第3章

CANN自定义算子开发

深度学习通过组合算子构建不同应用功能的网络结构。昇腾CANN软件栈提供的算子都是预先实现和编译的，是华为公司工程师使用达芬奇架构专用编程语言开发的高度优化的内核函数，能够较好地适配底层的硬件架构，具有较高的性能。当遇到昇腾CANN软件栈不支持网络中的算子、用户想要修改现有算子中的计算逻辑或者用户想要开发更高性能的算子这三种情况时，昇腾CANN软件栈允许用户根据自己的需求借助张量加速引擎（Tensor Boost Engine，TBE）工具开发自定义算子，并支持开发运行在AI Core上或AI CPU上的算子。

本章首先简要介绍TBE开发，然后对TBE自定义算子开发工具体系进行叙述，通过三种方式（TBE DSL、TBE TIK、AI CPU）介绍算子开发的原理及实现方法，并提供了自定义算子样例。

3.1 TBE开发概述

本节首先简单概述算子和TBE基本概念，接着介绍TBE开发方式的选择、开发流程、开发交付件等内容。

3.1.1 算子基本概念

算子（Operator，简称OP）是深度学习算法的基本函数计算单元。在网络模型中，算子对应网络层中的计算逻辑。例如卷积层（Convolution Layer）是一个卷积函数的算子；全连接层（Fully-Connected Layer，FC Layer）中的权值加权求和过程也是一个算子。这些算子计算所涉及数据的容器称为张量（Tensor）。张量具有特定的数据排布格式（如NCHW或NHWC）。下面介绍算子中常用术语的基本概念。

1. 算子名称与类型

算子名称用于标识网络中的某个算子，同一网络中算子的名称需要保持唯一。例如在一个网络中可以定义conv1、pool1、conv2等算子名称，其中conv1与conv2算子的类型为Convolution，表示分别做一次卷积运算。

算子类型是代表算子的函数运算类型，例如卷积算子的类型为卷积运算，在一个网络中同一类型的算子可能存在多个。

2. 张量的描述

算子中的数据经常是多维的，那么 Tensor 就是记录数据的容器。在几何代数中定义的张量是基于向量和矩阵的推广。通俗一点理解，可以将标量视为零阶张量，向量视为一阶张量，那么矩阵就是二阶张量。数据包括输入数据与输出数据，TensorDesc（Tensor 描述符）是对输入数据与输出数据的描述，TensorDesc 数据结构说明如表 3-1 所示。

表 3-1　TensorDesc 数据结构说明

属　　性	定　　义
名称(name)	用于对张量进行索引，不同张量的名称需要保持唯一
形状(shape)	张量的形状：比如(10,)或者(1024,1024)或者(2,3,4)等 默认值：无
数据类型(dtype)	功能描述：指定张量对象的数据类型 默认值：无 取值范围：float16，float32，int8，int16，int32，uint8，uint16，bool 说明：不同计算操作支持的数据类型不同
数据排布格式(format)	数据的物理排布格式，定义解读数据的维度

3. 张量排布格式

在深度学习领域，多维数据通过多维数组存储，比如卷积神经网络的特征图（Feature Map）通常用四维数组保存，即 4D。四维数组格式解释如下：

（1）N：Batch 数量，例如图像的数目。

（2）H：Height，特征图高度，即垂直高度方向的像素个数。

（3）W：Width，特征图宽度，即水平宽度方向的像素个数。

（4）C：Channels，特征图通道，例如彩色 RGB 图像的 Channels 为 3。

由于数据只能线性存储，因此这四个维度有对应的顺序。不同深度学习框架会按照不同的顺序存储特征图数据。比如 Caffe，排列顺序为[Batch，Channels，Height，Width]，即 NCHW。TensorFlow 中，排列顺序为[Batch，Height，Width，Channels]，即 NHWC。

以一张格式为 RGB 的图片为例，如图 3-1 所示。使用 NCHW 排布格式时，C 排列在外层，实际存储的是“RRRRRRGGGGGGBBBBBB”，即同一通道的所有像素值顺序存储

图 3-1　不同排布格式下 RGB 图片的存储

在一起；而 NHWC 中 C 排列在最内层，实际存储的则是“RGBRGBRGBRGBRGBRGB”，即多个通道的同一位置的像素值顺序存储在一起。

在不同的硬件加速的情况下，选用的数据排布格式不同：

（1）对于 CPU，NHWC 比 NCHW 稍快一些，因为 NHWC 的局部性更好，缓存利用率更高。

（2）对于 GPU，图像处理比较多，希望访问同一个通道的像素是连续的，则一般存储采用 NCHW。

3.1.2　TBE 基本概念

TBE（张量加速引擎）提供了基于张量虚拟机（Tensor Virtual Machine，TVM）这一开源神经网络编译器开发自定义算子的功能。一般情况下，通过深度学习框架中的标准算子实现的神经网络模型已经通过 GPU 或者其他类型神经网络芯片的训练。如果将这个神经网络模型继续运行在昇腾 AI 处理器上时，希望尽量在不改变原始代码的前提下发挥它的最大性能，TBE 提供了一套完整的算子加速库，库中的算子功能与神经网络中的常见标准算子保持了一一对应关系，并且由 CANN 软件栈提供了编程接口供调用算子使用。

