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 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 开发实验）

5.1.1　实验目的

掌握使用智能编程语言 BANG 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 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 编程流程

智能编程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 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）编写的程序编译成 MLU 底层指令。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，以提升生成的二进制代码性能。

CNCC 的编译过程如图5.1所示，开发者使用 BANG 开发出的异构混合程序首先通过 CNCC 前端分离为主机端和设备端代码，然后由前端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代码，设备端汇编代码由 CNAS 汇编器生成 MLU 上运行的二进制机器码，最后设备端二进制会被链接进主机端二进制生成最终的主机端二进制可执行文件。在实际使用中，CNCC 编译器将自动完成上述过程，直接将 BANG 代码编译生成二进制机器码。

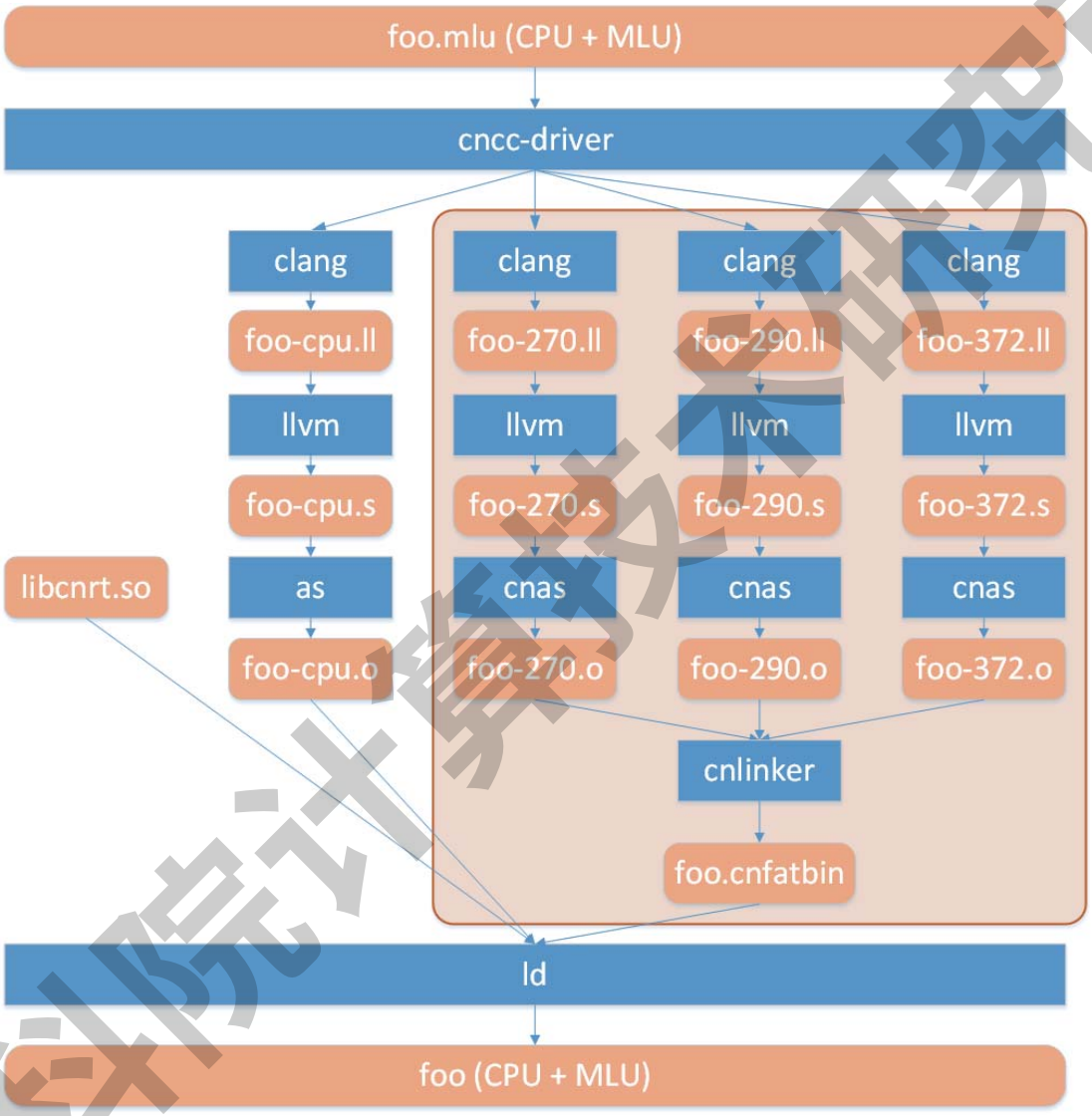

图 5.1　CNCC 编译流程

在使用 CNCC 编译 BANG 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。

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

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 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。

如果要使用 CNGDB 进行调试，需要在用 CNCC 编译 BANG 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。

```
1 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 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 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。

```
#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

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 语言的编程插件。为了使 BANG 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，可按照图5.4所示的安装流程即可完成 CNStudio 插件的安装。

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：MLU 端使用 BANG 编写的 Kernel 程序源文件 mlu.mlu（设备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），主机端的 C++ 程序 main.cpp，以及

