案例名称：　智能操作系统实践案例：国产软硬件大模型基础软件适配与优化

专业学位类别：　电子信息

适用课程：　物联网技术；大模型；适配；优化

作者姓名：　姚建国

工作单位：　10248 上海交通大学

案例正文：

智能操作系统实践案例：国产软硬件大模型基础软件适配与优化

摘　要：本案例描述了面向国产软硬芯的大模型基础软件适配与优化，重点解决在硬件资源资源受限场景下大模型高效推理，将综合培养学生对工程问题的发现、识别、分析和解决等能力。本案例引导学生将大模型推理框架适配至国产 AI 芯片与基础软硬件环境中，为我国软硬芯生态积蓄生态力量

关键词：操作系统、大模型、适配、优化

1.工程项目背景及现状

1）神经网络处理单元（NPU）

i. NPU概述

神经网络处理单元（NPU）是一种专用计算硬件，专为计算神经网络中的大量矩阵运算而设计。采用“数据驱动并行”，加速机器学习任务的同时降低功耗，成为多种领域AI应用的关键硬件。

随着机器学习、人工智能的发展，全球多家厂商提出了自己的NPU产品：Google自2011年起开始研发TPU（张量处理单元）；Intel、AMD、高通也分别在各自发布的处理器中集成了内置NPU；华为以自研昇腾NPU为基础，发布昇腾全栈AI软硬件平台，在国产AI芯片领域迈出了重要一步。

ii. 华为昇腾NPU

华为昇腾NPU针对密集矩阵运算场景，对硬件架构进行了创新设计。在昇腾NPU中，张量算子运行在AI Core上，AI Core包含三种主要单元：计算单元、存储单元、控制单元，三者相互配合，共同为神经网络计算提供加速功能。

- 计算单元

计算单元包括三种基础单元：矩阵计算单元、向量计算单元、标量计算单元。标量计算单元架构类似于CPU，提供标量数据运算和程序流程控制，拥有ALU，可以发射指令；向量计算单元的计算方式类似于传统CPU的单指令多数据（SIMD）指令；矩阵计算单元一次执行可以完成一次矩阵乘法。

- 存储单元

类似于CPU，NPU中同样存在内存阶层（Memory Hierarchy），AI Core中的存储单元为高速内部存储。计算时，数据从低速外部存储（HBM等）搬运到内部存储中，供计算单元取值。为了配合AI Core中的数据传输和搬运，AI Core中还包含MTE（Memory Transfer Engine，存储转换引擎）搬运单元，在搬运过程中可执行随路数据格式/类型转换。

- 控制单元

控制单元用于指令的缓存与发射。多条指令从主存进入指令缓存，对于标量指令，直接交给标量单元，其他指令则会进入相应队列，并分配给相应单元执行。不同指令执行队列之间可以并行执行，提升整体执行效率。对于并行执行过程中的数据依赖，可通过事件同步模块插入同步指令来控制流水线的同步。

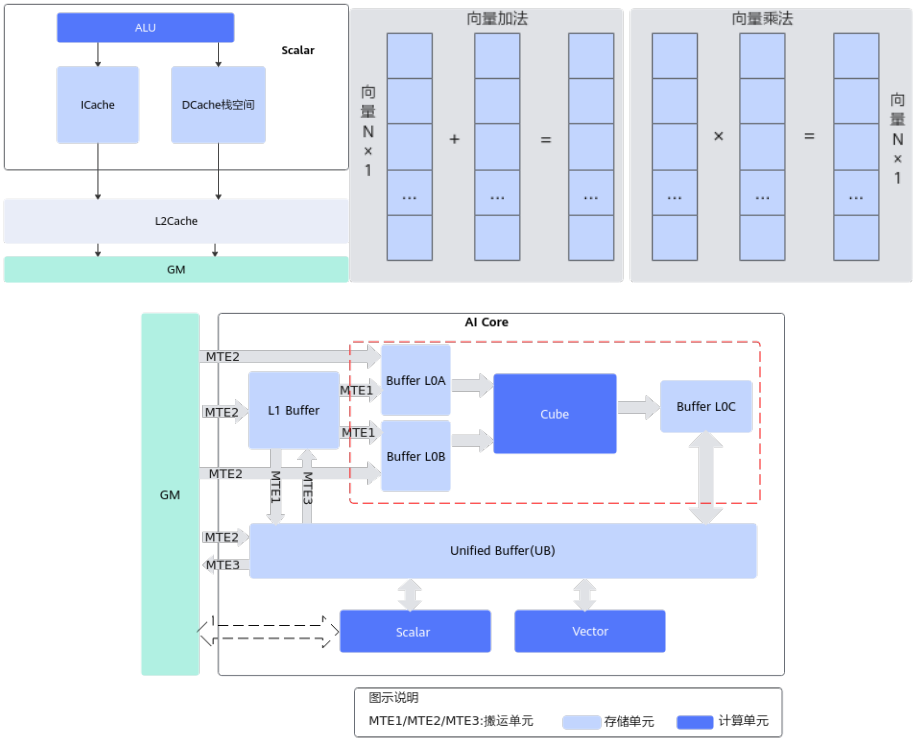

图1 昇腾NPU计算单元架构

iii. 昇腾异构计算架构CANN

异构计算架构CANN是华为针对AI场景推出的异构计算架构，是上层应用（如深度学习训练，推理框架）和底层NPU硬件的之间的中间件，封装了图开发，算子开发，应用开发等API，为开发者在昇腾NPU开发AI应用提供极大的便利。

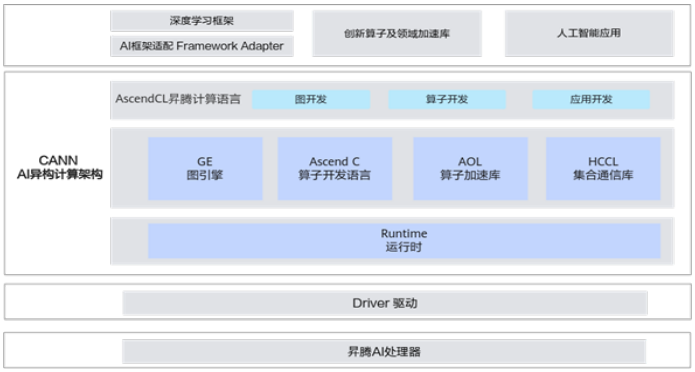

图2 CANN异构计算架构

2）llama.cpp推理框架

i. llama.cpp简介

llama.cpp 是一个主要用 C++ 编写的开源软件库，可以对各种大型语言模型（如 Llama）进行推理。它与 ggml 库（一个通用张量库）共同开发。

llama.cpp 由 Georgi Gerganov 于 2023 年 3 月开始开发，作为纯 C/C++ 实现的 Llama 推理代码，没有任何依赖。这提高了在没有 GPU 或其他专用硬件的计算机上的性能。截至 2024 年 7 月，它在 GitHub 上获得了 61k star。llama.cpp 吸引了没有专用硬件的用户，因为它可以仅在 CPU 上运行，包括在 Android 设备上。

