[②]或 `CppExtension`，自定义算子在框架中的实现和 PyTorch 框架的 C++ 扩展机制可以参考**??**节介绍。`CUDAExtension` 和 `CppExtension` 是返回 `setuptools.Extension` 类的函数。其中，`CppExtension` 扩展的对象为 CPU，支持的语言是 C++，提供了一些头文件和 PyTorch C++ 相关的静态链接库、动态链接库；而 `CUDAExtension` 扩展的对象是 GPU，支持的语言是 CUDA C++。因此对于 DLP 的 BCL 编程语言，需要添加 `DLPExtension` 来完成编译器查找、编译参数指定等一系列操作；

(3) 编写 setup.py 脚本，使用 setuptools 工具将 BCL 源码编译为动态库，并通过 pybind11 将算子的 API 从 BCL 语言封装为 Python 语言；

然后，结合实验代码分步骤实现、编译、集成、测试一个 sigmoid 算子（函数名称为 active_sigmoid_mlu），算子为单输入单输出，函数接口为 torch::Tensor active_sigmoid_mlu(torch::Tensor x)（函数名称可根据实际需求更改）。

实验的代码目录结构如图5.8所示，其中实验代码的 Python 函数、C++ 函数、MLU 核函数释义如表5.2所示。在完成实验代码补全后，在代码目录结中需要执行的命令步骤如下：

(1) 在根目录下执行 python setup.py install，完成定义算子的编译和安装。

(2) 进入 tests 目录，执行 python test_sigmoid.py 测试完成精度测试。

(3) 对于性能测试，在 test_sigmoid.py 文件中添加计时函数后，执行 python test_sigmoid.py。

实验步骤分为如下几个步骤：

[①] BANG C 自定义算子和 CNNL 内置算子统称为 MLU 算子。

[②] CUDAExtension 的定义参考：https://github.com/pytorch/pytorch/blob/v2.1.0/torch/utils/cpp_extension.py#L975

表 5.2　Sigmoid 函数释义

文件名	函数名	释义
test_sigmoid.py	test_forward_with_shape()	pytest 测试函数
test_sigmoid.py	test_backward_with_shape()	pytest 测试函数
mlu_functions.py	forward()	继承 torch.autograd.function 类的正向 sigmoid 接口
mlu_functions.py	backward()	继承 torch.autograd.function 类的反向 sigmoid 接口
bang_sigmoid.cpp	active_sigmoid_mlu()	C++ 函数接口，属于 torch_mlu 命名空间， 操作的是 PyTorch 的 Tensor， 被 pybind11 封装进 libmlu_custom_ext 库
bang_sigmoid_sample.mlu	bang_sigmoid_kernel_entry()	BANG C 编程中主机端的 C++ 函数入口， 被 Pytorch 的 C++ 接口调用
bang_sigmoid_sample.mlu	bang_sigmoid_kernel()	BANG C 编程中设备端核函数， 被主机端程序使用<<<>>> 核函数调用
bang_sigmoid_sample.mlu	bang_sigmoid_sample()	C++ 测试用例封装的函数接口，仅供 C++ 测试使用， PyTorch 自定义算子中并未调用

```
# Sigmoid自定义算子实验根目录
├── README.md: 描述算子功能的说明文档
├── mlu_custom_ext: 生成的module模块用于在python层导入。
|   ├── __init__.py: python包固有文件
|   ├── mlu: mlu代码文件，根据实际情况自己创建，在setup.py中修改即可。
|   |   ├── include: 头文件目录（头文件和实现分离，属于代码习惯，建议采用此布局）
|   |   |   ├── bang_sigmoid_sample.h: 实现对mlu函数的封装。
|   |   |   ├── kernel.h: BANG C代码中的宏，良好的组织代码的需要。
|   |   |   └── custom_ops.h: 算子对外头文件。
|   |   └── src
|   |       ├── bang_sigmoid.cpp: 对PyTorch层面Tensor的封装，和自定义算子中xxx_internal的实现类似。
|   |       └── bang_sigmoid_sample.mlu: 核心BangC实现。
|   └── mlu_functions: 合理的组织自己的代码方便后续调用。
|       ├── __init__.py: 包必备文件。
|       └── mlu_functions.py: 对C++代码的封装。
├── setup.py: 构建包的脚本。
└── tests
    └── test_sigmoid.py: 对绑定代码的python侧测试。
```

图 5.8　实验代码目录结构

5.1.5.1 实现 Sigmoid 主程序

通过 PyTorch 提供的能力获取 PyTorch Tensor 提供的 Tensor 数据指针，数据指针在 Host 侧无法操作，因此需要实现一个 Device 函数计算 Sigmoid。如代码示例图5.9所示，主程序调用了核函数 bang_sigmoid_kernel_entry 实现对 Device 上的数据计算。

```
//filename: bang_sigmoid.cpp
#include "bang_sigmoid_sample.h"

#include "customed_ops.h"

#include "ATen/Tensor.h"
#include "aten/operators/bang/bang_kernel.h"
#include "aten/operators/bang/internal/bang_internal.h"

using namespace torch_mlu;
torch::Tensor active_sigmoid_mlu(torch::Tensor x) {
  auto x_contiguous = torch_mlu::cnnl_contiguous(x);
  auto x_impl = getMluTensorImpl(x_contiguous);
  auto x_ptr = x_impl->mlu_data_ptr();

  auto y = at::empty_like(x_contiguous);
  auto y_contiguous = torch_mlu::cnnl_contiguous(y);
  auto y_impl = getMluTensorImpl(y_contiguous);
  auto y_ptr = y_impl->mlu_data_ptr();

  int32_t size = x_contiguous.numel();

  cnrtQueue_t queue = getCurQueue();
  // TODO: 请补充Sigmoid主程序函数接口的签名
  ___________________________(
      queue,
      reinterpret_cast<float*>(y_ptr),
      reinterpret_cast<float*>(x_ptr),
      size);

  return y;
}

PYBIND11_MODULE(libmlu_custom_ext, m) {
  m.def("active_sigmoid_mlu", &active_sigmoid_mlu);
}
```

图 5.9 基于智能编程语言 BANG C 的 Sigmoid 主程序

5.1.5.2 实现 Sigmoid 核函数

在 bang_sigmoid_sample.mlu 文件中实现 bang_sigmoid_kernel_entry 函数并通过头文件对外暴露。见代码示例图5.10。