如果在神经网络模型构造中出现了新的算子，这时张量加速引擎中提供的标准算子库无法满足开发需求。此时需要通过 TBE 语言进行自定义算子开发，这种开发方式和 GPU 上利用 CUDA C++ 的方式相似，可以实现更多功能的算子，灵活编写各种网络模型。用户需要进行自定义算子开发的场景有：

（1）昇腾 CANN 软件栈不支持网络中的算子。

（2）用户想要修改现有算子中的计算逻辑。

（3）用户想要自己开发算子来提高计算性能。

通过 TBE 语言和自定义算子编程开发界面可以完成相应神经网络算子的开发，TBE 功能框架如图 3-2 所示，包含了算子逻辑描述语言模块、调度（Schedule）模块、中间表示（Intermediate Representation，IR）模块、编译器传递（Pass）模块以及代码生成（CodeGen）模块。

图 3-2　TBE 功能框架

对于一个 TBE 算子的开发流程叙述如下。

（1）TBE 算子开发分为计算过程的编写与调度开发，TBE 提供直接基于特性域语言（Domain-Specific Language，DSL）以及张量迭代内核（Tensor Iterator Kernel，TIK）开发算子的计算过程和调度过程。算子计算过程描述指明算子的计算方法和步骤，而调度过程描述完成数据切块和数据流向的规划。算子每次计算都按照固定数据形状进行处理，这就需要提前针对在昇腾 AI 处理器中的不同计算单元上执行的算子进行数据形

状切分，如矩阵计算单元、向量计算单元以及 AI CPU 上执行的算子对输入数据形状的需求各不相同。

(2) 在完成算子的基本实现过程定义后，需要启动调度模块中分块(Tiling)子模块，对算子中的数据按照调度描述进行切分，同时指定好数据的搬运流程，确保在硬件上的执行达到最优。除了数据形状切分之外，TBE 的算子融合和优化能力也是由调度模块中的融合(Fusion)子模块提供的。

(3) 算子编写完成后，需要生成中间表示进一步优化，而中间表示模块通过类似于 TVM 的 IR(Intermediate Representation)格式进行中间表示的生成。在中间表示生成后，需要将模块针对各种应用场景进行编译优化，优化的方式有双缓冲(Double Buffer)、流水线(Pipeline)同步、内存分配管理、指令映射、分块适配矩阵计算单元等。

(4) 在算子经过编译器传递模块处理后，由 CodeGen 生成类 C 代码的临时文件，这个临时代码文件可以通过编译器生成算子的实现文件，可被网络模型直接加载调用。

综上所述，一个完整的自定义算子可由 TBE 中的子模块完成整个开发流程，首先基于 DSL 或者 TIK 提供算子计算逻辑和调度描述，构成算子原型后，由调度模块进行数据切分和算子融合，进入中间表示模块，生成算子的中间表示。编译器传递模块以中间表示进行内存分配等编译优化，最后由代码生成模块产生类 C 代码可供编译器直接编译。张量加速引擎在算子的定义过程中不但完成了算子编写，而且还完成了相关的优化，提升了算子的执行性能。

3.1.3 TBE 开发方式与流程

用户可以基于 TBE 使用 Python 语言开发自定义算子，或者使用 C++语言开发 AI CPU 算子，有以下三种方式：DSL 算子开发、TIK 算子开发、AI CPU 算子开发。本节首先介绍三种算子开发方式，随后对三者进行对比并给出各自适应的场景，接着介绍开发所需的交付件内容，最后提供算子开发总体流程。

1. DSL 算子开发

为了方便用户进行自定义算子开发，DSL 接口已高度封装，用户仅需要使用 DSL 接口完成计算过程的表达，后续的调度(Schedule)创建、优化及编译都可通过已有接口一键式完成，适合初级开发用户。DSL 开发的算子性能可能较低。

2. TIK 算子开发

TIK 是一种基于 Python 语言的动态编程框架，呈现为一个 Python 模块。用户可以通过调用 TIK 提供的 API 基于 Python 语言编写自定义算子，即 TIK DSL，然后 TIK 编译器会将 TIK DSL 编译为 CCEC(Cube-based Computing Engine C，面向 C 语言编程的矩阵计算引擎)代码，最终 CCEC 编译器会将 CCEC 代码编译为二进制文件。

基于 TIK 的自定义算子开发，提供了对 Buffer 的管理和数据自动同步机制，但需要用户手动计算数据的分片和索引，需要用户对达芬奇架构非常了解，入门难度更高。TIK

对矩阵的操作更加灵活，性能会更优。

3. AI CPU 算子开发

AI CPU 算子是运行在昇腾 AI 处理器中 AI CPU 计算单元中的表达一个完整计算逻辑的运算。昇腾处理器包含了 AI Core 和 AI CPU 两种计算单元，在某些特殊情况下可能存在 AI Core 不支持的算子(比如部分算子需要 int64 类型)，这时可以通过开发 AI CPU 自定义算子实现昇腾 AI 处理器对此算子的支持。