ii. llama.cpp特点

- CPU推理

Justine Tunney 创建的 llamafile 是一个开源工具，将 llama.cpp 与模型打包成一个可执行文件。Tunney 等人引入了新的优化矩阵乘法内核，适用于 x86 和 ARM CPU，提升了 FP16 和 8 位量化数据类型的提示评估性能。llama.cpp 利用了多种 CPU 扩展进行优化：X86-64 的 AVX、AVX2 和 AVX-512，以及 ARM 的 Neon。Apple 芯片是该项目的重要目标。

- 多后端支持

llama.cpp 最初只能在 CPU 上运行，但现在可以通过多个不同的后端在 GPU 上运行，包括 Vulkan 和 SYCL。这些后端构成了 GGML 张量库，被前端特定模型的 llama.cpp 代码所使用。对于大于总VRAM容量的大规模模型，该库还支持如CPU+GPU混合推理模式进行部分加速。

- 模型量化

llama.cpp 支持预先模型量化，而不是即时量化。支持2 位到 8 位量化整数类型。

2.工程的实施过程

本案例将基于高速大型语言模型推理引擎进行工程实践，技术路线将包括以下两步：

1) 学生对高速大型语言模型推理框架llama.cpp源代码进行修改，以兼容国产AI 芯片（昇腾 NPU），充分利用NPU的计算特性和管理机制：具体包括适配 NPU的设备管理，内存管理机制，以及算子在NPU设备上的移植，确保代码能够被正确编译。

2) 对高速大型语言模型推理引擎进行性能测试和验证，确保在 NPU上的推理速度和精度达到预期的水平，同时保证系统的稳定性与可靠性。在此过程中，需要选择并集成合适的模型，研究多种不同算力卸载方案，以适应 NPU 的资源限制，同时尽可能增加NPU的资源利用率。

具体实施过程如下：

1）高速大型语言模型推理引擎源代码修改与适配

首先，针对高速大型语言模型推理引擎的源代码进行修改与适配是本工程的首要任务。这一阶段的具体步骤如下：

i. 详细研究昇腾NPU的技术文档和开发指南，理解其系统配置，运行时管理机制以及与其他硬件组件的交互方式：CANN提供了一系列的系统结构，包括程序启动时设备初始化接口，设备应用运行时的Device管理接口，Context上下文管理接口，内存管理接口，执行计算任务时所需的Stream流管理接口等，与目前广泛使用的Nvidia GPU提供的Cuda接口类似。

阅读llama.cpp对于CPU，GPU设备的backend实现模块，明确其接口的实现，并根据任务需求确定需要移植的部分。如下图所示，llama.cpp的的Cuda backend部分实现了一系列的以C语言风格声明的接口，例如ggml_backend_cuda_malloc，ggml_backend_cuda_free，ggml_backend_cpy_tensor_async等内存接口的封装，内部调用CudaMalloc，CudaFree等GPU提供的设备提供的内存接口；其他接口包括Cuda内存池的管理，ggml_context的Cuda后端实现，compute_forward计算接口，这些属于较为上层的封装，便于与应用层更好地解耦。

```
static ggml_backend_i ggml_backend_cuda_interface = {
    /* .get_name                = */ ggml_backend_cuda_name,
    /* .free                    = */ ggml_backend_cuda_free,
    /* .get_default_buffer_type = */ ggml_backend_cuda_get_default_buffer_type,
    /* .set_tensor_async        = */ ggml_backend_cuda_set_tensor_async,
    /* .get_tensor_async        = */ ggml_backend_cuda_get_tensor_async,
    /* .cpy_tensor_async        = */ ggml_backend_cuda_cpy_tensor_async,
    /* .synchronize             = */ ggml_backend_cuda_synchronize,
    /* .graph_plan_create       = */ NULL,
    /* .graph_plan_free         = */ NULL,
    /* .graph_plan_update       = */ NULL,
    /* .graph_plan_compute      = */ NULL,
    /* .graph_compute           = */ ggml_backend_cuda_graph_compute,
    /* .supports_op             = */ ggml_backend_cuda_supports_op,
    /* .supports_buft           = */ ggml_backend_cuda_supports_buft,
    /* .offload_op              = */ ggml_backend_cuda_offload_op,
    /* .event_new               = */ ggml_backend_cuda_event_new,
    /* .event_free              = */ ggml_backend_cuda_event_free,
    /* .event_record            = */ ggml_backend_cuda_event_record,
    /* .event_wait              = */ ggml_backend_cuda_event_wait,
    /* .event_synchronize       = */ ggml_backend_cuda_event_synchronize,
};
```

图3 llama.cpp中的Cuda接口声明

在梳理NPU的设备接口和llama.cpp在GPU设备上的后端实现的基础上，将与之对应的昇腾NPU后端实现集成在ggml-ascend.cpp文件：首先实现log部分，该部分为通用接口，为调试提供便利。

```
static void ggml_ascend_log(enum ggml_log_level level, const char * format, ...) {
    if (ggml_ascend_log_callback != NULL) {
        va_list args;
        va_start(args, format);
        char buffer[128];
        int len = vsnprintf(buffer, 128, format, args);
        if (len < 128) {
            ggml_ascend_log_callback(level, buffer, ggml_ascend_log_user_data);
        } else {
            std::vector<char> buffer2(len + 1);
            va_end(args);
            va_start(args, format);
            vsnprintf(&buffer2[0], buffer2.size(), format, args);
            ggml_ascend_log_callback(level, buffer2.data(), ggml_ascend_log_user_data);
        }
        va_end(args);
    }
}
```

图4 ascend中log接口实现(部分)