在上述实现中，为了充分利用 MLU 硬件计算能力，使用了向量计算函数来完成 Sigmoid 的运算。为了使用向量计算函数必须满足两个前提：第一是调用计算函数时数据的输入和输出存放位置必须在 NRAM 上，因此必须在计算前使用 memcpy 将数据从 GDRAM 拷贝到

```
//filename: bang_sigmoid_sample.mlu

#include <bang_sigmoid_sample.h>
#include <kernel.h>

__nram__ char NRAM_BUFFER[MAX_NRAM_SIZE];

template<typename T>
__mlu_global__ void bang_sigmoid_kernel(T *d_dst, T *d_src, int N) {
  const int NRAM_LIMIT_SIZE = FLOOR_ALIGN(MAX_NRAM_SIZE / 2, 64);
  int nram_limit = NRAM_LIMIT_SIZE / sizeof(T);
  // 对列数据切分
  int32_t num_per_core = N / taskDim;
  int32_t repeat = num_per_core / nram_limit;
  int32_t rem = num_per_core % nram_limit;

  T *d_input_per_task = d_src + taskId * nram_limit;
  T *d_output_per_task = d_dst + taskId * nram_limit;
  T *nram_out = (T *)NRAM_BUFFER;
  T *nram_in = (T *)(NRAM_BUFFER + NRAM_LIMIT_SIZE);

  const int align_rem = CEIL_ALIGN(rem, 64);

  int i = 0;
  for (; i < repeat; i++) {
    // TODO: 请补充拷贝方向
    __memcpy_async(nram_in, d_input_per_task + i * nram_limit, NRAM_LIMIT_SIZE,
                   __________);
    __sync_io();
    // TODO: 请补充BANG的sigmoid函数
    ____________________(nram_out, nram_in, nram_limit);
    __sync_compute();

    // TODO: 请补充拷贝方向
    __memcpy_async(d_output_per_task + i * nram_limit, nram_out,
                   NRAM_LIMIT_SIZE, __________);

    __sync_io();
  }
  if (rem > 0) {
    // TODO: 请补充拷贝方向
    __memcpy_async(nram_in, d_input_per_task + i * nram_limit,
                   rem * sizeof(T), __________);
    __sync_io();
    // TODO: 请补充BANG的sigmoid函数
    ____________________(nram_out, nram_in, align_rem);
    __sync_compute();
    // TODO: 请补充拷贝方向
    __memcpy_async(d_output_per_task + i * nram_limit, nram_out,
                   rem * sizeof(T), __________);

    __sync_io();
  }
}
template<typename T>
void bang_sigmoid_kernel_entry(cnrtQueue *queue, T *d_dst, T *d_src,
                               int elem_count) {
  cnrtDim3_t dim = {1, 1, 1};
  int taskDims = dim.x * dim.y * dim.z;
  // TODO: 请补充Kernel函数类型
  cnrtFunctionType_t c = ____________________;
  if (elem_count < taskDims) {
    dim.x = 1;
    dim.y = 1;
  }
  // TODO: 请补充Kernel函数的调用
  ____________________________________________________________;
  cnrtQueueSync(queue);
}
```

图 5.10　基于智能编程语言 BANG C 的 Sigmoid 核函数

```
template<typename T>
void bang_sigmoid_sample(T *h_dst, T *h_src, const int elem_count) {

  T *d_src, *d_dst;
  cnrtQueue_t queue;
  cnrtQueueCreate(&queue);
  cnrtRet_t ret;
  ret =
      cnrtMalloc(reinterpret_cast<void **>(&d_src), elem_count * sizeof(T));
  ret =
      cnrtMalloc(reinterpret_cast<void **>(&d_dst), elem_count * sizeof(T));

  ret = cnrtMemcpy(d_src, h_src, elem_count * sizeof(T),
                   CNRT_MEM_TRANS_DIR_HOST2DEV);

  bang_sigmoid_kernel_entry(queue, d_dst, d_src, elem_count);
  cnrtQueueSync(queue);
  // TODO: 请补充Host和Device间的内存拷贝方向
  ret = cnrtMemcpy(h_dst, d_dst, elem_count * sizeof(T),
                   ____________________________);

  ret = cnrtQueueDestroy(queue);
}
template void bang_sigmoid_sample(float*, float*, int);
template void bang_sigmoid_kernel_entry(cnrtQueue *, float *, float *, int);
```

图 5.11　基于智能编程语言 BANG C 的 Sigmoid 核函数（续）

NRAM 上，在计算完成后将结果从 NRAM 拷贝到 GDRAM 上；第二是向量操作的输入规模如果不能被多核和多次循环整除时需要增加分支来处理余数部分。

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，因此需要对原输入数据进行分块。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 NRAM_LIMIT_SIZE = FLOOR_ALIGN(MAX_NRAM_SIZE / 2, 64)。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 NRAM_LIMIT_SIZE 的倍数，所以最后会有一部分长度小于 NRAM_LIMIT_SIZE、大于 0 的余数段。读者在实验时需注意该部分数据的处理逻辑。

5.1.5.3　通过 pybind11 暴露 Op 接口

此接口可以在 Python 中调用，如图5.12即 Python 层可以调用同名函数实现算子计算。

5.1.5.4　使用 setuptools 编译和安装

使用 python setup.py install 将.cpp 和.mlu 文件通过不同的编译器编译，生成最终的动态库。主要包括以下步骤：

(1) CNCC 将.mlu 代码编译为.o 文件。

(2) 将.cpp 代码编译为.so 文件，链接.mlu 文件编译的.o。

实现见图5.13。

```
1 //filename: customed_ops.h
2
3 #pragma once
4 #include <pybind11/pybind11.h>
5 #include <torch/extension.h>
6 torch::Tensor active_sigmoid_mlu(torch::Tensor x);
```

图 5.12　基于智能编程语言 BANG C 的 Sigmoid 接口

编译完成后，在本地会生成一个动态库。一般格式为 name.cpython_version-abi.so，例如：libmlu_custom_ext.cpython-37m-x86_64-linux-gnu.so。此时，即可使用该算子。

5.1.5.5　精度对比测试

mlu_custom_ext 安装后可以通过 pip 工具查看安装情况，接下来就可以参照图5.14调用 PyTorch 框架的 CPU Sigmoid 算子和 MLU Sigmoid 自定义算子做精度对比。这里需要注意的是图5.14中第 22 行 x_cpu.sigmoid() 调用的算子是 PyTorch 框架的原生算子，而不是第4.4小节实现的 CPU Sigmoid 自定义算子，也就是说本小结实现的 MLU 自定义算子和第4.4节实现的 CPU 自定义算子都以 PyTorch 框架原生算子的精度作为对比的真值。