4. 开发方式的选择

DSL、TIK、AI CPU 三种算子开发方式的比较如表 3-2 所示。

表 3-2　DSL、TIK、AI CPU 三种算子开发方式的比较

参　数	DSL	TIK	AI CPU
语言	Python	Python	C++
计算单元	AI Core	AI Core	AI CPU
运用场景	常用于各种算术逻辑简单的向量运算或内置支持的矩阵运算及池化运算，例如 element-wise 类操作	适用各类算子的开发，对于无法通过 lambda 表达描述的复杂计算场景也有很好的支持，例如排序类操作	在某些场景下，无法通过 AI Core 实现的自定义算子
入门难度	较低	较高	中等
适用人群	入门用户，需要了解神经网络、DSL 相关知识	高级用户，需要了解神经网络，深入理解昇腾 AI 处理器架构、指令集、数据搬运等相关知识	具备 C++程序开发能力，对机器学习、深度学习、AI CPU 开发流程有一定的了解
特点	DSL 接口已高度封装，用户仅需要使用 DSL 接口完成计算过程的表达，后续的调度创建、优化及编译都可通过已有接口一键式完成	入门难度高，程序员直接使用 TIK 提供的 API 完成计算过程的描述及调度过程，需要手工控制数据搬运的参数和调度。用户无须关注 Buffer 地址的分配及数据同步处理，由 TIK 工具进行管理	开发的流程与 DSL 类似，不需要了解 AI Core 的内部架构设计，入门较快
不足	在某些场景下性能可能较低，复杂算子逻辑无法支持表达	TIK 对数据的操作更加灵活，但需要手工控制数据搬运的参数和调度过程。代码编写接近底层硬件架构，过程优化等基于特定硬件特性	因为没有相关封装接口，计算过程相对比较烦琐，因为 AI CPU 性能较低，一般只有在 AI Core 不支持或者临时快速打通的场景下使用

用户在进行算子开发前，需要先进行算子的初步分析，分析算子算法的原理，提取出算子的数学表达式，分析 DSL 提供的接口是否可满足计算逻辑描述的要求，若能够满足，

则优先选用 DSL 方式进行开发。若 DSL 接口无法满足计算逻辑描述或者实现后性能无法满足用户要求，则选择 TIK 算子开发方式，如图 3-3 所示，若 DSL 和 TIK 都不适合开发，则使用 AI CPU 开发方式。

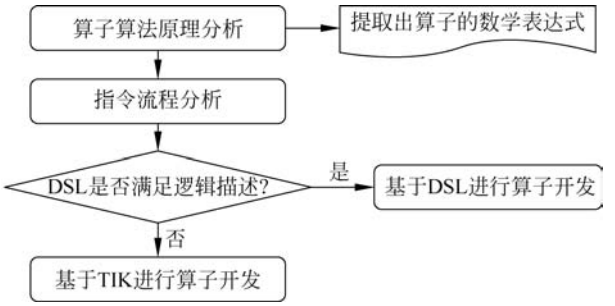

图 3-3　开发方式选择

5. 开发交付件

TBE 算子开发完成后在昇腾 AI 处理器硬件平台上的运行架构如图 3-4 所示。从图中可以看出，算子实现、算子适配插件、算子原型库、算子信息库是开发人员在自定义算子开发时需要实现的交付件。