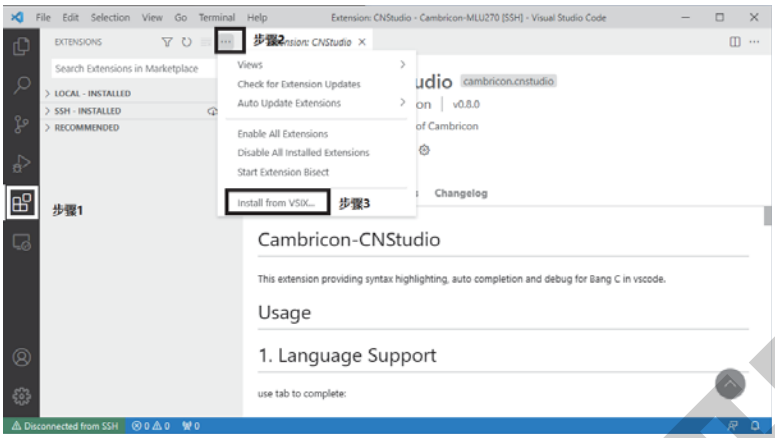

图 5.4　CNStudio 安装流程图

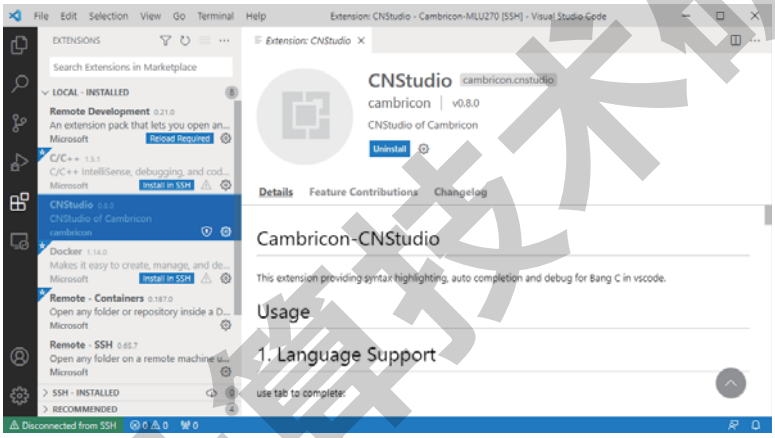

图 5.5　CNStudio 安装完成

头文件 kernel.h。通过 VSCode 工具栏中“File”→“Save Workspace As…”，将打开的 MLU 工程保存为 workspace，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 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 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，以及与之配套的数据结构和运行时相关的接口 mluOpCreateTensorSetDescriptor、mluOpCreate、mluOpGetQueue 等。

```
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 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 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 源码和 Kernel 的异构 <<<>>>函数调用，实现主要的计算逻辑。

```
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

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

借助 PyTorch 自身提供的设备扩展接口将 MLU 后端库中所包含的算子操作动态注册到 PyTorch 中，MLU 后端库可处理 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。

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。对于推理任务，暂时只支持 float32 数据类型。

为了能在 Torch 模块方便使用 MLU 设备，在 PyTorch 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、MLU 开发工具包 CNToolkit。

5.1.4 实验内容

本节实验在第4.4节中的高性能库算子实验的基础上，进一步用智能编程语言 BANG 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节的实现进行性能对比。实验流程主要包括：

(1) BANG 自定义算子的 Kernel 实现：采用智能编程语言 BANG 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 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；

(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[②]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

本节介绍如何实现本实验涉及的各个步骤，包括 BANG 自定义算子的计算逻辑开发、框架集成、对比测试等。

PyTorch 提供的自定义算子实现接口的原理是 MLUExtension 将获取的参数通过内部实现的 BuildExtension 完成对编译器查找、编译参数指定等一系列操作，最后将源码编译为动态库，通过 Python 实现调用。其中，编译动态库通过 Python 基础功能实现，Python 调用对应函数则通过 pybind11 实现。

算子实现逻辑主要分为两步：

(1) 确定算子在 Tensor 层的输入输出，定义算子接口，对应 *.cpp 文件。

(2) 实现 Device 端的计算，对应 *.mlu 文件。

下面，我们实现一个计算 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。

实验步骤分为如下几个步骤：

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。

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

[②]BANG 自定义算子和 CNNL 内置算子统称为 MLU 算子。

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 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 编程中设备端核函数， 被主机端程序使用 «<>» 语法调用
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

```
# mlu_extension
├── 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（建议使用目录管理BANG代码）。
│   │   ├── 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

```cpp
//filename: bang_sigmoid.cpp


#include <bang_sigmoid_sample.h>
#include <customed_ops.h>

#include "ATen/Tensor.h"
#include "aten/cnnl/cnnlHandle.h"
#include "aten/cnnl/cnnl_util.h"
#include "aten/operators/bang/bang_kernel.h"
#include "aten/operators/bang/internal/bang_internal.h"
#include "aten/util/tensor_util.h"
#include "aten/util/types.h"

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

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

  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) {
  // TODO: 请补充pybind函数定义
  m.def("__________________", ____________________);
}
```

图 5.9　基于智能编程语言 BANG 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 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。

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 算子做精度对比。

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。

5.1.6 实验评估

本实验主要关注 BANG 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：

```
//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();
  }
}
```

图 5.10　基于智能编程语言 BANG 的 Sigmoid 核函数

```
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);
}
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 的 Sigmoid 核函数（续）

```
//filename: customed_ops.h

#pragma once
#include <pybind11/pybind11.h>
#include <torch/extension.h>
torch::Tensor active_sigmoid_mlu(torch::Tensor x);
```