5.1.5.6　性能测试

精度测试通过后，接下来就可以参照图5.15调用 torch.mlu.Event 接口来做性能测试。性能测试时需要注意的是，由于测试代码调用 PyTorch 框架和自定义算子时会触发底层运行时对设备的初始化和核函数二进制模型的加载等动作，所以为了更准确的使用 torch.mlu.Event 对算子计时需要在正式计时前预先执行一遍相同的测试，称为预热。

5.1.6　实验评估

本实验主要关注 BANG C 自定义算子的实现与验证、与框架的集成以及完整的模型推断。模型推断的性能和精度应同时作为主要参考指标。因此，本实验的评估标准设定如下：

- 60 分标准：实现 Sigmoid 主程序、核函数、pybind 接口，使用 setuptools 编译并安装成功；
- 80 分标准：在 60 分基础上，完成精度对比测试用例，使用 numpy 的 assert_array_almost_equal 接口评估精度误差在 decimal=3 以内。
- 100 分标准：在 80 分基础上，完成性能测试，前向 Sigmoid 测试耗时小于 25ms，反向 Sigmoid 测试耗时小于 70ms。

5.1.7　实验思考

(1) 本小结实现的 MLU Sigmoid 自定义算子第4.4小节实现的 CPU Sigmoid 自定义算子精度是否一致?

(2) 使用 BANG C 如何实现一个和 CPU 精度一致的 MLU Sigmoid 算子?

```python
#filename: setup.py

import os
import sys
from setuptools import setup, find_packages

from torch.utils import cpp_extension
from torch_mlu.utils.cpp_extension import MLUExtension, BuildExtension
import glob
import shutil
from setuptools.dist import Distribution

mlu_custom_src = "mlu_custom_ext"
cpath = os.path.join(
    os.path.abspath(os.path.dirname(__file__)),
    os.path.join(mlu_custom_src, "mlu")
)


def source(src):
    cpp_src = glob.glob("{}/*.cpp".format(src))
    mlu_src = glob.glob("{}/*.mlu".format(src))
    cpp_src.extend(mlu_src)
    return cpp_src


def main():
    mlu_extension = MLUExtension(
        name="libmlu_custom_ext",
        sources=source(os.path.join(cpath, 'src')),
        include_dirs=[os.path.join(cpath, "include")],
        verbose=True,
        extra_cflags=['-w'],
        extra_link_args=['-w'],
        extra_compile_args={
            "cxx": [
                "-O3",
                "-std=c++14",
            ],
            "cncc": ["-O3", "-I{}".format(os.path.join(cpath, "include"))]
        })
    dist = Distribution()
    dist.script_name = os.path.basename(sys.argv[0])
    dist.script_args = sys.argv[1:]
    if dist.script_args == ["clean"]:
        if os.path.exists(os.path.abspath('build')):
            shutil.rmtree('build')
    setup(name="mlu_custom_ext",
          version="0.1",
          packages=find_packages(),
          ext_modules=[mlu_extension],
          cmdclass={
              "build_ext":
                  BuildExtension.with_options(no_python_abi_suffix=True)
          })


if __name__ == "__main__":
    main()
```


图 5.13　基于智能编程语言 BANG C 的 Sigmoid 集成

```python
#filename: test_sigmoid.py

import torch
import numpy as np
import torch_mlu
import copy
from mlu_custom_ext import mlu_functions
import unittest


class TestSigmoid(unittest.TestCase):
    """
    test sigmoid
    """

    def test_forward_with_shapes(self, shapes=[(3, 4)]):
        for shape in shapes:
            x_cpu = torch.randn(shape)
            x_mlu = x_cpu.to('mlu')
            # TODO: 请补充mlu_custom_ext库的Sigmoid函数调用
            y_mlu = ____________________(x_mlu)
            y_cpu = x_cpu.sigmoid()
            np.testing.assert_array_almost_equal(y_mlu.cpu(), y_cpu, decimal=3)

    def test_backward_with_shapes(self, shapes=[(3, 4)]):
        for shape in shapes:
            x_mlu = torch.randn(shape, requires_grad=True, device='mlu')
            # TODO: 请补充mlu_custom_ext库的Sigmoid函数调用
            y_mlu = ____________________(x_mlu)
            z_mlu = torch.sum(y_mlu)
            z_mlu.backward()
            grad_mlu = x_mlu.grad
            with torch.no_grad():
                grad_cpu = (y_mlu * (1 - y_mlu)).cpu()
            np.testing.assert_array_almost_equal(grad_mlu.detach().cpu(),
                                                 grad_cpu,
                                                 decimal=3)


if __name__ == '__main__':
    unittest.main()
```

图 5.14　PyTorch Sigmoid 算子的 CPU 和 MLU 对比测试

```python
#filename: test_sigmoid_benchmark.py

import torch
import numpy as np
import torch_mlu
import copy
from mlu_custom_ext import mlu_functions
import unittest


class TestSigmoidBenchmark(unittest.TestCase):
    """
    test sigmoid benchmark
    """

    def test_forward_with_shapes(self, shapes=[(3, 4)]):
        # TODO: 为了计时能准确统计运算部分耗时，请补充预热代码
        for shape in shapes:
            ____________________________

        for shape in shapes:
            event_start = torch.mlu.Event()
            event_end = torch.mlu.Event()

            event_start.record()
            x_cpu = torch.randn(shape)
            x_mlu = x_cpu.to('mlu')
            # TODO: 请补充mlu_custom_ext库的Sigmoid函数调用
            y_mlu = ______________________(x_mlu)
            y_cpu = x_cpu.sigmoid()
            np.testing.assert_array_almost_equal(y_mlu.cpu(), y_cpu, decimal=3)
            event_end.record()

            event_end.synchronize()
            print('forward time: ', event_start.hardware_time(event_end), 'ms')

    def test_backward_with_shapes(self, shapes=[(3, 4)]):
        # TODO: 为了计时能准确统计运算部分耗时，请补充预热代码
        for shape in shapes:
            ____________________________________________________________

        for shape in shapes:
            event_start = torch.mlu.Event()
            event_end = torch.mlu.Event()

            event_start.record()
            x_mlu = torch.randn(shape, requires_grad=True, device='mlu')
            # TODO: 请补充mlu_custom_ext库的Sigmoid函数调用
            y_mlu = ______________________(x_mlu)
            z_mlu = torch.sum(y_mlu)
            z_mlu.backward()
            grad_mlu = x_mlu.grad
            with torch.no_grad():
                grad_cpu = (y_mlu * (1 - y_mlu)).cpu()
            np.testing.assert_array_almost_equal(grad_mlu.detach().cpu(),
                                                 grad_cpu,
                                                 decimal=3)
            event_end.record()

            event_end.synchronize()
            print('backward time: ', event_start.hardware_time(event_end), 'ms')

if __name__ == '__main__':
    unittest.main()
```

图 5.15　PyTorch Sigmoid 算子的性能测试

(3) CPU 和 MLU 算子精度不一致（以 Sigmoid 算子为例）对神经网络的推理精度有何影响?