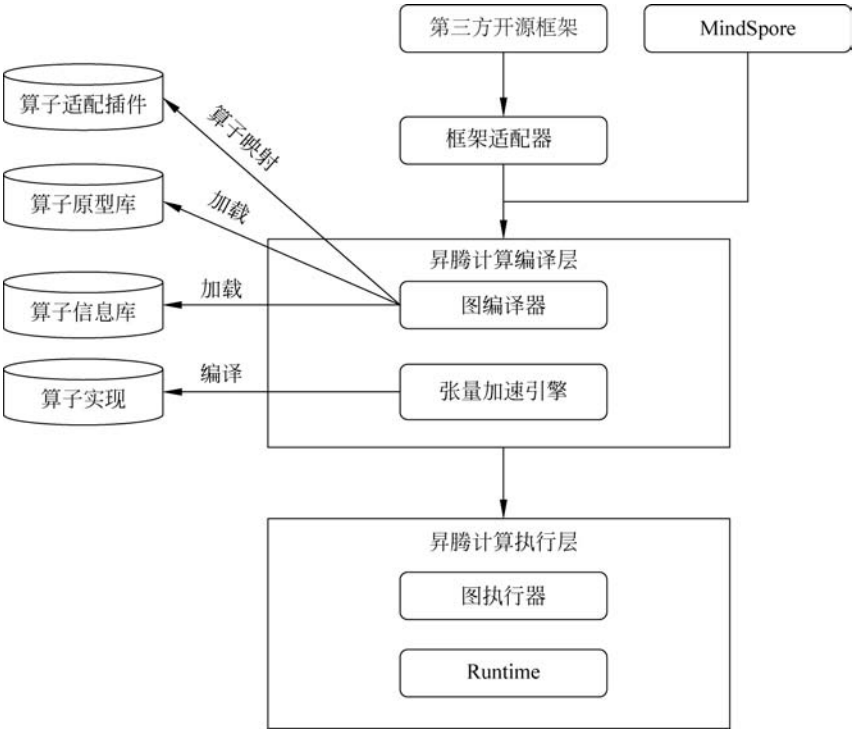

图 3-4　TBE 算子运行架构

（1）算子实现：主要提交内容是算子实现的 Python 文件，包含算子的计算实现及调度实现。

（2）算子适配插件：主要提交内容是基于第三方框架，例如 Tensorflow 进行自定义算子开发的场景，开发人员完成自定义算子的实现代码后，需要进行适配插件的开发将基于第三方框架的算子映射成适合昇腾 AI 处理器的算子，将算子信息注册到图编译器（Graph Compiler）中。基于第三方框架的网络运行时，首先会加载并调用算子适配插件信息，将第三方框架网络中的算子进行解析并映射成昇腾 AI 处理器中的算子。

（3）算子原型库：算子原型库主要体现算子的数学含义，包含定义算子输入、输出、属性和取值范围，基本参数的校验和形状（shape）的推导。网络运行时，图编译器会调用算子原型库的校验接口进行基本参数的校验，校验通过后，会根据原型库中的推导函数推导每个节点的输出形状与数据类型，进行输出张量的静态内存的分配。

（4）算子信息库：算子信息库主要体现算子在昇腾 AI 处理器上物理实现的限制，包括算子的输入/输出数据类型、数据排布格式（format）以及输入形状信息。网络运行时，图编译器会根据算子信息库中的算子信息做基本校验，判断是否需要为算子插入合适的转换节点，并根据算子信息库中信息找到对应的算子实现文件进行编译，生成算子二进制文件进行执行。

6. 开发流程

本章将结合几个简单的算子示例讲述如何基于 TBE 的 DSL、TIK、AI CPU 三种方式进行算子开发。总体的开发流程如图 3-5 所示，其中算子单元测试（UT）仅在 MindStudio 开发场景下支持。

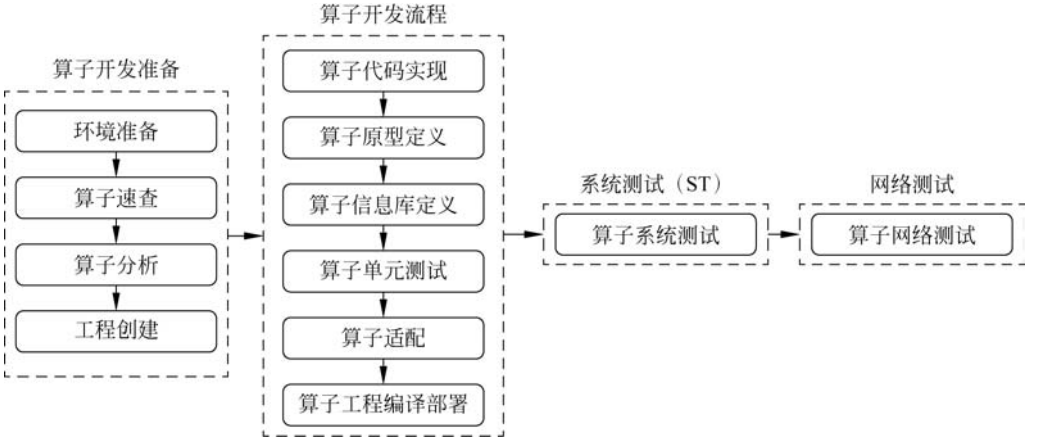

图 3-5　总体的开发流程

3.2 TBE DSL 算子开发

为了方便用户进行自定义算子开发，昇腾 CANN 软件栈借鉴了 TVM 中的 TOPI 机制，预先提供一些常用运算的调度，封装成一个个运算接口，称为基于 TBE DSL 开发。用户只需要利用这些特定域语言声明计算的流程，再使用自动调度（Auto Schedule）机制，指定目标生成代码，即可进一步使目标生成代码被编译成专用内核。

3.2.1 开发环境准备

算子开发可以通过命令行方式开发，也可以通过 IDE MindStudio 方式开发，推荐使用 MindStudio 方式开发。MindStudio 是一套基于 IntelliJ 框架的开发工具链平台，提供了应用开发、调试、模型转换功能，同时还提供了网络移植、优化和分析功能，为用户开发应用程序带来了极大的便利。

MindStudio 只能安装在 Linux 服务器上，因为 MindStudio 是一款 GUI 程序，所以在 Windows 服务器上通过 SSH 登录到 Linux 服务器进行安装时，需要使用集成了 X 服务器的 SSH 终端（比如 MobaXterm，该工具版本需要为 v20.2 及以上）。

MindStudio 可以通过脚本安装，也可以手动安装，为了简化操作，可以选择脚本 msInstaller 安装，具体的安装请参考网址链接 https://support.huawei.com/enterprise/zh/doc/EDOC1100180787/1d74210。