图 5.12　基于智能编程语言 BANG 的 Sigmoid 接口

```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
import distutils
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 的 Sigmoid 集成

```python
#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
#test_sigmoid.py

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

    def test_forward_with_shapes(self, shapes=[(3, 4)]):
        # TODO: 请补充Warm up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()

            torch.mlu.synchronize()
            print('forward time: ', event_start.elapsed_time(event_end), 'ms')

    def test_backward_with_shapes(self, shapes=[(3, 4)]):
        # TODO: 请补充Warm up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()

            torch.mlu.synchronize()
            print('backward time: ', event_start.elapsed_time(event_end), 'ms')

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

图 5.15　PyTorch Sigmoid 算子的性能测试

• 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 0.5ms，反向 Sigmoid 测试耗时小于 1.2ms。

5.1.7 实验思考

(1) 有哪些方法可以提升 Sigmoid 算子的性能?

(2) 融合方式为何可以提升性能?

(3) 如何更好地利用 MLU 的多核架构来提升性能?

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 600 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，包含服务器级（Server）、板卡级（Card）、芯片级（Chip）、处理器簇级（Cluster）和处理器核级（Core）。在服务器级，每个服务器系统包含若干个 CPU 构成的控制单元、本地 DDR 存储单元、以及通过 PCIe 总线互连的若干个 MLU 板卡构成的计算单元。在板卡级，每块 MLU 板卡包含本地存储（Global Memory，GDRAM）、作为计算和控制单元的处理器芯片。在芯片级，每个处理器芯片包含多个多处理器（Cluster）。在处理器簇级，每个 Cluster 包含 4 个计算核以及共享的片上存储（SRAM）。每个计算核中包含处理单元（NFU）、神经元存储（NRAM）和权重存储（WRAM）等。

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。为了详细地统计程序在 MLU 上的运行时间，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 ROIPoolingKernel 的执行时间的示例。

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

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

本节实验是使用 BANG 在 MLU 上实现向量加法计算并进行性能优化。从最简单的单核标量逐步修改成多核向量加流水优化版本，并对比测试性能。

在 MLU 上实现向量加法计算，需要编写主机端和设备端的异构混合代码，整个调优过程分为四个步骤：

(1) 完成一个基于标量加法的实现作为基准程序；

(2) 对基准程序进行向量化，从 SISD（Single Instruction Single Data，单指令单数据）到 SIMD（Single Instruction Multiple Data，单指令多数据）；

(3) 使用软件流水技术，将单核内部的访存和计算并行化处理；

(4) 充分利用 MLU 多核资源，实现多核并行处理。

5.2.5　实验步骤

5.2.5.1　单核标量实现

基准程序见图5.17实现了 float 类型的两组输入数据的加法，每个向量的长度是 10,000,000。

单核版本的任务规模计算函数 policyFunc 里设置成了单核任务，data_num 为每个核分到的数据量，计算过程为简单的循环遍历相加。

```
// file: 01_scalar_single_core.mlu

#include <bang.h>

#define ELEM_NUM 10 * 1000 * 1000

float src1_cpu[ELEM_NUM];
float src2_cpu[ELEM_NUM];
float dst_cpu[ELEM_NUM];

__mlu_entry__ void kernel(float *output, float *a, float *b, int data_num) {
  if (data_num == 0) {
    return;
  }
  for (int i = 0; i < data_num; i++) {
    // TODO: 请补充标量加法运算的表达式
    ____________________________________; // scalar add
  }
  return;
}

void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
  // TODO: 请补充单核任务的Kernel函数类型
  *func_type = ____________________; // single core
  dim->x = 1;
  dim->y = 1;
  dim->z = 1;
  return;
}
```

图 5.17　单核标量实现向量加法示例代码

```
int main() {
  CNRT_CHECK(cnrtSetDevice(0));
  cnrtNotifier_t st, et;
  CNRT_CHECK(cnrtNotifierCreate(&st));
  CNRT_CHECK(cnrtNotifierCreate(&et));
  cnrtQueue_t queue;
  // TODO: 请补充调用运行时接口创建异步Queue的函数调用
  CNRT_CHECK(_________________________);

  cnrtDim3_t dim;
  cnrtFunctionType_t func_type;
  policyFunction(&dim, &func_type);
  // 1.0f + 1.0f = 2.0f
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    src1_cpu[i] = 1.0f;
    src2_cpu[i] = 1.0f;
  }
  float* src1_mlu = NULL;
  float* src2_mlu = NULL;
  float* dst_mlu = NULL;
  CNRT_CHECK(cnrtMalloc((void **)&src1_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&src2_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&dst_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMemcpy(src1_mlu, src1_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtMemcpy(src2_mlu, src2_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtPlaceNotifier(st, queue));
  kernel<<<dim, func_type, queue>>>(dst_mlu, src1_mlu, src2_mlu, ELEM_NUM);
  CNRT_CHECK(cnrtPlaceNotifier(et, queue));
  // TODO: 请补充调用运行时接口同步Queue的操作
  CNRT_CHECK(_____________________);
  CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyDevToHost));
  float latency;
  // TODO: 请补充调用运行时接口统计硬件耗时的函数调用
  CNRT_CHECK(_____________________________________);
  CNRT_CHECK(cnrtFree(src1_mlu));
  CNRT_CHECK(cnrtFree(src2_mlu));
  CNRT_CHECK(cnrtFree(dst_mlu));
  CNRT_CHECK(cnrtQueueDestroy(queue));

  float diff = 0.0;
  float baseline = 2.0;
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    diff += fabs(dst_cpu[i] - baseline);
  }
  double theory_io = ELEM_NUM * 4.0 * 3.0; // bytes
  double theory_ops = ELEM_NUM * 4.0; // ops
  // ops_per_core/ns * core_num_per_cluter * cluster_num
  double peak_compute_force = 128 * 4 * 8;
  double io_bandwidth = 307.2; // bytes/ns
  double io_efficiency = theory_io / (latency * 1000) / io_bandwidth;
  double cp_efficiency = theory_ops / (latency * 1000) / peak_compute_force;
  printf("[MLU Hardware Time ]: %.3f us\n", latency);
  printf("[MLU IO Efficiency ]: %f\n", io_efficiency);
  printf("[MLU Compute Efficiency]: %f\n", cp_efficiency);
  printf("[MLU Diff Rate ]: %f\n", diff);
  printf(diff == 0 ? "PASSED\n" : "FAILED\n");
  return 0;
}
```