5.2 智能编程语言性能优化实验

5.2.1 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 MLU 上加速矩阵乘的计算。

实验工作量：代码量约 700 行，实验时间约 6 小时。

5.2.2 背景介绍

5.2.2.1 智能编程模型

如前所述，智能计算系统的层次化抽象[1] 如图**??**所示，多卡的 DLP 服务器可以抽象为五个层次，即服务器级（Server）、板卡级（Card）、芯片级（Chip）、核心簇级（Cluster）和核心级（Core）。第一层是服务器级，整个服务器系统包含若干 CPU 构成的控制单元，以及片外存储器构成的存储单元（主机端内存），由 PCIe 总线互连的若干 DLP 板卡作为该层的计算单元。第二层是板卡级，每块 DLP 板卡上包含片外存储器，板卡上可以有多个 DLP 芯片通过芯粒（Chiplet）封装，多个 DLP 芯片共享片外存储器，每个 DLP 芯片作为计算和控制单元；第三层为芯片级，每个芯片包含多个核心簇作为计算单元，核心簇间共享高速缓存；第四层为核心簇级，核心簇内封装了单个或多个 DLP 核心作为控制和计算单元，核心簇内有核心簇级的存储器做片上的多核数据通信，相比片外存储器可以极大降低访存延迟；第五层为核心级，每个 DLP 核心包含功能单元、寄存器、以及神经元存储器和权重存储器等片上高速存储器。该架构可以很方便地通过增加板卡、芯片、核心簇或者核心等方式提升整个系统的计算能力。

从服务器级依次到处理器核级，存储单元的数据访问延迟依次递减，数据访问带宽依次递增，存储单元的空间大小依次递减。在编程实现时，如果需要将数据从 GDRAM 拷贝到 SRAM，只需调用智能编程语言中的 Memcpy 函数，同时指定拷贝方向为 GDRAM2SRAM。

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。特别地，我们称一次执行只调用一个 Core 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 的任务分别为 UNION2 和 UNION4。

关于智能编程模型更详细的介绍，请参考《智能计算系统》第 8 章。

5.2.2.2 MLU 并行编程

智能编程语言提供了与 Kernel 函数内部任务切分相关的内置变量，方便开发者有效利用 MLU 资源。

Core 变量：

coreDim（核维数）表示一个 Cluster 包含的 Core 个数，例如 MLU370 上等于 4。

coreId（核序号）表示每个 Core 在 Cluster 内的逻辑 ID，例如 MLU370 上的取值范围为 [0-3]。

Cluster 变量：

clusterDim（簇维数）表示启动 Kernel 时指定的 UNION 类型任务调用的 Cluster 个数，例如 UNION4 时等于 4。

clusterId（簇序号）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 分别表示 1 个任务在 X/Y/Z 方向的任务规模，其值等于主机端所指定的任务规模。

taskDim（任务维数）表示用户指定任务的总规模，$taskDim = taskDimX \times taskDimY \times taskDimZ$。

taskIdX/taskIdY/taskIdZ 分别表示程序运行时所分配的逻辑规模在在 X/Y/Z 方向的任务 ID。

taskId（任务序号）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，$taskId = taskIdZ \times taskDimY \times taskDimX + taskIdY \times taskDimX + taskIdX$。

表5.3是一个实际的内置变量取值示例。当程序调用 8 个计算核（UNION2）时，每个核上的并行变量取值如表5.3所示。这里 $\{taskDimX, taskDimY, taskDimZ\}$ 设为 $\{8, 1, 1\}$。

表 5.3　MLU 并行内置变量示例

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

5.2.2.3　Notifier 接口

本实验不涉及深度学习框架集成等系统开发内容，侧重使用智能编程语言进行程序优化。主机端 CPU 代码使用 gettimeofday 来统计矩阵乘运算函数的耗时，设备端 MLU 的硬件耗时，可以使用 Notifier（通知）接口。Notifier 是一种轻量级任务，不像计算任务那样占用计算资源，而是通过驱动从硬件读取一些运行参数，只占用很少的执行时间（几乎可以忽略不计）。Notifier 可以像计算任务任务一样放入 Queue（队列）中执行，在队列中均遵循 FIFO（先进先出）调度原则。可以使用 Notifier 来统计 Kernel 计算任务的硬件执行时间。代码示例5.16是使用 Notifier 机制统计 Kernel 的执行时间的示例，注意 cnrtNotifierElapsedTime 接口返回的耗时单位是毫秒。

```
cnrtNotifier_t start, end;
CNRT_CHECK(cnrtNotifierCreate(&start));
CNRT_CHECK(cnrtNotifierCreate(&end));
CNRT_CHECK(cnrtPlaceNotifier(start, queue));
Kernel<<<...>>>(...);
CNRT_CHECK(cnrtPlaceNotifier(end, queue));
CNRT_CHECK(cnrtSyncQueue(queue));
CNRT_CHECK(cnrtNotifierElapsedTime(start, end, &mlu_time_used));
printf("MLU Time taken: %.3f ms\n", mlu_time_used);
```

图 5.16　Notifier 机制代码示例

5.2.3　实验环境

硬件平台：MLU 云平台环境。

软件环境：CNToolkit 开发工具包。

5.2.4　实验内容

本节实验对于矩阵乘运算，首先使用 BANG C 语言实现一个标量版本，然后利用 NRAM 存储优化、向量化、多核并行优化、三级流水优化、五级流水优化，逐步实现一个高性能矩阵乘，每一个优化步骤都除了和 CPU 版本做性能对比外，还和上一步的 BANG C 优化做对比，从而逐步理解智能编程语言的优化技巧。