然后是包装设备管理和内存管理接口，包括set/getDevice，aclMalloc/Free；实现了一个用于管理中间计算结果的内存池，该内存池在程序开始时分配一整块连续的内存，NPU算子产生的中间结果直接放在内存池，内存池中的地址可以通过指针运算直接获取，内存池资源耗尽后，会再次申请一块内存进行扩容；释放内存时，修改内存池元数据，将指定内存标记为已释放。

```cpp
int ggml_ascend_get_device() {
    int id;
    // todo check: fixed
    auto ret = aclrtGetDevice(&id);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("acl get device failed at [ggml_ascend_get_device]
    return id;
}

static aclError ggml_ascend_device_malloc(void ** ptr, size_t size, int device) {
    ggml_ascend_set_device(device);

    return aclrtMalloc(ptr, size, ACL_MEM_MALLOC_HUGE_FIRST);
}

//buffer pool
struct ggml_ascend_pool_leg : public ggml_ascend_pool {
    static const int MAX_BUFFERS = 256;

    int device;
    struct ggml_ascend_buffer {
        void *ptr;
        size_t size = 0;
    };

    ggml_ascend_buffer buffer_pool[MAX_BUFFERS] = {};
    size_t pool_size = 0;

    explicit ggml_ascend_pool_leg(int device) : device(device) {

    }
```

图5 ascend运行时接口封装与内存池定义

实现后端buffer的接口，buffer接口负责多个tensor的存取，对于不同的后端，buffer有着不同的名称。因此，项目实现了get_name接口：用于获取名称；free_buffer接口：将buffer中的context进行释放，回收资源；get_base：获取buffer中的设备指针，一般指向内存的起始位置；init/get/set/cpy/clear tensor：这些是buffer中tensor层面上的接口，包含初始化，从NPU设备中获取tensor，将host侧tensor放入NPU，在NPU侧拷贝tensor，将tensor内存全部置零的操作。接着是buffer_type的后端接口，包括设备侧的接口和host主机侧的接口，其负责的功能有分配buffer，获取对齐量，获取需要分配空间大小。

```cpp
static ggml_backend_buffer_i ggml_backend_ascend_buffer_interface = {
    /* .get_name    = */ ggml_backend_ascend_buffer_get_name,
    /* .free_buffer = */ ggml_backend_ascend_buffer_free_buffer,
    /* .get_base    = */ ggml_backend_ascend_buffer_get_base,
    /* .init_tensor = */ ggml_backend_ascend_buffer_init_tensor,
    /* .set_tensor  = */ ggml_backend_ascend_buffer_set_tensor,
    /* .get_tensor  = */ ggml_backend_ascend_buffer_get_tensor,
    /* .cpy_tensor  = */ ggml_backend_ascend_buffer_cpy_tensor,
    /* .clear       = */ ggml_backend_ascend_buffer_clear,
    /* .reset       = */ NULL,
};
```

图6 ascend实现的后端buffer接口

最后是后端接口的核心模块：ascend_backend。通过CANN内存管理接口实现tensor的相关操作；实现计算函数，根据算子类型，检查模型是否被支持，目前支持llama2和bamboo模型的算子；然后调用自行实现的ASCEND算子接口；实现上层的计算接口，从计算图中取

出计算节点，对src和dst的数据类型进行检查，对于Float16和Float32数据类型，可以直接调用NPU提供的算子进行运算，对于8bit量化类型，会先调用反量化接口，然后，调用计算函数进行计算。

```
GGML_CALL static enum ggml_status ggml_backend_ascend_graph_compute(ggml_backend_t backend, ggml_cgraph *
    auto ctx = (ggml_backend_ascend_context *)backend->context;
    ggml_ascend_set_device(ctx->device);

    bool use_graph = false;
    bool graph_update_required = false;
    //GGML_UNUSED(graph_update_required);

    bool graph_evaluated_or_captured = false;

    while (!graph_evaluated_or_captured) {
        if (!use_graph || graph_update_required) {
            for (int i = 0; i < cgraph->n_nodes; i++) {
                ggml_tensor * node = cgraph->nodes[i];
                                                    enum ggml_op::GGML_OP_RESHAPE = 30
                if (ggml_is_empty(node) || node->op == GGML_OP_RESHAPE || node->op == GGML_OP_TRANSPOSE ||
                    continue;
                }
#ifndef NDEBUG
                assert(node->buffer->buft == ggml_backend_ascend_buffer_type(ctx->device));
                for (int j = 0; j < GGML_MAX_SRC; j++) {
                    if (node->src[j] != nullptr) {
                        assert(node->src[j]->buffer->buft == ggml_backend_ascend_buffer_type(ctx->device)
                    }
                }
#endif
                // TODO implement: marked
                bool ok = ggml_ascend_compute_forward(*ctx, node);
```

图7 ascend实现的计算接口

ii. 其次，进一步研究昇腾NPU的计算特性，算子开发方法和AOL算子加速库的API调用方法，参照llama.cpp项目结构在ggml-Ascend文件夹下开发基于华为CANN库的算子。在开发过程中，具体工程的实施依赖于两个方面：对llama.cpp入参的重整，以及根据llama.cpp中cuda算子和cpu算子的实现逻辑，组合复用CANN库提供的算子接口以达成和ggml算子相同语义的实现。