图 5.18　单核标量实现向量加法示例代码（续）

5.2.5.2 单核向量实现

接下来，将标量运算转换成向量计算，代码示例见图5.19，保持 policyFunc 不变，主要思想就是将单个元素的相加，变成了向量式的相加，拷贝和计算都是一条指令处理一个向量，并行性明显提高。受到片上空间容量限制，代码中使用循环做了数据切块处理，一次尽可能处理更多的元素。

5.2.5.3 单核向量异步流水实现

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5.22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。

5.2.5.4 多核向量异步流水实现

对于多核架构的 MLU 硬件，可以配置并行任务类型为 CNRT_FUNC_TYPE_UNION1 ，其含义为 MLU 硬件调度器以多个 Block 联合成一个 Union 为最小调度单元，使用 Union 任务的好处是可以启用 Cluster 内部的 SRAM 片上共享存储空间。假设多核实现配置为 Union1 任务类型，那么任务规模可以设置成 dim->x 为每个 Cluster 包含的 MLU Core 的个数，dim->y 为 Cluster 个数，dim->z 设置为 1，即可将所有 MLU Core 利用起来。具体策略函数和多核实现如下：

5.2.5.5 测试性能

测试环境性能配置参数：MLU370-X8@1.0GHz

ELEM_NUM=10*1000*1000 规模	耗时 (us)	提升幅度
单核标量实现	7876643.000	1x
单核向量实现	4536.000	1700x
单核向量异步流水实现	4284.000.000	1800x
多核向量异步流水实现	516.000	15000x

5.2.6 实验评估

本实验设定的评估标准如下：

- 60 分标准：在向量长度 ELEM_NUM=10*1000*1000 规模下，MLU 计算结果与 CPU 计算结果误差为 0，完成单核标量实现。
- 80 分标准：在向量长度 ELEM_NUM=10*1000*1000 规模下，MLU 计算结果与 CPU 计算结果误差为 0，完成单核向量实现，相比单核标量版本性能提升 1700 倍左右。
- 90 分标准：在向量长度 ELEM_NUM=10*1000*1000 规模下，MLU 计算结果与 CPU 计算结果误差为 0，完成单核向量异步流水实现，相比单核标量版本性能提升 1800 倍左右。

• 100 分标准：在向量长度 ELEM_NUM=10*1000*1000 规模下，MLU 计算结果与 CPU 计算结果误差为 0，完成多核向量异步流水实现，相比单核标量版本性能提升 15000 倍左右。

5.2.7 实验思考

(1) __bang_add 函数的底层指令是什么，为什么没有对齐约束？

(2) 在代码示例 **??**中，使用了 __sync_all_ipu()。这是为什么？

(3) 上述程序还有哪些瓶颈？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) 如果输入数据的字节数不是 NFU_ALIGN_SIZE 的整数倍，相比对齐到整数倍，会有性能下降么？为什么？

```c
// file: 02_vector_single_core.mlu
#include <bang.h>

#include <bang.h>

#define ELEM_NUM 10 * 1000 * 1000
#define MAX_NRAM_SIZE 655360
#define NFU_ALIGN_SIZE 128

__nram__ uint8_t nram_buffer[MAX_NRAM_SIZE];

float src1_cpu[ELEM_NUM];
float src2_cpu[ELEM_NUM];
float dst_cpu[ELEM_NUM];

__mlu_entry__ void kernel(float *output, float *a, float *b, int data_num) {
  if (data_num == 0) {
    return;
  }
  uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
  uint32_t data_ram_num =
    MAX_NRAM_SIZE / sizeof(float) / 2 / align_num * align_num;
  float *a_ram = (float *)nram_buffer;
  float *b_ram = (float *)a_ram + data_ram_num;
  // TODO: 请补充计算循环次数的表达式
  uint32_t loop_time = ________________________;
  // TODO: 请补充非data_ram_num对齐的元素个数计算表达式
  uint32_t rem_ram_num = ________________________;

  for (int i = 0; i < loop_time; i++) {
    // load
    __memcpy(a_ram, a + i * data_ram_num,
             data_ram_num * sizeof(float), GDRAM2NRAM);
    __memcpy(b_ram, b + i * data_ram_num,
             data_ram_num * sizeof(float), GDRAM2NRAM);
    // TODO: 请补充BANG内置向量加法函数签名
    __________(a_ram, a_ram, b_ram, data_ram_num); // vector add
    // store
    __memcpy(output + i * data_ram_num, a_ram,
             data_ram_num * sizeof(float), NRAM2GDRAM);
  }
  if (rem_ram_num != 0) {
    uint32_t rem_align_num =
      (rem_ram_num + align_num - 1) / align_num * align_num;
    // load
    __memcpy(a_ram, a + loop_time * data_ram_num,
             rem_ram_num * sizeof(float), GDRAM2NRAM);
    __memcpy(b_ram, b + loop_time * data_ram_num,
             rem_ram_num * sizeof(float), GDRAM2NRAM);
    // compute
    __bang_add(a_ram, a_ram, b_ram, rem_align_num);
    // store
    __memcpy(output + loop_time * data_ram_num, a_ram,
            rem_ram_num * sizeof(float), NRAM2GDRAM);
  }
  return;
}
```