(1) 实现一个 CPU 版本的标量矩阵乘，将标量矩阵乘实现迁移到 MLU 上，做性能对比；

(2) 利用 BANG C 的 NRAM 地址空间加速标量版本的矩阵乘，对比性能提升；

(3) 利用 BANG C 提供的向量化接口 __bang_matmul 加速矩阵乘，对比性能提升；

(4) 利用 BANG C 提供的 Block 任务，使用 MLU 的多核做并行加速，对比性能提升；

(5) 利用 BANG C 提供的异步拷贝接口 __memcpy_async 做三级流水优化，对比性能提升；

(6) 利用 BANG C 提供的 Union 任务和 SRAM 地址空间，做五级流水优化，对比性能提升；

5.2.5　实验步骤

5.2.5.1　矩阵乘标量实现

首先实现主机端程序负责为矩阵乘生成随机初始值，然后在主机端实现一个 CPU 版本的标量矩阵乘作为性能基准，参考代码见图**??**

MLU 版本的标量矩阵乘实现和 CPU 一致，由于核函数是执行在 MLU 上的，所以核函数需要在声明时加入特殊的属性，并在主机端使用<<<>>>核函数调用语法。本实验步骤 MLU 标量实现和 CPU 一样只使用一个计算核心，所以在<<<>>>核函数调用时需要指定核函数任务类型 cnrtFunctionType_t 和核函数并行规模 cnrtDim3_t。对于 CPU 使用 gettimeofday 函数计时，精度为微秒级，对于 MLU 使用 cnrtNotifierElapsedTime 函数接口统计核函数的硬

```
// file: 01_scalar.mlu

#include <bang.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define DEBUG 0

#define M 128
#define N 256
#define K 128

float relativeError(float a, float b) {
  float abs_diff = fabs(a - b);
  float max_value = fmax(fabs(a), fabs(b));
  float result = abs_diff / max_value;
  return result;
}

float generateRandomFloat(float min, float max) {
#if DEBUG
  return 1.1;
#else
  float scale = rand() / (float)RAND_MAX;
  return min + scale * (max - min);
#endif
}

__attribute__((noinline)) void multiplyMatricesCPU(float *left, float *right,
                                                   float *result, int m, int n,
                                                   int k) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < k; j++) {
      result[i * k + j] = 0.0f;
      for (int x = 0; x < n; x++) {
        // TODO: 请补充标量矩阵乘计算部分代码
        ----------------------------------------------------------
      }
    }
  }
}

__mlu_entry__ void multiplyMatricesMLU(float *left, float *right, float *result,
                                       int m, int n, int k) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < k; j++) {
      result[i * k + j] = 0.0f;
      for (int x = 0; x < n; x++) {
        // TODO: 请补充标量矩阵乘计算部分代码
        ----------------------------------------------------------
      }
    }
  }
}
```

图 5.17　矩阵乘代码示例

```
int main() {
  int m = M, n = N, k = K;
  printf("\nM = %d, N = %d, K = %d\n", m, n, k);

  float *left = (float *)malloc(m * n * sizeof(float));
  float *right = (float *)malloc(n * k * sizeof(float));
  float *result = (float *)malloc(m * k * sizeof(float));

  float *left_mlu = NULL, *right_mlu = NULL, *result_mlu = NULL;
  CNRT_CHECK(cnrtMalloc((void **)&left_mlu, m * n * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&right_mlu, n * k * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&result_mlu, m * k * sizeof(float)));

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      left[i * n + j] = generateRandomFloat(1.0f, 1.1f);
    }
  }

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < k; j++) {
      right[i * k + j] = generateRandomFloat(1.0f, 1.1f);
    }
  }

  CNRT_CHECK(
      cnrtMemcpy(left_mlu, left, m * n * sizeof(float), cnrtMemcpyHostToDev));
  CNRT_CHECK(
      cnrtMemcpy(right_mlu, right, n * k * sizeof(float), cnrtMemcpyHostToDev));

  struct timeval st_cpu, et_cpu;
  gettimeofday(&st_cpu, NULL);
  multiplyMatricesCPU(left, right, result, m, n, k);
  gettimeofday(&et_cpu, NULL);

  float cpu_time_used = (et_cpu.tv_sec - st_cpu.tv_sec) * 1e3 +
                        (et_cpu.tv_usec - st_cpu.tv_usec) / 1e3;
  printf("\nCPU Time taken: %.3f ms\n", cpu_time_used);
#if DEBUG
  printf("\nCPU Result Matrix:\n");
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < k; j++) {
      printf("%f\t", result[i * k + j]);
    }
    printf("\n");
  }
#endif

  cnrtNotifier_t st_mlu, et_mlu;
  CNRT_CHECK(cnrtNotifierCreate(&st_mlu));
  CNRT_CHECK(cnrtNotifierCreate(&et_mlu));
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));
  cnrtDim3_t dim = {1, 1, 1};
  cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;
  // TODO: 请补充核函数开始计时代码
  ------------------------------------------------
  multiplyMatricesMLU<<<dim, func_type, queue>>>(left_mlu, right_mlu,
                                                 result_mlu, m, n, k);
  // TODO: 请补充核函数结束计时代码
  ------------------------------------------------
  CNRT_CHECK(cnrtQueueSync(queue));
```


图 5.18　矩阵乘代码示例（续 1）

```
56    float mlu_time_used = 0.0f;
57    // TODO: 请补充核函数耗时统计代码
58    ----------------------------------------------------------------------
59    printf("\nMLU Time taken: %.3f ms\n", mlu_time_used);
60
61    float *result_actual = (float *)malloc(m * k * sizeof(float));
62    CNRT_CHECK(cnrtMemcpy(result_actual, result_mlu, m * k * sizeof(float),
63                          cnrtMemcpyDevToHost));
64  #if DEBUG
65    printf("\nMLU Result Matrix:\n");
66    for (int i = 0; i < m; i++) {
67      for (int j = 0; j < k; j++) {
68        printf("%f\t", result_actual[i * k + j]);
69      }
70      printf("\n");
71    }
72  #endif
73
74    bool is_passed = true;
75    for (int i = 0; i < m; i++) {
76      for (int j = 0; j < k; j++) {
77        float diff_rel =
78            relativeError(result[i * k + j], result_actual[i * k + j]);
79        if (diff_rel > 0.1) {
80          printf("diff_rel = %.3f\n", diff_rel);
81          printf("[%d, %d]: cpu = %f, mlu = %f\n", i, j, result[i * k + j],
82                 result_actual[i * k + j]);
83          is_passed = false;
84        }
85      }
86    }
87    printf(is_passed ? "\nPASSED\n" : "\nFAILED\n");
88
89    free(left);
90    CNRT_CHECK(cnrtFree(left_mlu));
91    free(right);
92    CNRT_CHECK(cnrtFree(right_mlu));
93    free(result);
94    free(result_actual);
95    CNRT_CHECK(cnrtFree(result_mlu));
96    CNRT_CHECK(cnrtNotifierDestroy(st_mlu));
97    CNRT_CHECK(cnrtNotifierDestroy(et_mlu));
98    CNRT_CHECK(cnrtQueueDestroy(queue));
99
100   return 0;
101 }
```