3.2.2 DSL 的 API 接口

TBE 提供了一套计算接口供用户组装算子的计算逻辑，使得 70%以上的算子可以基于这些接口进行开发，极大地降低自定义算子的开发难度。TBE 提供的这套计算接口，称为 DSL 接口。该接口涵盖了向量运算，包括 Math 操作、NN 操作等。

1. 各种计算接口分类

各种计算接口分类与描述如表 3-3 所示。

表 3-3 各种计算接口的分类与描述

分　类	描　述
Math	对张量中每个原子值分别做相同操作的计算接口
NN	与神经网络相关的计算接口
Cast	取整计算接口，对输入张量中的每个元素按照一定的规则进行取整操作

续表

分　　类	描　　述
Inplace	对张量按行进行相关计算的接口
Reduce	对张量按轴进行相关操作的计算接口
Matmul	矩阵乘计算接口
Gemm	通用矩阵乘计算接口
Conv	包含 2D 卷积运算和 3D 卷积运算的相关接口
Pooling2d	2D 池化接口
Pooling3d	3D 池化接口
Array	在指定轴上对输入张量进行重新连接或者切分的接口

2. 自动调度(Auto Schedule)接口

自动调度是一个比较特殊的接口,如果要基于 TBE DSL 编写一个算子,首先通过组合 DSL 计算接口表达算子的计算逻辑,然后调用自动调度接口进行算子的自动调度,完成数据切块和数据流向的划分。调度是与硬件相关的,其功能主要是调整计算过程的逻辑,意图优化计算过程,使计算过程更高效,保证计算过程中占用硬件存储空间不会超过上限。通过自动调度可以简化开发过程,实现过程中只需要关注算子的逻辑表达,底层的优化交给自动调度就可以了。同时,自动调度机制是 TBE 底层的默认调度调优机制,用户无法在算子开发代码过程中进行控制,具体自动调度的原理将在后面代码实现后进行简要介绍。更详细的 API 介绍可以参考链接 https://www.hiascend.com/document 中的 TBE 自定义算子开发。

3.2.3　DSL 算子开发示例

本节正式开始开发一个 DSL 算子程序。为了方便用户进行自定义算子的开发,基于 DSL 开发的算子可以直接使用 TBE 提供的自动调度机制,自动完成调度过程,省去最复杂的调度编写过程。目前昇腾官网提供在线实验 TBE 算子开发(DSL)供用户练习,参考链接为 https://www.hiascend.com/zh/college/onlineExperiment/codeLabTbe/tab,同时也提供了 MindStudio 图形化界面体验算子开发流程,参考链接 https://lab.huaweicloud.com/testdetail_462。

DSL 算子开发大致流程如图 3-6 所示,具体步骤详述如下。

1. 算子分析

使用 TBE DSL 方式开发算子前,首先需要确定算子的功能、输入/输出和逻辑表达,其中重点分析算子算法的原理,提取出算子的数学表达式,再查询 TBE DSL API 接口,看看是否满足算子实现的要求,若 TBE DSL 接口无法满足算子实现的要求,请考虑使用 TIK 开发方式(请参考 3.3 节)。

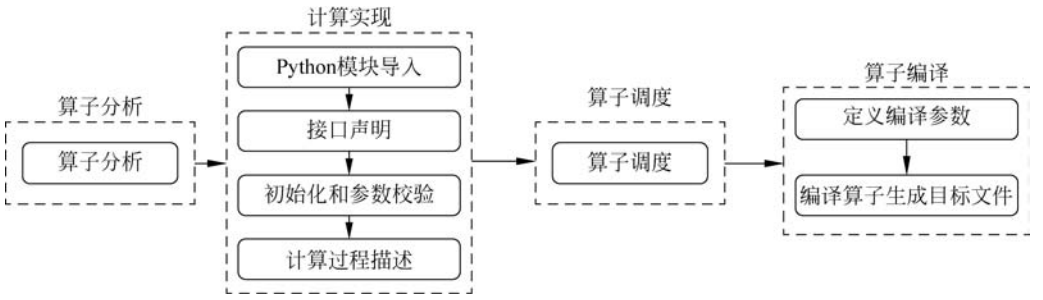

图 3-6　DSL 算子开发大致流程

本节开发的算子就是开平方，输入数据 x，对其进行开平方计算得到输出值 y，即 $y=\sqrt{x}$。

假设当前 TBE DSL API 并没有开平方的现存接口，那么可以通过等价变化，利用已经实现的 API 接口实现开平方功能。对于开平方，转换后的数学表达式为：

$$y=\sqrt{x}=\exp(0.5\times\ln(x))$$

具体的转换逻辑，可以自己推算或者查询，这里不再展开。转换后的表达式主要包括求对数、乘法以及幂的运算。查询 API 手册，相关接口如下：

(1) tbe.dsl.vlog(raw_tensor)；

(2) tbe.dsl.vmuls(raw_tensor,scalar)；

(3) tbe.dsl.vexp(raw_tensor)。

当然，开平方的实现逻辑可以有多种，不一定非要按照上面的表达式实现，比如，可以通过牛顿迭代实现，后面在讲精度优化时会介绍。

2. 算子计算代码实现

算子实现会使用 IDE 工具 MindStudio 进行开发，打开 MindStudio，如果是首次登录 MindStudio，在 MindStudio 欢迎界面中可以单击 Create New Project，创建新工程，也可以直接在 MindStudio 提供的样例工程中新建算子，对于初学者来说可以参考样例代码，提升学习效率。MindStudio 算子样例工程在如下路径：MindStudio→samples→tbe_operator_sample。