图 5.19　单核向量实现向量加法示例代码

```
void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
  *func_type = CNRT_FUNC_TYPE_BLOCK; // single core
  dim->x = 1;
  dim->y = 1;
  dim->z = 1;
  return;
}

int main() {
  CNRT_CHECK(cnrtSetDevice(0));
  cnrtNotifier_t st, et;
  CNRT_CHECK(cnrtNotifierCreate(&st));
  CNRT_CHECK(cnrtNotifierCreate(&et));
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));

  cnrtDim3_t dim;
  cnrtFunctionType_t func_type;
  policyFunction(&dim, &func_type);

  // 1.0f + 1.0f = 2.0f
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    src1_cpu[i] = 1.0f;
    src2_cpu[i] = 1.0f;
  }
  float* src1_mlu = NULL;
  float* src2_mlu = NULL;
  float* dst_mlu = NULL;
  CNRT_CHECK(cnrtMalloc((void **)&src1_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&src2_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&dst_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMemcpy(src1_mlu, src1_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtMemcpy(src2_mlu, src2_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtPlaceNotifier(st, queue));
  kernel<<<dim, func_type, queue>>>(dst_mlu, src1_mlu, src2_mlu, ELEM_NUM);
  CNRT_CHECK(cnrtPlaceNotifier(et, queue));
  CNRT_CHECK(cnrtQueueSync(queue));
  CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyDevToHost));
  float latency;
  CNRT_CHECK(cnrtNotifierDuration(st, et, &latency));
  CNRT_CHECK(cnrtFree(src1_mlu));
  CNRT_CHECK(cnrtFree(src2_mlu));
  CNRT_CHECK(cnrtFree(dst_mlu));
  CNRT_CHECK(cnrtQueueDestroy(queue));
```

图 5.20　单核向量实现向量加法示例代码（续 1）

```
  float diff = 0.0;
  float baseline = 2.0;
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    diff += fabs(dst_cpu[i] - baseline);
  }
  double theory_io = ELEM_NUM * 4.0 * 3.0; // bytes
  double theory_ops = ELEM_NUM * 4.0; // ops
  // ops_per_core/ns * core_num_per_cluter * cluster_num
  double peak_compute_force = 128 * 4 * 8;
  double io_bandwidth = 307.2; // bytes/ns
  double io_efficiency = theory_io / (latency * 1000) / io_bandwidth;
  double cp_efficiency = theory_ops / (latency * 1000) / peak_compute_force;
  printf("[MLU Hardware Time ]: %.3f us\n", latency);
  printf("[MLU IO Efficiency ]: %f\n", io_efficiency);
  printf("[MLU Compute Efficiency]: %f\n", cp_efficiency);
  printf("[MLU Diff Rate ]: %f\n", diff);
  printf(diff == 0 ? "PASSED\n" : "FAILED\n");

  return 0;
}
```

图 5.21　单核向量实现向量加法示例代码（续 2）

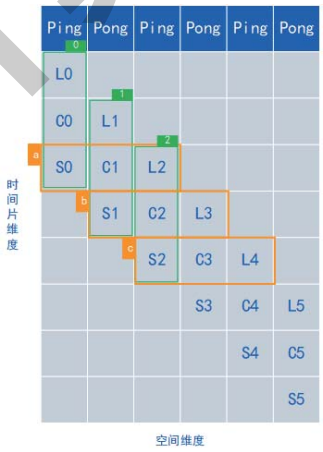

图 5.22　异步流水示意图

```
// file: 03_vector_single_core_pipeline.mlu
#include <bang.h>

#define ELEM_NUM 10 * 1000 * 1000
#define MAX_NRAM_SIZE 655360
#define NFU_ALIGN_SIZE 128

__nram__ uint8_t nram_buffer[MAX_NRAM_SIZE];

float src1_cpu[ELEM_NUM];
float src2_cpu[ELEM_NUM];
float dst_cpu[ELEM_NUM];

// load in pipeline
__mlu_func__ void L(float *a_ram, float *a, float *b_ram,
                    float *b, int data_ram_num, int i) {
  // TODO: 请补充数据拷贝方向
  mluMemcpyDirection_t direction = __________;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(a_ram + offset, a + i * data_ram_num,
                 data_ram_num * sizeof(float), direction);
  __memcpy_async(b_ram + offset, b + i * data_ram_num,
                 data_ram_num * sizeof(float), direction);
}

// compute in pipeline
__mlu_func__ void C(float *a_ram, float *b_ram, int data_ram_num, int i) {
  int offset = i % 2 * data_ram_num * 2;
  // TODO: 请补充调用BANG内置向量加法函数的参数
  __bang_add(_____________________________________________________________);
}

// store in pipeline
__mlu_func__ void S(float *output, float *a_ram, int data_ram_num, int i) {
  // TODO: 请补充数据拷贝方向
  mluMemcpyDirection_t direction = __________;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(output + i * data_ram_num, a_ram + offset,
  data_ram_num * sizeof(float), direction);
}
```