在传入参数的重整方面，以数据类型为例，华为提供的aclnn算子大多支持互推导关系，即当一个API输入的aclTensor数据类型不一致时，API内部会推导出一个数据类型，将输入数据转换成该数据类型进行计算，因而团队提供了ggml_to_acl_map和ggml_type_size_t，以将llama.cpp提供的ggml_tesnor支持的数据类型和aclnn算子API推导的数据类型进行转换，以及在NPU上通过aclrtMalloc接口申请device侧内存。

```cpp
aclDataType ggml_to_acl_map[GGML_TYPE_COUNT] = {
    ACL_FLOAT,          // GGML_TYPE_F32
    ACL_FLOAT16,        // GGML_TYPE_F16
    ACL_DT_UNDEFINED, // GGML_TYPE_Q4_0
    ACL_DT_UNDEFINED, // GGML_TYPE_Q4_1
    ACL_DT_UNDEFINED, // GGML_TYPE_Q4_2 (removed)
    ACL_DT_UNDEFINED, // GGML_TYPE_Q4_3 (removed)
    ACL_DT_UNDEFINED, // GGML_TYPE_Q5_0
    ACL_DT_UNDEFINED, // GGML_TYPE_Q5_1
    ACL_DT_UNDEFINED, // GGML_TYPE_Q8_0
    ACL_DT_UNDEFINED, // GGML_TYPE_Q8_1
    ACL_DT_UNDEFINED, // GGML_TYPE_Q2_K
    ACL_DT_UNDEFINED, // GGML_TYPE_Q3_K
    ACL_DT_UNDEFINED, // GGML_TYPE_Q4_K
    ACL_DT_UNDEFINED, // GGML_TYPE_Q5_K
    ACL_DT_UNDEFINED, // GGML_TYPE_Q6_K
    ACL_DT_UNDEFINED, // GGML_TYPE_Q8_K
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ2_XXS
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ2_XS
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ3_XXS
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ1_S
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ4_NL
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ3_S
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ2_S
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ4_XS
    ACL_INT8,           // GGML_TYPE_I8
    ACL_INT16,          // GGML_TYPE_I16
    ACL_INT32,          // GGML_TYPE_I32
    ACL_INT64,          // GGML_TYPE_I64
    ACL_DOUBLE,         // GGML_TYPE_F64
    ACL_DT_UNDEFINED, // GGML_TYPE_IQ1_M
    ACL_BF16            // GGML_TYPE_BF16
};
```

```cpp
size_t ggml_type_size_t[GGML_TYPE_COUNT] = {
    sizeof(float),      // GGML_TYPE_F32
    sizeof(short),      // GGML_TYPE_F16
    1,                  // GGML_TYPE_Q4_0
    1,                  // GGML_TYPE_Q4_1
    1,                  // GGML_TYPE_Q5_0
    1,                  // GGML_TYPE_Q5_1
    1,                  // GGML_TYPE_Q8_0
    1,                  // GGML_TYPE_Q8_1
    1,                  // GGML_TYPE_Q2_K
    1,                  // GGML_TYPE_Q3_K
    1,                  // GGML_TYPE_Q4_K
    1,                  // GGML_TYPE_Q5_K
    1,                  // GGML_TYPE_Q6_K
    1,                  // GGML_TYPE_Q8_K
    1,                  // GGML_TYPE_IQ2_XXS
    1,                  // GGML_TYPE_IQ2_XS
    1,                  // GGML_TYPE_IQ3_XXS
    1,                  // GGML_TYPE_IQ1_S
    1,                  // GGML_TYPE_IQ4_NL
    1,                  // GGML_TYPE_IQ3_S
    1,                  // GGML_TYPE_IQ2_S
    1,                  // GGML_TYPE_IQ4_XS
    sizeof(char),       // GGML_TYPE_I8
    sizeof(short),      // GGML_TYPE_I16
    sizeof(int),        // GGML_TYPE_I32
    sizeof(long long),  // GGML_TYPE_I64
    sizeof(double),     // GGML_TYPE_F64
    1,                  // GGML_TYPE_IQ1_M
    2                   // GGML_TYPE_BF16
};
```

图8 为适配ggml数据类型构建的ggml_to_acl_map和ggml_type_size_t

在组合复用算子和语义实现方面，考虑到llama.cpp已支持的算子大多为适合大模型推理，缩减计算图的复合算子，需要将aclnn算子以符合原有语义的方式组合。以RoPE算子为例，在运行llama2-7b-f16模型时，llama.cpp的入参ggml_tensor结构体包含 freq_base，n_dim，freq_scale，pos等参数，以符合以下公式的语义逻辑进行计算：

$$\theta_{j,i} = pos[j] * freq\_scale * freq\_base^{-\frac{2i}{n\_dim}}$$

$$\begin{bmatrix} dst_{j,2i} \\ dst_{j,2i+1} \end{bmatrix} = \begin{bmatrix} cos\theta_{j,i} & -sin\theta_{j,i} \\ sin\theta_{j,i} & cos\theta_{j,i} \end{bmatrix} \begin{bmatrix} x_{j,2i} \\ x_{j,2i+1} \end{bmatrix}$$

因而，根据aclnn算子已提供的接口，将llama.cpp的RoPE算子逻辑拆解为等效的aclnn算子拼接，具体执行流如下：

首先通过aclnn_pow_scalar_tensor_func调用AOL算子加速库中的aclnnPowScalarTensor算子，计算得到$freq_base^{-\frac{2i}{n_dim}}$。

```cpp
ret = aclnn_pow_scalar_tensor_func(theta_scale, powExpDeviceAddr, powOutDeviceAddr,
    powExpShape, powOutShape,
    ACL_FLOAT, ACL_FLOAT, ACL_FLOAT,
    stream);
CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("ggml_ascend_rope failed. ERROR: %d\n", ret); return);
```

图9 调用aclnnPowScalarTensor

然后，由于数据格式上的对齐要求，通过aclnn_repeat_func调用aclnnRepeat算子，将$freq_base^{-\frac{2i}{n_dim}}$和$pos$复制扩充以满足进一步算子计算的需要。

```
ret = aclnn_repeat_func(powOutDeviceAddr, mulOtherDeviceAddr,
                        repeatSelfShape, repeatOutShape,
                        ACL_FLOAT, ACL_FLOAT,
                        repeatsArray, stream);
CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("ggml_ascend_rope failed. ERROR: %d\n", ret); return);
```

```
ret = aclnn_repeat_func(pos, mulSelfDeviceAddr,
                        repeatSelfShape2, repeatOutShape2,
                        ACL_INT32, ACL_INT32,
                        repeatsArray2, stream);
CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("ggml_ascend_rope failed. ERROR: %d\n", ret); return);
```

图10，11 aclnn_repeat函数调用

进一步，对$pos[j] * freq_scale * freq_base^{-\frac{2i}{n_dim}}$，根据运算元素的数据格式不同，分别调用aclnnMul和aclnnMuls算子，得到$\theta_{j,i}$，而后按照计算公式，调用aclnnSin和aclnnCos算子，计算出$sin\theta_{j,i}$和$cos\theta_{j,i}$方便下一步运算。

```
ret = aclnn_mul_func(mulSelfDeviceAddr, mulOtherDeviceAddr, mulOutDeviceAddr,
                     mulSelfShape, mulOtherShape, mulOutShape,
                     ACL_INT32, ACL_FLOAT, ACL_FLOAT,
                     stream);
CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("ggml_ascend_rope failed. ERROR: %d\n", ret); return);
```

```
ret = aclnn_muls_func(mulsSelfDeviceAddr, mulsOutDeviceAddr, mulsSelfShape, mulsOutShape, ACL_FLOAT, ACL_FLOAT, freq_scale, stream);
CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("ggml_ascend_rope failed. ERROR: %d\n", ret); return);
```

图12，13 aclnn_mul_mat函数调用

最后，通过aclnn_rope_func调用对应算子逻辑实现，需要注意的是，由于华为提供的aclnnApplyRotaryPosEmb算子接口实现逻辑公式如下：

$$\begin{bmatrix} dst_i \\ dst_{\frac{n\_dim}{2}+i} \end{bmatrix} = \begin{bmatrix} cos\theta_i & -sin\theta_i \\ sin\theta_i & cos\theta_i \end{bmatrix} \begin{bmatrix} x_i \\ x_{\frac{n\_dim}{2}+i} \end{bmatrix}$$