图 5.19　矩阵乘代码示例（续 2）

件耗时,精度同样为微秒级。5.19代码示例中选用的较小规模的矩阵乘(MxNxK=128x256x128)的原因是标量单核版本的实现在 MLU 上执行过慢会触发运行时的超时检查，后续实现步骤中随着优化加速，会选用较大规模的矩阵乘和 CPU 版本做性能对比。

本实验步骤中 CPU 和 MLU 的乘累加顺序一致，所以计算精度误差为 0，后面的实验步骤中由于使用了 BANG C 提供的向量化运算接口，乘累加的顺序和基准版本的 CPU 实现不一致，所以会导致校验精度时误差不为 0，所以测试代码中应采用相对误差来评估精度。

5.2.5.2 矩阵乘标量 NRAM 实现

本实验步骤目的是利用 BANG C 的 NRAM 地址空间来加速标量版本的矩阵乘，原理是通过 BANG C 的静态数组在 NRAM 上为左右矩阵和结果矩阵声明空间，将小规模的矩阵一次性拷贝至 NRAM，在 NRAM 上运算完成后一次性拷出到 GDRAM 空间，由于标量读写 NRAM 空间的性能远高于 GDRAM 空间，所以性能会有几十倍的提升。

```
1  // file: 02_scalar_nram.mlu
2  #define M 128
3  #define N 256
4  #define K 128
5
6  __mlu_entry__ void multiplyMatricesMLU(float *left, float *right, float *result,
7                                         int m, int n, int k) {
8    __nram__ float left_nram[M * N];
9    __nram__ float right_nram[N * K];
10   __nram__ float result_nram[M * K];
11   // TODO: 请补充将左矩阵从GDRAM拷贝至NRAM的代码
12   ----------------------------------------------------------------------
13   // TODO: 请补充将右矩阵从GDRAM拷贝至NRAM的代码
14   ----------------------------------------------------------------------
15   for (int i = 0; i < m; i++) {
16     for (int j = 0; j < k; j++) {
17       result_nram[i * k + j] = 0.0f;
18       for (int x = 0; x < n; x++) {
19         // TODO: 请补充标量矩阵乘计算部分代码
20         ------------------------------------------------------
21       }
22     }
23   }
24   // TODO: 请补充将结果矩阵从NRAM写回GDRAM的代码
25   ----------------------------------------------------------------------
26 }
```

图 5.20 矩阵乘标量 NRAM 实现

5.2.5.3 矩阵乘向量 NRAM 实现

MLU 硬件架构支持 SIMD 指令集，所以为了发挥算力优势，必须调用 BANG C 封装的 Builtin 函数 __bang_matmul 进行向量化加速。本实验步骤仍然用较小规模的单核矩阵乘，仅仅将标量循环实现替换为调用 __bang_matmul 函数实现，性能将有几千倍量级的提升。

```
// file: 03_vector_nram.mlu
#define M 128
#define N 256
#define K 128

__mlu_entry__ void multiplyMatricesMLU(float *left, float *right, float *result,
                                       int m, int n, int k) {
  __nram__ float left_nram[M * N];
  __wram__ float right_wram[N * K];
  __nram__ float result_nram[M * K];
  // TODO: 请补充将左矩阵从GDRAM拷贝至NRAM的代码
  ----------------------------------------------------------------------------
  // TODO: 请补充将右矩阵从GDRAM拷贝至WRAM的代码
  ----------------------------------------------------------------------------
  __bang_matmul(result_nram, left_nram, right_wram, m, n, k);
  // TODO: 请补充将结果矩阵从NRAM写回GDRAM的代码
  ----------------------------------------------------------------------------
}
```

图 5.21　矩阵乘标量 NRAM 实现

5.2.5.4　矩阵乘多核向量 NRAM 实现

对于多核向量实现，128x256x128 的矩阵乘规模过小，上一个实验步骤中 MLU 的耗时已经降到了 50 微秒量级，计时函数的精度为微秒量级，所以本小结将 M 扩大到 M_PER_BLOCK * BLOCKS = 524288，其中 M_PER_BLOCK 表示每个任务处理的 M 维度的元素个数，BLOCKS 表示<<<>>>核函数的并行任务数。本实验步骤利用多核加速的原理是将超大规模的矩阵乘，在 M 维度上拆分为 BLOCKS 个可并行的小矩阵乘，然后利用 BANG C 提供的 taskId、taskDim 等并行变量将核函数改写为并行版本，改写后相比 CPU 的标量单核版本基线要快几千至上万倍。本实验步骤参考代码见图5.22。由于片上 NRAM 和 WRAM 空间容量有限，所以对于 M=524288 这样大规模的矩阵乘，需要首先做多核拆分 M = M_PER_BLOCK * BLOCKS，其次还可以在单核内做循环拆分 M_PER_BLOCK = M_PER_LOOP * LOOPS，多核拆分的 BLOCKS 不是越多越好，因为 BLOCKS 个并行任务最终要映射至有限个 MLU 和计算核心，每启动一次 MLU 核心都会有一定的硬件和软件开销，软件开销与核函数的实现有关，例如核函数中循环体外部代码，如果启用的 BLOCKS 过多，每个 MLU 核心都要执行一遍重复的运算，所以最优的 BLOCKS 数量应该等于 MLU 芯片的物理核心数，然后将并行运算代码放在循环体内。

5.2.5.5　矩阵乘多核向量 NRAM 三级流水实现

MLU 架构的 NRAM、WRAM、GDRAM 等各级存储之间支持异步的 DMA 数据拷贝，而且访存单元和运算单元之间是可以并行执行的，这就为软件流水优化提供了可能。BANG C 提供了异步拷贝接口 __memcpy_async 和同步接口 __sync，参考图5.23的三级流水实现，可以进一步缓解访存瓶颈，即缓解访存墙带来的喂不饱 MLU 的运算单元的情况。