右击工程名 tbe_operator_sample，选择 New→Operator 命令，弹出如图 3-7 所示界面，在 Operator Type 栏输入算子类型名称 sqrt，在 Plugin Framework 栏选择 TensorFlow，即算子所在模型文件的 AI 框架类型，Compute Unit 栏是选择开发 TBE 算子还是 AI CPU 算子，Unit Type 栏用于根据实际昇腾 AI 处理器版本通过下拉列表选择算子计算单元，单击 OK 按钮，完成算子工程的创建。

关于算子名称需要采用大驼峰的命名方式，即采用大写字母区分不同的语义，当前需要将大写字母转换为下画线加小写字母(首字符只转换为小写字母不带下画线)，作为算

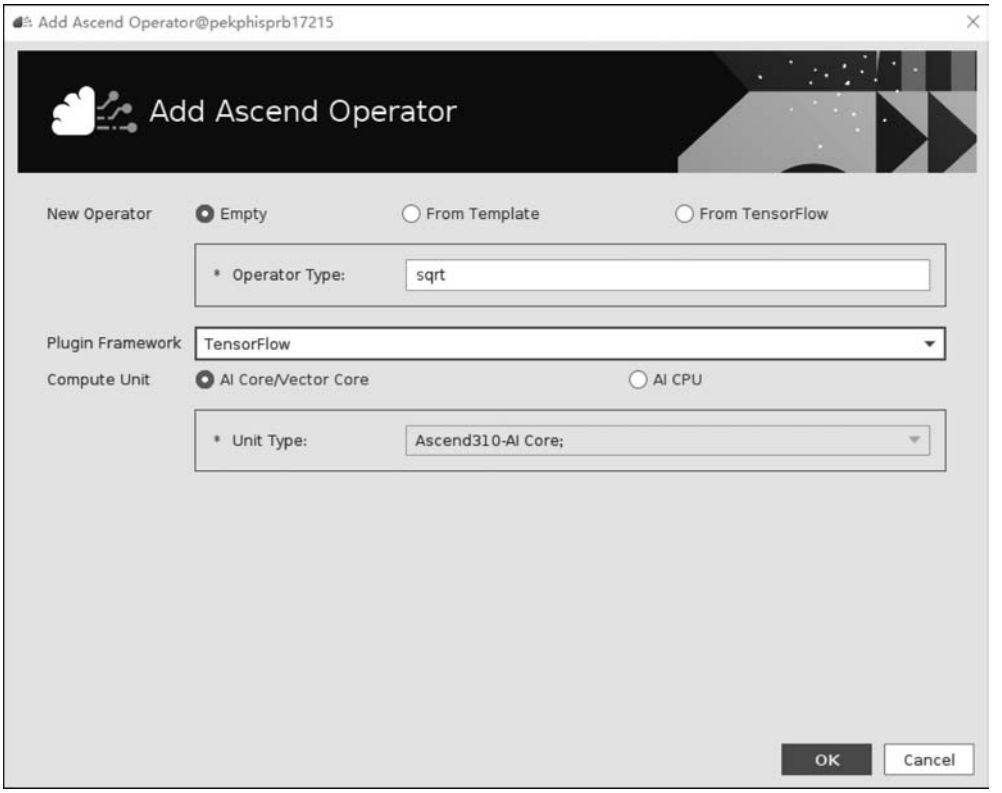

图 3-7　MindStudio 算子开发界面

子文件名称和算子函数名称。比如算子类型命名为 AbcDef，那么算子文件名和函数名就为 abc_def。

MindStudio 会自动生成算子框架代码，在工程 tbe→impl 目录下，自动生成了一个名为 sqrt.py 的 Python 程序，里面包含以下两个函数：

```
def sqrt(input_x, output_y, kernel_name = "sqrt"):
def sqrt_compute(input_x, output_y, kernel_name = "sqrt"):
```

为了保证函数功能的单一性，函数 sqrt 主要是进行参数检查、实现主干逻辑，sqrt_compute 函数实现具体的算子计算逻辑，被函数 sqrt 调用。

3. 导入 Python 模块

进行 TBE DSL 算子开发时，首先需要在算子实现文件中导入昇腾 CANN 软件栈提供的 Python 模块，代码示例如下。

```
import tbe.dsl
from tbe import tvm
from tbe.common.utils import para_check
```

```
from tbe.common.utils import shape_util
```

对这段代码做如下解释：

（1）tbe.dsl：引入 TBE 支持的特定域语言接口，包括常见的运算 vmuls、vadds、matmul 等，以及编译接口。具体的接口定义可查看 ATC 工具安装路径下 atc/python/site-packages/tbe/dsl/目录下的 Python 函数。