与llama.cpp要求的算子逻辑并不相同，因而需要在调用对应aclnn算子前后分别调用两次aclnnPermute算子进行矩阵转置以满足算子语义的实现。

```
ret = aclnn_permute_func(permuteSelfDeviceAddr, permuteOutDeviceAddr,
                         permuteSelfShape, permuteOutShape,
                         ggml_to_acl_map[src0->type], ggml_to_acl_map[src0->type],
                         permuteDims, stream);
```

图14 aclnn_permute函数调用

2）模型集成运行与测试验证

完成高速大型语言模型推理引擎的源代码修改与适配后，运行模型推理并进行相应的测试验证。具体实施过程如下：

在模型方面，集成llama2 AI模型到修改后的llama.cpp推理框架中，确保模型能够正确地通过NPU内存接口加载，通过检查NPU内存占用来确定相关指标。当模型全部卸载到NPU上时，可以看到NPU的内存已经占用16GB。之后进行

运行和调试，通过在实际设备上运行修改后的llama.cpp推理框架，验证模型在NPU上的运行效果和性能表现，根据测试结果确认系统的实际可用性和成熟度。

```
+------------------------------------------------------------------------------------------------+
| npu-smi 23.0.1                                Version: 23.0.1                                  |
+------------------------------+----------------+----------------------------------------------+
| NPU     Name                 | Health         | Power(W)     Temp(C)           Hugepages-Usage(page) |
| Chip    Device               | Bus-Id         | AICore(%)    Memory-Usage(MB)                        |
+==============================+================+==============================================+
| 8       310P3                | OK             | NA           48                7543  / 7543          |
| 0       0                    | 0000:01:00.0   | 5            16889/ 21527                            |
+==============================+================+==============================================+
+------------------------------+----------------+----------------------------------------------+
| NPU     Chip                 | Process id     | Process name             | Process memory(MB)        |
+==============================+================+==============================================+
| 8       0                    | 2473510        | llama-cli                | 15097                     |
+==============================+================+==============================================+
```

图 15 模型加载后 NPU 内存占用

i. 测试计划制定

准备测试的场景为CPU：Intel i7-13700k; NPU：昇腾310P3

计划分为功能性测试(正确性)和性能测试两部分：

功能性测试：测试会将llama2-7b模型分别放在CPU和完全卸载到NPU上运行，设置参数--top-k, --top-p均为1，使得推理框架的采样方式为贪婪采样——该模式下模型的输出不具有随机性，仅取决于模型预测的概率分布，正确实现时，任何卸载参数下模型输出均应一致。

性能测试：llama2-7b的decoder层数为33层，设置四组卸载参数：完全不卸载，卸载11层，卸载22层，卸载33层，每组都用相同的提示进行测试，模型加载时间，响应时间和token生成时间以llama.cpp运行的统计结果为准。

ii. 测试验证与修复完善

正确性方面，在ggml_xx_compute_forward函数中，将算子的输入与输出写入文件，进行批量对比，发现mul_mat算子实现与CPU版本有所偏差，对于rope函数的理解与实现有错误。因此，在CPU上的推理结果与在NPU上差距较大。在正确性的修复中，对于mul_mat算子，llama.cpp实现的算子功能公式如下：

$$Mul\_Mat(A_{b*n*k}, B_{b*n*m})_{b,i,j} = \sum_{k=1}^{n} B_{b,k,i} * A_{b,k,j}$$

因而，对于ggml_ascend_mul_mat算子而言，需要将AOL算子加速库中的aclnnPermute和aclnnBatchMatMul算子进行组合以符合llama.cpp算子逻辑。

```
ret = aclnn_permute_func(permuteSelfDeviceAddr, permuteOutDeviceAddr,
                         permuteSelfShape, permuteOutShape,
                         ggml_to_acl_map[src0->type], ggml_to_acl_map[src0->type],
                         permuteDims, stream);
```

```
ret = aclnn_batch_mat_mul_func(batchMatMulSelfDeviceAddr, batchMatMulMat2DeviceAddr, batchMatMulOutDeviceAddr,
                        batchMatMulSelfShape, batchMatMulMat2Shape, batchMatMulOutShape,
                        ggml_to_acl_map[src1->type], ggml_to_acl_map[src1->type], ggml_to_acl_map[dst->type],
                        stream);
```

图 16，17 mul_mat 算子的修复

对于 rope 算子，团队起初错误地理解了 ggml 中 rope 算子的算法，实现的算子并没有考虑到 pos 参数的正确 broadcast 维度，错误使用 aclMalloc 和 aclrtMemcpy 反复对参数进行拷贝，不仅对结果正确性产生了较大影响，同时显著降低了性能。经过对算子语义的重新审视和排查后，团队修正了算子语义的具体公式，通过多次利用 aclnnRepeat 接口，将 pos 参数在正确维度进行 broadcast，从而得到了误差较小的结果。最终在多次 rope 算子中间结果和最终输出的比较中，CPU 推理结果和 NPU 推理结果的误差降低到了 1e-5 级别，表明该修复正确。

```
ret = aclnn_repeat_func(powOutDeviceAddr, mulOtherDeviceAddr,
                        repeatSelfShape, repeatOutShape,
                        ACL_FLOAT, ACL_FLOAT,
                        repeatsArray, stream);
```

图 18 rope 算子中 repeat 函数调用

NpuID(Idx)	ChipId(Idx)	Pwr(W)	Temp(C)	AI Core(%)	AI Cpu(%)	Ctrl Cpu(%)	Memory(%)	Memory BW(%)
8	0	NA	53	9	0	90	80	16
8	0	NA	54	7	0	93	78	14
8	0	NA	54	5	0	90	78	14
8	0	NA	54	7	0	89	78	16
8	0	NA	54	8	0	90	79	16
8	0	NA	54	8	0	91	80	16
8	0	NA	54	8	0	90	78	13
8	0	NA	54	9	0	90	79	13
8	0	NA	55	7	0	91	78	14
8	0	NA	55	7	0	90	80	14
8	0	NA	55	13	0	91	78	18

图 19 推理过程中 NPU 资源占用

性能方面，在修复算子正确性后，团队对在NPU设备上的推理性能进行测试，发现NPU的计算缓慢，在NPU上卸载的模型层数越多，则生成token的速度越慢，为查明原因，团队使用CANN提供的msprof工具来寻找性能瓶颈。分析结果如下：在算子的组合实现过程中，生成大量的中间结果，这些中间结果每次都需要调用aclMalloc接口分配，再通过aclMemcpy接口分配，算子最后还要使用aclFree接口进行释放，内存操作占用了大量的时间，NPU的计算资源没有被充分利用。团队重新排查了算子的实现，以softmax为例，