图 5.23　单核向量异步流水实现示例代码

```
// load in pipeline
__mlu_func__ void L_rem(float *a_ram, float *a, float *b_ram, float *b,
                        int data_ram_num, int rem_ram_num, int loop_time,
                        int i) {
  // TODO: 请补充数据拷贝方向
  mluMemcpyDirection_t direction = __________;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(a_ram + offset, a + loop_time * data_ram_num,
                 rem_ram_num * sizeof(float), direction);
  __memcpy_async(b_ram + offset, b + loop_time * data_ram_num,
                 rem_ram_num * sizeof(float), direction);
}

// compute in pipeline
__mlu_func__ void C_rem(float *a_ram, float *b_ram,
                        int data_ram_num, int rem_align_num, int i) {
  int offset = i % 2 * data_ram_num * 2;
  __bang_add(a_ram + offset, a_ram + offset, b_ram + offset, rem_align_num);
}

// store in pipeline
__mlu_func__ void S_rem(float *output, float *a_ram, int data_ram_num,
                        int rem_ram_num, int loop_time, int i) {
  mluMemcpyDirection_t direction = NRAM2GDRAM;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(output + loop_time * data_ram_num, a_ram + offset,
                 rem_ram_num * sizeof(float), direction);
}
```

图 5.24　单核向量异步流水实现示例代码（续 1）

```
__mlu_entry__ void kernel(float *output, float *a, float *b, int data_num) {
  if (data_num == 0) {
    return;
  }
  // ping: a(out), b || pong: a(out), b
  uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
  uint32_t data_ram_num =
    MAX_NRAM_SIZE / sizeof(float) / 4 / align_num * align_num;
  float *a_ram = (float *)nram_buffer;
  float *b_ram = a_ram + data_ram_num;

  uint32_t loop_time = data_num / data_ram_num;
  uint32_t rem_ram_num = data_num % data_ram_num;
  int rem_num = 0;
  uint32_t rem_align_num =
    (rem_ram_num + align_num - 1) / align_num * align_num;
  if (rem_ram_num != 0) {
    rem_num = 1;
  }
  for (int i = 0; i < loop_time + 2 + rem_num; i++) {
    if (i >= 2) {
      // S(i - 2)
      if (i < loop_time + 2 + rem_num - 1 || rem_num == 0) {
        S(output, a_ram, data_ram_num, i - 2);
      } else if (rem_num == 1) {
       S_rem(output, a_ram, data_ram_num, rem_ram_num, loop_time, i - 2);
      }
    }
    if (i >= 1 && i < loop_time + 1 + rem_num) {
    // C(i - 1)
      if (i < loop_time + 1 + rem_num - 1 || rem_num == 0) {
        C(a_ram, b_ram, data_ram_num, i - 1);
      } else if (rem_num == 1) {
        C_rem(a_ram, b_ram, data_ram_num, rem_align_num, i - 1);
      }
    }
    if (i < loop_time + rem_num) {
    // L(i)
      if (i < loop_time + rem_num - 1 || rem_num == 0) {
        L(a_ram, a, b_ram, b, data_ram_num, i);
      } else if (rem_num == 1) {
        L_rem(a_ram, a, b_ram, b, data_ram_num, rem_ram_num, loop_time, i);
      }
    }
    __sync_all_ipu();
  }
  return;
}
```

图 5.25　单核向量异步流水实现示例代码（续 2）

```
void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
  *func_type = CNRT_FUNC_TYPE_BLOCK; // single core
  dim->x = 1;
  dim->y = 1;
  dim->z = 1;
  return;
}

int main() {
  CNRT_CHECK(cnrtSetDevice(0));
  cnrtNotifier_t st, et;
  CNRT_CHECK(cnrtNotifierCreate(&st));
  CNRT_CHECK(cnrtNotifierCreate(&et));
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));

  cnrtDim3_t dim;
  cnrtFunctionType_t func_type;
  policyFunction(&dim, &func_type);

  // 1.0f + 1.0f = 2.0f
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    src1_cpu[i] = 1.0f;
    src2_cpu[i] = 1.0f;
  }
  float* src1_mlu = NULL;
  float* src2_mlu = NULL;
  float* dst_mlu = NULL;
  CNRT_CHECK(cnrtMalloc((void **)&src1_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&src2_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&dst_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMemcpy(src1_mlu, src1_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtMemcpy(src2_mlu, src2_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtPlaceNotifier(st, queue));
  kernel<<<dim, func_type, queue>>>(dst_mlu, src1_mlu, src2_mlu, ELEM_NUM);
  CNRT_CHECK(cnrtPlaceNotifier(et, queue));
  CNRT_CHECK(cnrtQueueSync(queue));
  CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyDevToHost));
  float latency;
  CNRT_CHECK(cnrtNotifierDuration(st, et, &latency));
  CNRT_CHECK(cnrtFree(src1_mlu));
  CNRT_CHECK(cnrtFree(src2_mlu));
  CNRT_CHECK(cnrtFree(dst_mlu));
  CNRT_CHECK(cnrtQueueDestroy(queue));
```