（2）tbe.tvm：引入 TVM 后端代码生成机制。

（3）tbe.common.utils.para_check：提供了通用的算子参数校验接口。

（4）tbe.common.utils.shape_util：提供了通用的处理算子形状的接口。

4. 算子函数定义

算子接口函数中包含了算子输入信息、算子输出信息以及内核名称，函数的声明信息需要与算子信息定义文件中的信息对应（算子信息定义将在后面章节进行介绍）。算子函数定义如下：

```
def operationname(input_x1, input_x2, output_y, attribute1 = None, attribute2 = None,...,
kernel_name = "KernelName", impl_mode = "high_performance")
```

参数说明：

（1）operationname：算子接口函数名称，请与算子实现文件名称保持一致。

（2）input_x1，input_x2：算子的输入张量，每个张量需要采用字典的形式进行定义，包含参数 shape、ori_shape、format、ori_format 与 dtype 信息，例如：

```
dict input_x1 = {'shape' : (2,2), 'ori_shape' : (2,2), 'format': 'ND', 'ori_format':'ND', 'dtype' :
'float16'} #输入张量的名称、个数及顺序，需要与算子信息库定义保持一致
```

之所以输入的张量要采用上面这种形式定义，其原因是 TBE DSL 只是定义算子过程，并不执行，最终会生成 CCE 代码，在 TBE 中是通过 tvm.placeholder 进行数据占位，这时并没有真正的数据，只是告诉 TVM 占位的形状和数据类型，最后通过 TVM 机制，把数据地址作为生成的 CCE 代码的输入，placeholder 相当于定义了一个位置，这个位置中的数据在程序运行时再指定。

（3）output_y：算子的输出张量，包含参数 shape 和 dtype 等信息，采用字典形式，此字段为预留位。

（4）attribute1，attribute2，…：算子的属性，此处需要为算子的属性赋默认值，算子属性的名称、个数与顺序需要与算子信息库中的定义保持一致。若算子无相关属性信息，此参数忽略。

（5）kernel_name：算子在内核中的名称（生成的二进制文件与算子描述文件的名称），由用户自定义，为了保持唯一，只能是字母、数字、“_”的组合，且必须是字母或者“_”开头，长度小于或等于 200 个字符。

（6）impl_mode（可选）：为字符串（String）类型，算子运行时可选择精度优先还是性

能优先模式，该字段仅影响输入为 float32 类型数据时的精度与性能。有 high_precision 与 high_performance 两种取值，默认值为 high_performance。

用户在进行算子接口函数声明时可使用装饰器函数 check_op_params 或者 check_input_type 对算子参数进基本的校验。其中 check_op_params 校验算子输入/输出是否满足必选与可选的要求，check_input_type 校验算子的参数类型是否合法。例如：

```
@para_check.check_op_params(para_check.REQUIRED_INPUT,para_check.REQUIRED_OUTPUT,para_
check.KERNEL_NAME)
@para_check.check_input_type(dict, dict, str)
def sqrt(x, y, kernel_name = "sqrt")
```

这里装饰器函数 check_input_type 会检查参数 x 的类型是不是字典，其他参数也是如此。

5. 算子函数实现

完成算子函数声明后，就要具体实现算子接口定义 sqrt 函数和 sqrt-compute 函数。首先在算子接口定义函数中，获取算子输入张量的形状(shape)以及数据类型(dtype)，并对其及其他属性信息进行校验，一些基本的校验检查，框架代码会自动生成：

```
para_check.check_shape_rule(shape)              # 校验算子的 shape 参数
para_check.check_shape_size(shape)              # 校验算子输入 shape 参数的大小
para_check.check_kernel_name(kernel_name)       # 校验算子的 kernel_name 参数
```

除了上面这些校验之外，还可以根据新开发算子本身的需要，新增相关的校验，比如，如果开发的算子只支持 float16 和 float32，就可以对输入的数据类型进行校验。代码如下：

```
para_check.check_dtype_rule(input_dtype, ("float16", "float32"))
```

校验完成之后，根据 shape 参数与 dtype 参数定义输入张量的张量占位符，使用 TVM 的 placeholder 接口对输入张量进行占位，返回一个张量对象，如前所述，此位置中的数据在程序运行时才被指定。在算子接口定义 sqrt 函数中调用 sqrt_compute()函数，输入张量为 tvm.placeholder 定义的占位张量，其他为算子接口定义函数传送的参数，具体如下(这些代码都为框架自动生成的，可能需要根据算子的实际需要进行修改)：

```
data_x = tvm.placeholder(shape, name = "data_input", dtype = input_dtype)
res = sqrt_compute(data_x, output_y, kernel_n6ame)
```