Device_id	Level	API Name	Time(us)	Count	Avg(us)	Min(us)	Max(us)	Variance
0	acl	aclrtMemcpy	10685596	1914	5582.861	4.408	96848.55	117932830
0	acl	aclrtFree	8243889	4955	1663.752	0.458	542480.3	67613767.3
0	acl	aclrtSynchronizeStream	2029714	3806	533.293	1.051	9236.757	788587.166
0	acl	aclrtMalloc	878256.5	5019	174.986	0.698	51367.03	651596.503
0	acl	aclMallocMemInner	877248.1	5019	174.785	0.566	51365.97	651572.31
0	acl	aclrtMemset	159093.4	66	2410.506	15.018	79119.39	183320668
0	acl	aclnnRmsNormGetWorkspaceSize	122030	130	938.692	1.058	121577	112818626
0	acl	aclnnBatchMatMul	52998.16	578	91.692	39.002	25981.1	1161790.12
0	node	launch	51316.52	7454	6.884	3.483	273.716	37.627

图20 msprof工具捕获的API调用信息

llama.cpp的实现为先将kq矩阵加上mask矩阵，然后才进行真正的softmax操作,起初为了实现这一组合功能,在移植的softmax矩阵中，先调用add算子计算中间结果，期间调用aclMalloc，aclMemcpy接口进行存放，然后才送入CANN提供的softmax接口。经过进一步查看文档发现：CANN已经提供了aclnnMaskedSoftmaxWithRelPosBias接口，可以直接传入src和mask矩阵，参数所需的bias矩阵只需要构造一个大的，全局存在的0矩阵，每次进入函数直接从这块内存切分即可，无需多次内存操作。经过对不同算子的优化调整，最终推理时延下降50%。

3.工程方案分析论证

本节将深入探讨llama.cpp推理框架在面向国产昇腾NPU的适配与优化过程中所面临的关键技术难点，并对解决方案进行详细的分析和论证。本案例旨在解决硬件资源受限场景下，如何实现大模型的高效推理，并培养对工程问题的发现、识别、分析和解决等能力。重点是将大模型推理框架适配至国产AI芯片与基础软硬件环境中，为我国软硬芯生态积蓄力量。

技术难点分析与解决方案

1）NPU运行时API不完善：

在项目初期进行NPU设备测试时，发现在Atlas310P3NPU设备上异步内存拷贝接口aclrtMemcpyAsync与流同步接口aclrtSynchronizeStream的交互存在bug，具体表现为异步内存拷贝之后，立刻调用同步流接口会引发设备错误，表现为同步繁忙；在llama.cpp的GPU后端中，通过显式调用cudaGetDeviceProperties接口获取deviceInfo，但是昇腾NPU并未提供等效接口，无法直接通过C/C++接口获取设备详细信息，对于一些依赖设备信息的上层接口，难以直接仿照cuda的形式进行移植。

解决方案：对于内存拷贝与同步的问题，将异步形式更换为了同步内存拷贝，防止设备内部产生问题；对于设备信息无法统一获取的问题，在CANN库的另一些头文件中寻找到了获取设备特定项目信息的接口，同时，通过在shell

输入npu-smi命令可以显示地获取到部分的NPU信息，可以将这些信息作为参数输入llama.cpp程序，这在一定程度上解决了设备信息接口缺失的问题。

2）AOL算子和ggml算子语义不对齐：

llama.cpp使用的ggml算子专门针对大语言模型的一般结构（典型的如Decoder-only Transformer）进行了优化，通过在单算子的核函数内，整合多个复杂计算逻辑，达到压缩计算图大小、减少核函数启动开销的目的。华为CANN提供的AOL算子加速库中，提供了一套通用算子的实现，基于AOL算子加速库，为ggml库增加NPU算子时，会遇到语义不对齐问题，具体表现为：i. AOL接口与ggml接口入参格式不同；ii. AOL算子负责单一计算逻辑，通常需要组合多个AOL算子以等效单个ggml算子，导致计算图扩大，核函数启动开销上升。

解决方案：对于 AOL 接口与 ggml 接口入参格式不同的问题，增加参数转换层，满足不同数据格式的类型转换与互推导关系的需求。对于频繁调用的热算子，根据其等效的 ggml 算子逻辑撰写核函数，使用 msopgen 创建子工程，完成其 op_kernel 和 op_host 侧的实现，并且编译和部署对应工程以方便使用 aclnn 接口方式调用。

为了有效应对上述技术难点，最大程度地完成推理框架从GPU搭配NPU的无缝切换，优化大模型在昇腾NPU软硬件环境下的推理效率和性能，提升系统的整体可用性和可靠性，在考虑过多种方案和场景对比的策略之后，并在优化折衷的过程中得出最终的工程方案：

1）软件框架适配：后端接口方面，复用和修改推理框架的原有执行流和加载执行策略以适应推理框架的模块层次结构，同时针对NPU设备实现设备信息系统配置和运行时管理接口；算子接口方面，充分利用NPU平台已开发算子，更改参数和执行流以匹配推理框架原有算子逻辑，进行替代以达到原有算子的计算功效。

2）软件算法优化：针对 NPU 平台特性，优化移植框架和算子的执行流，考虑修改模型卸载到 NPU 的方式，对于 NPU 平台已经实现的但是没有与推理框架功能一致的算子，考虑编写核函数开发 NPU 算子，以达到进一步降低开销的效果，持续对已移植部分进行迭代优化。

4.实施效果

在llama.cpp推理框架的适配与移植完成后，团队深入评估了其在多个方面的实施效果。这不仅包括系统是否满足了技术指标和功能需求，还包括项目是否通过了验收、经历了实践的检验、以及项目的下一步的移植与优化目标。

1）指标需求的评估

智能操作系统的实施目标是在华为昇腾NPU的软硬件生态上实现llama.cpp推理框架的适配与优化。在实施过程中，设定了多项关键的技术指标，以确保系统能够在性能、正确性和稳定性等方面达到预期的要求，为进一步的优化和改进提供数据支持。

首先，对性能进行评估。针对系统的各种负载情况进行了详细的性能测试，包括计算能力，系统运行速率等。这些测试的运行旨在精确评估系统在处理大规模数据和复杂推理任务时的表现，确保其在不同大小的模型和不同长度的输入下都能保持稳定和高效；其次，对正确性和稳定性进行评估。通过比较CPU和NPU上运行系统的输出内容，确保移植推理框架在国产昇腾NPU设备上的正确性。此外，长时间运行和各种异常情况下的系统稳定性也是测试的重点之一，以保证系统在不同卸载参数条件下均能稳定可靠地运行。