简单来说，软件流水就是对循环中的操作进行调整，使尽可能多的操作可以并行执行。如5.24所示，在向量加法案例中，整体逻辑可以分为三部分：Load，Compute 和 Store。为了

```
// file: 04_vector_nram_blocks.mlu
#define LOOPS 32
#define BLOCKS 128
#define M_PER_LOOP 128
#define M_PER_BLOCK (M_PER_LOOP * LOOPS)

#define M (M_PER_BLOCK * BLOCKS)
#define N 256
#define K 128

__mlu_entry__ void multiplyMatricesMLU(float *left, float *right, float *result,
                                       int m, int n, int k) {
  // TODO: 请补充左矩阵NRAM数组的声明
  __nram__ float left_nram[_________________];
  __wram__ float right_wram[N * K];
  // TODO: 请补充结果矩阵NRAM数组的声明
  __nram__ float result_nram[_________________];
  int m_per_block = m / taskDim;
  int m_per_loop = m_per_block / LOOPS;
  // TODO: 请补充将右矩阵从GDRAM拷贝至WRAM的代码
  ____________________________________________________________
  for (int loop = 0; loop < LOOPS; loop++) {
    // TODO: 请补充将左矩阵从GDRAM拷贝至NRAM的代码
    __memcpy(left_nram, ____________________________________________,
             _________________________________, GDRAM2NRAM);
    __bang_matmul(result_nram, left_nram, right_wram, m_per_loop, n, k);
    // TODO: 请补充将结果矩阵从NRAM写回GDRAM的代码
    __memcpy(_______________________________________________________,
             result_nram, _________________________________, NRAM2GDRAM);
  }
}

int main() {
  ...
  cnrtNotifier_t st_mlu, et_mlu;
  CNRT_CHECK(cnrtNotifierCreate(&st_mlu));
  CNRT_CHECK(cnrtNotifierCreate(&et_mlu));
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));
  // TODO: 请补充核函数并行规模
  cnrtDim3_t dim = {______, 1, 1};
  cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;
  // TODO: 请补充核函数开始计时代码
  ____________________________________________
  multiplyMatricesMLU<<<dim, func_type, queue>>>(left_mlu, right_mlu,
                                                 result_mlu, m, n, k);
  // TODO: 请补充核函数结束计时代码
  ____________________________________________
  CNRT_CHECK(cnrtQueueSync(queue));

  float mlu_time_used = 0.0f;
  // TODO: 请补充核函数耗时统计代码
  ____________________________________________________________________
  printf("\nMLU Time taken: %.3f ms\n", mlu_time_used);
  ...
}
```

图 5.22　矩阵乘多核向量 NRAM 实现

```
// file: 05_vector_nram_blocks_pipe3.mlu
#define STAGES 2
#define LOOPS 32
#define BLOCKS 128
#define M_PER_LOOP 128
#define M_PER_BLOCK (M_PER_LOOP * LOOPS)

#define M (M_PER_BLOCK * BLOCKS)
#define N 256
#define K 128

__mlu_entry__ void multiplyMatricesMLU(float *left, float *right, float *result,
                                       int m, int n, int k) {
  // TODO: 请补充左矩阵NRAM数组的声明
  __nram__ float left_nram[__________________________];
  __wram__ float right_wram[N * K];
  // TODO: 请补充结果矩阵NRAM数组的声明
  __nram__ float result_nram[__________________________];
  int m_per_block = m / taskDim;
  int m_per_loop = m_per_block / LOOPS;
  // TODO: 请补充将右矩阵从GDRAM异步拷贝至WRAM的代码
  ____________________________________________________________
  for (int loop = 0; loop < (LOOPS + STAGES); loop++) {
    if (loop < LOOPS) {
      // TODO: 请补充将左矩阵从GDRAM异步拷贝至NRAM的代码
      __memcpy_async(__________________________________________,
                     ____________________________________________________,
                     m_per_loop * n * sizeof(float), GDRAM2NRAM);
    }
    if (loop >= 1 && loop <= LOOPS) {
      __bang_matmul(result_nram + m_per_loop * k * ((loop - 1) % STAGES),
                    left_nram + m_per_loop * n * ((loop - 1) % STAGES),
                    right_wram, m_per_loop, n, k);
    }
    if (loop >= STAGES) {
      // TODO: 请补充将结果矩阵从NRAM写回GDRAM的代码
      __memcpy_async(
          ____________________________________________________________,
          ____________________________________________________,
          m_per_loop * k * sizeof(float), NRAM2GDRAM);
    }
    __sync();
  }
}
```

图 5.23　矩阵乘多核向量 NRAM 三级流水实现

避免 Load 和 Store 产生冲突，需要引入两块独立的 Ping-Pong 缓冲区。由于 NRAM 大小有限，要进行多次普通的向量计算过程如图中绿色方框所示，顺序执行 0->1->2，即 [L0-C0-S0]->[L1-C1-S1]-> [L2-C2-S2] 。如果通过软流水技术重新排列，则变成橘红色方框格式，顺序执行 a->b->c，即 [S0-C1-L2]->[S1-C2-L3]->[S2-C3-L4]，可以写成新的循环形式。