图 5.26　单核向量异步流水实现示例代码（续 3）

```
  float diff = 0.0;
  float baseline = 2.0;
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    diff += fabs(dst_cpu[i] - baseline);
  }
  double theory_io = ELEM_NUM * 4.0 * 3.0; // bytes
  double theory_ops = ELEM_NUM * 4.0; // ops
  // ops_per_core/ns * core_num_per_cluter * cluster_num
  double peak_compute_force = 128 * 4 * 8;
  double io_bandwidth = 307.2; // bytes/ns
  double io_efficiency = theory_io / (latency * 1000) / io_bandwidth;
  double cp_efficiency = theory_ops / (latency * 1000) / peak_compute_force;
  printf("[MLU Hardware Time ]: %.3f us\n", latency);
  printf("[MLU IO Efficiency ]: %f\n", io_efficiency);
  printf("[MLU Compute Efficiency]: %f\n", cp_efficiency);
  printf("[MLU Diff Rate ]: %f\n", diff);
  printf(diff == 0 ? "PASSED\n" : "FAILED\n");

  return 0;
}
```

图 5.27　单核向量异步流水实现示例代码（续 4）

```
// file: 04_vector_multi_core_pipeline.mlu
#include <bang.h>

#define ELEM_NUM 10 * 1000 * 1000
#define MAX_NRAM_SIZE 655360
#define NFU_ALIGN_SIZE 128

__nram__ uint8_t nram_buffer[MAX_NRAM_SIZE];

float src1_cpu[ELEM_NUM];
float src2_cpu[ELEM_NUM];
float dst_cpu[ELEM_NUM];

// load in pipeline
__mlu_func__ void L(float *a_ram, float *a, float *b_ram,
                    float *b, int data_ram_num, int i) {
  mluMemcpyDirection_t direction = GDRAM2NRAM;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(a_ram + offset, a + i * data_ram_num,
                 data_ram_num * sizeof(float), direction);
  __memcpy_async(b_ram + offset, b + i * data_ram_num,
                 data_ram_num * sizeof(float), direction);
}

// compute in pipeline
__mlu_func__ void C(float *a_ram, float *b_ram, int data_ram_num, int i) {
  int offset = i % 2 * data_ram_num * 2;
  __bang_add(a_ram + offset, a_ram + offset, b_ram + offset, data_ram_num);
}

// store in pipeline
__mlu_func__ void S(float *output, float *a_ram, int data_ram_num, int i) {
  mluMemcpyDirection_t direction = NRAM2GDRAM;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(output + i * data_ram_num, a_ram + offset,
  data_ram_num * sizeof(float), direction);
}
```

图 5.28　多核向量异步流水实现代码示例

```
// load in pipeline
__mlu_func__ void L_rem(float *a_ram, float *a, float *b_ram, float *b,
                        int data_ram_num, int rem_ram_num, int loop_time,
                        int i) {
  mluMemcpyDirection_t direction = GDRAM2NRAM;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(a_ram + offset, a + loop_time * data_ram_num,
                 rem_ram_num * sizeof(float), direction);
  __memcpy_async(b_ram + offset, b + loop_time * data_ram_num,
                 rem_ram_num * sizeof(float), direction);
}

// compute in pipeline
__mlu_func__ void C_rem(float *a_ram, float *b_ram,
                        int data_ram_num, int rem_align_num, int i) {
  int offset = i % 2 * data_ram_num * 2;
  __bang_add(a_ram + offset, a_ram + offset, b_ram + offset, rem_align_num);
}

// store in pipeline
__mlu_func__ void S_rem(float *output, float *a_ram, int data_ram_num,
                        int rem_ram_num, int loop_time, int i) {
  mluMemcpyDirection_t direction = NRAM2GDRAM;
  int offset = i % 2 * data_ram_num * 2;
  __memcpy_async(output + loop_time * data_ram_num, a_ram + offset,
  rem_ram_num * sizeof(float), direction);
}
```