2）验收情况与实践检验

llama.cpp推理框架的移植与适配成功与否不仅仅取决于技术指标的达标，还需要通过验收测试来确认系统的实际可用性和成熟度。验收测试涵盖了系统的功能完整性、性能稳定性两个方面。

在功能测试中，验证了llama.cpp推理框架的功能已经被正确地移植到NPU设备上，团队测试了将llama2-7b模型分别以不卸载(在CPU上推理)，部分卸载和完全卸载的方式进行推理，并对中间结果输出进行了对比(以CPU推理为基准)，其结果是每层张量的计算结果与预期结果的误差均小于1e-4，处于可接受范围之内；无论模型卸载层数多少，在贪婪采样方式下运行都能产生相同的结果，这也印证了当模型卸载到NPU时计算结果的正确性。

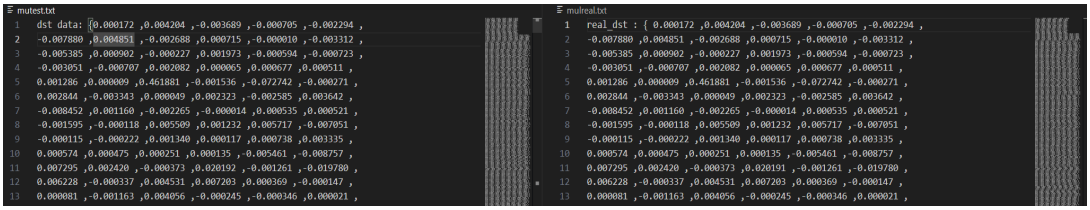

图21 对比CPU与NPU计算结果

性能测试则着重评估系统在生成时的表现，包括模型加载时间、响应时间、生成速度和资源利用率等方面，该项测试中，经过多次程序运行实验，实验环境为CPU Intel i7-13700k, NPU Atlas310P3，llama2-7b模型的加载时间可以稳定在3秒左右；程序的响应时间取决于对prompt的处理速度，对于中等长度的提示，每次对话模型需要等待大约5秒的时间可以生成第一个token；在后续的token生成过程中,最好的情况为2个/3秒。在追查token生成速度稍慢的原因时，团队发现：在计算过程中，NPU的AIcore利用率峰值仅达到45%，而其内存带宽则一直被大量占用，推测为内存操作过于频繁，抑制了计算任务的有效开展，

作为初次验收实验，性能在可接受范围内，仍有很大的可优化空间：比如异步内存拷贝的设备bug的消除；对于单一NPU设备无需使用过多的setDevice操作等。

```
> What is Lvalue an Rvalue in C++?
  Lvalue is a value which may be on the left side of an assignment or a statement.
It is a valid value at the time of assignment or statement.
Rvalue is a value which is only used in the right side of an assignment statement.
It is a temporary value that is not valid at the time of assignment.
For example:
int a = 5; // a is an Lvalue
int b = 5; // b is an Rvalue
C++11 introduces an rvalue reference as a unary operator, rvalue references are also called move constructors.
[INST]
> What about initializer_list in C++11 or later?
  C++11 has a new feature called initializer_list.
It allows you to initialize a variable with a list of values.
For example:
int a[5] = {1, 2, 3, 4, 5};
In the above example, a is an array of 5 ints, and each index of the array is initialized with a unique integer value.
int a[5] = {1, 2, 3, 4, 5}; // a is an Lvalue
int b[5] = {1, 2, 3, 4, 5}; // b is an Rvalue
> good bye thank you
  You are welcome.
 [INST]
> exit
  Bye.
 [INST] Bye.

>

llama_print_timings:        load time =    3467.07 ms
llama_print_timings:      sample time =      13.96 ms /   519 runs   (    0.03 ms per token, 37177.65 tokens per second)
llama_print_timings: prompt eval time =  114715.87 ms /    64 tokens ( 1792.44 ms per token,     0.56 tokens per second)
llama_print_timings:        eval time =  773325.62 ms /   514 runs   ( 1504.52 ms per token,     0.66 tokens per second)
llama_print_timings:       total time = _906839.18 ms /   578 tokens
```

图22 一次对话运行示例

以上验收测试的全部合格，不仅证明了llama.cpp推理框架的移植在技术实现上的成功，也为后续的应用和进一步扩展提供了强有力的保障。验收通过后，系统进入下一次迭代阶段，下一步的计划是优化内存操作的效率和算子实现的复杂度，及时发现和解决可能存在的问题，确保系统能够更快更准确地运行。

3）下一步的计划与目标

i. 对量化的进一步支持，在NPU设备上，CANN仅提供了8bit量化相关的接口。当前存在的模型，一般有fp16、int8、int4等量化方式，同时llama.cpp支持的量化数据类型还包括了int5，int6这样的类型，以支持模型在某些层上特殊的结构。因此考虑到支持更多量化模型，同时权衡实现难度，团队预计在CANN已经有int8量化的基础上，支持应用范围更广的int4量化，进一步提高模型推理性能。

ii. 对PowerInfer推理框架的支持，PowerInfer推理框架是对llama.cpp的改进，结合大模型的独特特征，对模型推理进行了深度优化——即模型存在的高局部性与稀疏激活，少部分神经元在总体上被激活的概率更高，在统计意义上属于幂律分布。PowerInfer提前静态分析神经元的冷热性，将少量热神经元加载到GPU内存，冷神经元加载到CPU内存。在已有算子的基础上，实现稀疏算子，CPU与GPU各自处理相关的神经元，并在GPU中进行合并。团队计划在之后的工作

中研究PowerInfer的稀疏算子实现，进一步阅读昇腾NPU算子文档，根据NPU对于稀疏算子的支持程度，开展对PowerInfer推理框架的NPU移植与优化。

5.结语部分

近年来，随着人工智能技术的快速发展，大型语言模型在自然语言处理领域的应用愈发广泛。昇腾 NPU 作为国产 AI 芯片的代表，具有高效的计算能力和出色的并行处理性能，针对密集矩阵运算场景，对硬件架构进行了创新设计，为大模型的推理提供了良好的硬件基础。因此，团队选择将 llama.cpp 推理框架这一支持各种大语言模型推理的开源软件库适配到昇腾 NPU 上，以验证其在国产硬件平台上的性能表现和优化潜力。