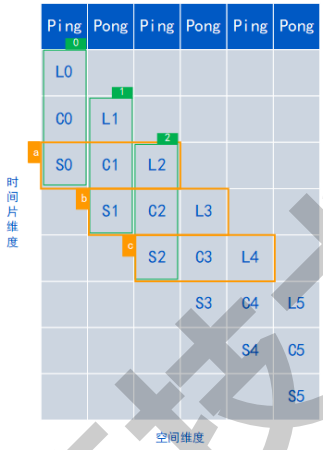

图 5.24　异步流水示意图

5.2.5.6　矩阵乘多核向量 SRAM 五级流水实现

五级流水相比三级流水优化要多出一级 SRAM 地中空间，见图5.26。

5.2.5.7　性能对比

测试环境参数如下：

CPU：Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz MLU: MLU370-X8@1000MHz MLU Driver: v5.10.26 MLU Firmware: v1.1.6 CNToolkit: v3.9.0

```
// file: 06_vector_sram_unions_pipe5.mlu
#define STAGES 2
#define LOOPS 32
#define BLOCKS 128
#define M_PER_LOOP 64
#define M_PER_LOOP_BLOCK (M_PER_LOOP * STAGES)
#define M_PER_LOOP_UNION (M_PER_LOOP_BLOCK * /*coreDim=*/4)
#define M_PER_BLOCK (M_PER_LOOP_BLOCK * LOOPS)

#define M (M_PER_BLOCK * BLOCKS)
#define N 256
#define K 128

__mlu_func__ void multiplyMatrices(float *left, float *right_wram,
                                   float *result, int m, int n, int k) {
  // TODO: 请补充左矩阵NRAM数组的声明
  __nram__ float left_nram[__________________________];
  // TODO: 请补充结果矩阵NRAM数组的声明
  __nram__ float result_nram[__________________________];
  int m_per_block = m / coreDim;
  int m_per_loop = m_per_block / STAGES;
  int loops = STAGES;
  for (int loop = 0; loop < (loops + STAGES); loop++) {
    if (loop < loops) {
    __sync_io();
      // TODO: 请补充将左矩阵从SRAM异步拷贝至NRAM的代码
      __memcpy_async(_____________________________________________,
                     _____________________________________________,
                     m_per_loop * n * sizeof(float), SRAM2NRAM);
    }
    if (loop >= 1 && loop <= loops) {
      __bang_matmul(result_nram + m_per_loop * k * ((loop - 1) % STAGES),
                    left_nram + m_per_loop * n * ((loop - 1) % STAGES),
                    right_wram, m_per_loop, n, k);
    }
    if (loop >= STAGES) {
      // TODO: 请补充将结果矩阵从NRAM写回SRAM的代码
      __memcpy_async(
          ____________________________________________________________________,
          ____________________________________________________,
          m_per_loop * k * sizeof(float), NRAM2SRAM);
    }
    __sync_move();
  }
}

__mlu_entry__ void multiplyMatricesMLU(float *left, float *right, float *result,
                                       int m, int n, int k) {
  // TODO: 请补充左矩阵SRAM数组的声明
  __mlu_shared__ float left_sram[_______________________________];
  __mlu_shared__ float right_sram[N * K];
  __wram__ float right_wram[N * K];
  // TODO: 请补充结果矩阵SRAM数组的声明
  __mlu_shared__ float result_sram[_______________________________];
  int m_per_block = m / taskDim;
  int m_per_union = m_per_block * coreDim;
  int m_per_loop_union = m_per_union / LOOPS;
  // TODO: 请补充将右矩阵从GDRAM异步拷贝至SRAM的代码
  ____________________________________________________________
  // TODO: 请补充将右矩阵从SRAM异步拷贝至NRAM的代码
  ____________________________________________________________________
```


图 5.25　矩阵乘多核向量 SRAM 五级流水实现

```
  for (int loop = 0; loop < (LOOPS + STAGES); loop++) {
    if (loop < LOOPS) {
      // TODO: 请补充将左矩阵从GDRAM异步拷贝至SRAM的代码
      __memcpy_async(_____________________________________________________,
                     ____________________________________________
                         ____________________________,
                     m_per_loop_union * n * sizeof(float), GDRAM2SRAM);
    }
    if (loop >= 1 && loop <= LOOPS) {
      multiplyMatrices(
          left_sram + m_per_loop_union * n * ((loop - 1) % STAGES), right_wram,
          result_sram + m_per_loop_union * k * ((loop - 1) % STAGES),
          m_per_loop_union, n, k);
    }
    if (loop >= STAGES) {
      // TODO: 请补充将结果矩阵从SRAM写回GDRAM的代码
      __memcpy_async(
          ____________________________________________
              ____________________________________,
          ____________________________________________________________,
          m_per_loop_union * k * sizeof(float), SRAM2GDRAM);
    }
    __sync_cluster();
  }
}
int main() {
  ...
  cnrtNotifier_t st_mlu, et_mlu;
  CNRT_CHECK(cnrtNotifierCreate(&st_mlu));
  CNRT_CHECK(cnrtNotifierCreate(&et_mlu));
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));
  // TODO: 请补充核函数并行规模
  cnrtDim3_t dim = {CNRT_FUNC_TYPE_UNION1, _________________________________, 1};
  cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_UNION1;
  // TODO: 请补充核函数开始计时代码
  ___________________________________________
  multiplyMatricesMLU<<<dim, func_type, queue>>>(left_mlu, right_mlu,
                                                 result_mlu, m, n, k);
  // TODO: 请补充核函数结束计时代码
  ___________________________________________
  CNRT_CHECK(cnrtQueueSync(queue));

  float mlu_time_used = 0.0f;
  // TODO: 请补充核函数耗时统计代码
  ____________________________________________________________________
  printf("\nMLU Time taken: %.3f ms\n", mlu_time_used);
  ...
}
```

图 5.26　矩阵乘多核向量 SRAM 五级流水实现（续）

实验步骤	矩阵乘规模 (MxNxK)	CPU 耗时 (ms)	MLU 耗时 (ms)
标量实现	128x256x128	5.857	2716.140
标量 NRAM 实现	128x256x128	5.847	126.067
向量 NRAM 实现	128x256x128	5.904	0.050
多核向量 NRAM 实现	524288x256x128	26075.229	4.563
多核向量 NRAM 三级流水实现	524288x256x128	26076.031	3.824
多核向量 SRAM 五级流水实现	524288x256x128	26081.332	4.073

5.2.6 实验评估

本实验设定的评估标准如下，每升高一级标准，不但要实现当前分值标准的代码，还要实现上一级分值标准的代码，即 100 分标准要求实现“标量 NRAM 实现”、“向量 NRAM 实现”、“多核向量 NRAM 实现”、“多核向量 NRAM 三级流水实现”、“多核向量 SRAM 五级流水实现”共 5 个版本的 BANG C 代码。

- 60 分标准：MxNxK = 128x256x128 规模，实现标量 NRAM 矩阵乘，MLU 计算结果与 CPU 计算结果误差为 0。
- 70 分标准：MxNxK = 128x256x128 规模，实现向量 NRAM 矩阵乘，精度达标，MLU 性能为 CPU 的 xxx 倍以上。
- 80 分标准：MxNxK = 524288x256x128 规模，实现多核向量 NRAM 矩阵乘，精度达标，MLU 性能为 CPU 的 xxx 倍以上。
- 90 分标准：MxNxK = 524288x256x128 规模，实现多核向量 NRAM 三级流水矩阵乘，精度达标，MLU 性能为 CPU 的 xxx 倍以上。
- 100 分标准：MxNxK = 524288x256x128 规模，实现多核向量 NRAM 五级流水矩阵乘，精度达标, 性能和三级流水在同一个数量级。

5.2.7 实验思考

(1) CPU 上实现矩阵乘有哪些可以加速的方法？请尝试改写 CPU 版本的矩阵乘实现和 MLU 重新做性能对比并分析峰值性能的上限由什么因素决定。

(2) 为什么矩阵乘的标量实现中 CPU 和 MLU 的精度一致，而使用了向量接口 __bang_matmul 后精度不一致？是 BANG C 提供的向量加速接口计算错误么？

(3) 什么是分块矩阵乘？分块矩阵乘和不分块时，精度由何差异？精度差异的来源是？