图 5.29　多核向量异步流水实现代码示例（续 1）

```
__mlu_func__ void add(float *output, float *a, float *b, int data_num) {
  if (data_num == 0) {
    return;
  }
  // ping: a(out), b || pong: a(out), b
  uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
  uint32_t data_ram_num =
    MAX_NRAM_SIZE / sizeof(float) / 4 / align_num * align_num;
  float *a_ram = (float *)nram_buffer;
  float *b_ram = a_ram + data_ram_num;
  uint32_t loop_time = data_num / data_ram_num;
  uint32_t rem_ram_num = data_num % data_ram_num;
  int rem_num = 0;
  uint32_t rem_align_num =
    (rem_ram_num + align_num - 1) / align_num * align_num;
  if (rem_ram_num != 0) {
    rem_num = 1;
  }

  for (int i = 0; i < loop_time + 2 + rem_num; i++) {
    if (i >= 2) {
      // S(i - 2)
      if (i < loop_time + 2 + rem_num - 1 || rem_num == 0) {
        S(output, a_ram, data_ram_num, i - 2);
      } else if (rem_num == 1) {
       S_rem(output, a_ram, data_ram_num, rem_ram_num, loop_time, i - 2);
      }
    }
    if (i >= 1 && i < loop_time + 1 + rem_num) {
    // C(i - 1)
      if (i < loop_time + 1 + rem_num - 1 || rem_num == 0) {
        C(a_ram, b_ram, data_ram_num, i - 1);
      } else if (rem_num == 1) {
        C_rem(a_ram, b_ram, data_ram_num, rem_align_num, i - 1);
      }
    }
    if (i < loop_time + rem_num) {
    // L(i)
      if (i < loop_time + rem_num - 1 || rem_num == 0) {
        L(a_ram, a, b_ram, b, data_ram_num, i);
      } else if (rem_num == 1) {
        L_rem(a_ram, a, b_ram, b, data_ram_num, rem_ram_num, loop_time, i);
      }
    }
    __sync_all_ipu();
  }
  return;
}
```

图 5.30　多核向量异步流水实现代码示例（续 2）

```
__mlu_entry__ void kernel(float *output,
                          const float *a,
                          const float *b,
                          const int data_num) {
  // No need mpu core, return.
  if (coreId == 0x80) {
    return;
  }

  // TODO: 请补充BANG内置变量，获取并行任务总规模
  uint32_t task_dim = taskDim;
  // TODO: 请补充BANG内置变量，获取当前并行任务的索引
  uint32_t task_id = taskId;
  // TODO: 请补充计算每个核的计算数据量的表达式
  uint32_t data_per_core = ____________________;
  // TODO: 请补充当总数据量不能被并行任务总规模整除时，计算最后一个核数据量的表达式
  uint32_t data_last_core = ____________________________________;

  float *a_fix = (float *)a + task_id * data_per_core;
  float *b_fix = (float *)b + task_id * data_per_core;
  float *output_fix = (float *)output + task_id * data_per_core;

  if (task_id != task_dim - 1) {
    add(output_fix, a_fix, b_fix, data_per_core);
  } else {
    add(output_fix, a_fix, b_fix, data_last_core);
  }
}

void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
  // TODO: 请补充多核Kernel任务的类型
  *func_type = ____________________;
  dim->x = 4;
  dim->y = 8;
  dim->z = 1;
  return;
}

int main() {
  CNRT_CHECK(cnrtSetDevice(0));
  cnrtNotifier_t st, et;
  CNRT_CHECK(cnrtNotifierCreate(&st));
  CNRT_CHECK(cnrtNotifierCreate(&et));
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));

  cnrtDim3_t dim;
  cnrtFunctionType_t func_type;
  policyFunction(&dim, &func_type);
```

图 5.31　多核向量异步流水实现代码示例（续 3）

```
  // 1.0f + 1.0f = 2.0f
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    src1_cpu[i] = 1.0f;
    src2_cpu[i] = 1.0f;
  }
  float* src1_mlu = NULL;
  float* src2_mlu = NULL;
  float* dst_mlu = NULL;
  CNRT_CHECK(cnrtMalloc((void **)&src1_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&src2_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMalloc((void **)&dst_mlu, ELEM_NUM * sizeof(float)));
  CNRT_CHECK(cnrtMemcpy(src1_mlu, src1_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtMemcpy(src2_mlu, src2_cpu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyHostToDev));
  CNRT_CHECK(cnrtPlaceNotifier(st, queue));
  kernel<<<dim, func_type, queue>>>(dst_mlu, src1_mlu, src2_mlu, ELEM_NUM);
  CNRT_CHECK(cnrtPlaceNotifier(et, queue));
  CNRT_CHECK(cnrtQueueSync(queue));
  CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                        cnrtMemcpyDevToHost));
  float latency;
  CNRT_CHECK(cnrtNotifierDuration(st, et, &latency));
  CNRT_CHECK(cnrtFree(src1_mlu));
  CNRT_CHECK(cnrtFree(src2_mlu));
  CNRT_CHECK(cnrtFree(dst_mlu));
  CNRT_CHECK(cnrtQueueDestroy(queue));

  float diff = 0.0;
  float baseline = 2.0;
  for (unsigned i = 0; i < ELEM_NUM; ++i) {
    diff += fabs(dst_cpu[i] - baseline);
  }
  double theory_io = ELEM_NUM * 4.0 * 3.0; // bytes
  double theory_ops = ELEM_NUM * 4.0; // ops
  // ops_per_core/ns * core_num_per_cluter * cluster_num
  double peak_compute_force = 128 * 4 * 8;
  double io_bandwidth = 307.2; // bytes/ns
  double io_efficiency = theory_io / (latency * 1000) / io_bandwidth;
  double cp_efficiency = theory_ops / (latency * 1000) / peak_compute_force;
  printf("[MLU Hardware Time ]: %.3f us\n", latency);
  printf("[MLU IO Efficiency ]: %f\n", io_efficiency);
  printf("[MLU Compute Efficiency]: %f\n", cp_efficiency);
  printf("[MLU Diff Rate ]: %f\n", diff);
  printf(diff == 0 ? "PASSED\n" : "FAILED\n");

  return 0;
}
```

图 5.32　多核向量异步流水实现代码示例（续 4）