在项目实施过程中，团队针对诸多技术挑战，包括适配 NPU 设备配置、内存管理机制以及算子在 NPU 设备上的移植等问题，通过详细研究昇腾 NPU 的技术文档和开发指南，理解其系统配置、内存管理机制以及与其他硬件组件的交互方式，提出自己的技术路线，有机结合 CANN 提供的算子加速库，对高速大型语言模型推理框架的源代码进行了修改与适配。具体实施过程中，根据项目要求和出现的困难，团队通过修补技术路线缺漏，借助 msprof，msopgen 等工具辅助开发，遵循快速原型模型进行程序开发等方式，利用多次正确性测试不断修正项目开发进度和实现内容，确保了项目实现的正确性，避免了可能的早期错误以及过长的软件阶段开发周期。而在项目后期的集成测试和系统测试阶段，团队不仅结合先前的正确性测试样例进行进一步的功能测试，同时进行了性能测试以确认系统的实际可用性和成熟度，为接下来优化系统性能和进行下一步迭代迭代提供了技术路线和工程方案。

总的来说，本次项目的成功不仅证明了 llama.cpp 推理框架在技术实现上的可行性，也为后续的应用和扩展打下了基础。在未来的研究中，团队将继续优化推理框架，确保系统能够更快、更准确地运行，为大模型的高效推理和广泛应用奠定坚实基础。可以期待，在未来端侧设备，如智能手机，个人电脑，甚至电子手表中都可以离线运行大模型应用，在为用户带来便利的同时也提供模型安全性的保障。

参考文献

（1） 昇腾NPU开发者文档，华为技术有限公司。

（2） Intel NPU参考文档 www.intel.com/content/www/us/en/products/details/processors/core-ultra.html

（3） Jouppi, Norman P., et al. ″In-datacenter performance analysis of a tensor processing unit.″ Proceedings of the 44th annual international symposium on computer architecture. 2017.

（4） ggerganov/llama.cpp: LLM inference in C/C++ (github.com)

（5） https://qwen.readthedocs.io/zh-cn/latest/run_locally/llama.cpp.html

（6） SJTU-IPADS/PowerInfer: High-speed Large Language Model Serving on PCs with Consumer-grade GPUs (github.com)

（7） 算子规格说明 (Atlas 推理系列产品，Ascend 310P处理器)，华为技术有限公司。

（8） 《关于加快场景创新以人工智能高水平应用促进经济高质量发展的指导意见》，科技部。

教学指导手册：

智能操作系统实践案例：国产软硬芯的大模型基础软件适配与优化

1. 教学目标

（1）适用课程：本案例主要适用于物联网技术课程，也适用于软件工程、仪表仪器课程。

（2）适用对象：本案例主要为工程相关实践教学目标开发，适用于工程硕博士人才培养。

（3）教学目的：

培养学生在硬件资源受限环境下，进行大模型基础软件适配与优化的能力。

引导学生理解并掌握大模型推理框架在国产AI芯片与基础软硬件环境中的应用与优化技术。

培养学生解决实际工程问题的能力，包括问题发现、分析、综合解决方案等方面的综合能力。

2. 讨论问题

如何评估一个大模型在资源受限环境下的适配性和性能优化空间？

在国产AI芯片与传统硬件平台之间的性能差异和优劣如何？如何进行比较和评估？

在大模型推理过程中，如何平衡推理速度、功耗效率和模型准确率之间的关系？

如何利用现有的优化技术和工具，提高大模型在硬件资源受限环境下的运行效率？

未来如何进一步优化和扩展国产软硬芯生态系统，以支持更复杂和大规模的AI应用？

3. 分析思路

本案例分析的逻辑结构主要围绕以下几个方面展开：

理解大模型在各种应用场景下的关键技术挑战和优化需求。

探讨国产AI芯片特性及其在大模型推理中的优势与劣势。

分析现有的大模型适配和优化方法，包括硬件加速、算法改进和软件优化等方

面。

总结不同优化方案的比较和评估，找出最佳的技术折衷方案以及未来的发展方向。

4. 案例分析

在课堂上，将选取合适的理论模型和分析工具，对大模型基础软件适配与优化进行深入分析，重点关注：

国产软硬芯特有的技术挑战和解决方案。

大模型推理过程中的关键性能指标评估和优化策略。

实际案例中的应用效果和技术成果展示。

5. 课堂设计

时间安排：案例分析阶段占据大部分时间，包括理论讲解、案例分析、实验演示等，预留时间用于讨论和总结。

教学形式与环节设计：结合理论授课、案例讨论、小组讨论和实验室实践，以促进学生的理论与实际操作能力结合。记录重要概念、案例关键信息和学生讨论结果，以帮助学生理清思路和概念。

6. 要点汇总

梳理案例涉及的主要教学知识点，包括大模型适配与优化的关键技术、国产AI芯片特性、优化策略和应用案例等。

总结案例启示，包括技术应用前景、工程实践经验和未来研究方向的建议。

7. 其他说明

推荐阅读的相关资料：包括最新的学术论文、行业报告和技术手册，以帮助学生深入理解和应用案例中涉及的技术和方法。

案例后续进展：随着技术的发展和应用场景的变化，案例可能会有进一步的优化和更新，建议关注相关领域的最新动态和趋势。

通过以上教学指导手册，希望能够有效引导学生深入理解和应用智能操作系统实践案例，培养其解决复杂工程问题的能力，为国产软硬芯的发展贡献更多积极力量。

作者授权书

教育部学位与研究生教育发展中心：

本人同意案例 智能操作系统实践案例：国产软硬芯的大模型基础软件适配与优化 被教育部学位与研究生教育发展中心所属的中国专业学位案例中心收录。

本人郑重声明如下：

1.该案例为作者原创，未公开发表，未一稿多投。

2.该案例所有引用资料均已注明出处，不涉及保密与知识产权的侵权等问题，对于署名无异议。

3.该案例被教育部学位与研究生教育发展中心收录后：

（1）作者享有案例的署名权、修改权、改编权，教育部学位与研究生教育发展中心享有并有权同意第三方享有以下权利：

案例的复制权、修改权、发表权、发行权、信息网络传播权、改编权、汇编权和翻译权；代表本人与其他机构或个人进行案例交换、购买、出版等商务谈判、合作的权利。

（2）未经教育部学位与研究生教育发展中心书面同意，本人不得授权第三方以任何方式使用该案例。

本授权书由第一作者签字确认，并对各项承诺负全责。

授权书所涉及事项对该案例全体作者具有约束力。

如本案例未被中国专业学位案例中心收录，本授权书自动失效。

第一作者签字（手签）：

所属单位：

日　　期：

单位授权书

教育部学位与研究生教育发展中心：

_____撰写（指导）的案例___智能操作系统实践案例：国产软硬芯的大模型基础软件适配与优化_是在对我单位有关人员采访的基础上完成的，案例中涉及到对于我单位的相关描述是客观的，我单位予以认可。

特此声明。

单位名称（公章）：

授权代表：

日　期：