在 sqrt-compute 函数中，完成算子的计算过程。计算过程的实现主要根据算子分析中的 TBE DSL API 进行代码开发。再回顾开平方的计算逻辑：

$$y = \sqrt{x} = \exp(0.5 \times \ln(x))$$

(1) 调用 tbe.dsl.vlog(raw_tensor),计算 ln(x)。

(2) 调用 tbe.dsl.vmuls(raw_tensor,scalar),计算 0.5×ln(x)。

(3) 调用 tbe.dsl.vexp(raw_tensor)接口得到最终结果。

完整代码如程序清单 3-1 所示。

程序清单 3-1　平方根计算函数

```
def sqrt_compute(x, y, kernel_name = "sqrt"):
    log_val = tbe.dsl.vlog(x)
    const_val = tvm.const(0.5, "float32")
    mul_val = tbe.dsl.vmuls(log_val, const_val)
    res = tbe.dsl.vexp(mul_val)
    return res
```

6. 自动调度

至此,算子的基本逻辑都开发完成了,当定义完计算逻辑后,需要在算子接口实现函数中实现调度与编译。通过调用 auto_schedule 接口,便可以自动生成相应的调度,此处通过 TVM 的打印机制可以看到相应计算的中间表示。配置信息包括是否需要打印 IR、是否编译,以及算子内核名及输入、输出张量的列表(张量列表需要根据算子实际情况进行修改,框架代码默认是单输入单输出,如果不是这样就需要修改)。具体代码如程序清单 3-2 所示(都是框架自动生成的)。

程序清单 3-2　自动调度代码

```
with tvm.target.cce():
    sch = tbe.dsl.auto_schedule(result)
config = {
    "print_ir": True,
    "need_build": True,
    "name": kernel_name,
    "tensor_list": [input_data, result]
    "bool_storage_as_bit":True
}
tbe.dsl.build(sch,config)
```

使用 auto_schedule 接口,自动生成相应的调度(schedule),auto_schedule 接口的参数为算子的输出张量。其中的参数说明如下。

(1) schedule:可以理解为描述的计算过程如何在硬件上高效执行,即把相关的计算和硬件设备上的相关指令对应起来。schedule 对象中包含一个“中间表示(IR)”,它用一种类似伪代码来描述计算过程,可以通过“print_ir”参数把它打印出来查看。

(2) “need_build”:表示是否进行编译并生成,默认是 True。

（3）“name”：编译生成的算子二进制文件名称，只能是字母、数字、“_”的组合，且必须是字母或者“_”开头，长度小于或等于 200 个字符。

（4）“tensor_list”：用于保存输入张量、输出张量，这个顺序需要严格按照算子本身的输入、输出数据顺序排列。注意：输入张量需要是 placeholder 接口返回的张量对象，此张量对象的内存地址不能被覆盖。例如："tensor_list"：[tensor_a，tensor_b，res]，tensor_a 与 tensor_b 是输入张量，res 为输出张量。

（5）“bool_storage_as_1bit”：bool 类型数据存储时是否按照 1 位存储。True 表示按照 1 位存储，False 表示按照 8 位进行存储，默认值为 True。当 tbe.dsl.vcmp(lhs，rhs，op，mode)接口的 mode(模式)为 bool 类型时，需要设置此参数为 False。

（6）tbe.dsl.build：根据调度和配置使用“tbe.dsl”提供的“build”接口来进行算子编译，算子编译过程会根据输入的数据形状、类别、算子参数等编译出专用内核，这个过程在生成模型时发生。

（7）sch：生成的算子计算调度对象。

（8）config：编译参数配置的映射。

编译完成后，会生成算子目标文件 *.o 与算子描述文件 *.json。

sqrt 算子接口函数完整的示例代码如程序清单 3-3 所示。

程序清单 3-3　sqrt 算子接口函数

```
@para_check.check_input_type(dict, dict, str)
def sqrt(x, y, kernel_name = "sqrt"):
    shape_x = shape_util.scalar2tensor_one(x.get("shape"))
    para_check.check_kernel_name(kernel_name)
    para_check.check_shape_rule(shape_x)
    para_check.check_shape_size(shape_x, SHAPE_SIZE_LIMIT)
    check_tuple = ("float16", "float32", "int32")
    x_dtype = x.get("dtype").lower()
    para_check.check_dtype_rule(x_dtype, check_tuple)

    data_x = tvm.placeholder(x.get("shape"), dtype = x_dtype, name = "data_x")
    res = sqrt_compute(data_x, y, kernel_name)

    # auto schedule
    with tvm.target.cce():
        schedule = tbe.dsl.auto_schedule(res)

    # operator build
    config = {"name": kernel_name,"tensor_list": [data_x, res]}
tbe.dsl.build(schedule, config)
```

7. 简单精度优化

绝对误差和相对误差是常用的衡量精度的概念，通俗地说，绝对误差是指真实值和实

际值的差值，相对误差是这个差值和真实值的比。用公式表示为：

绝对误差＝|真实值－实际值|

相对误差 ＝|真实值－实际值|/真实值

在昇腾 910 AI 处理器场景下，在 float16 的情况下，将入参的数据类型转成 float32 进行计算，可以提高中间计算过程的精度，从而提升最终结果的精度，尤其当中间计算过程较为复杂时效果比较明显。对于对精度要求较高的场景，可通过牛顿迭代、泰勒展开式的方式对计算公式进行变换。这里先介绍将数据类型转成 float32 进行计算的优化方法，其他方法后续章节进行详细介绍。

在 sqrt 算子接口函数中，sqrt_compute 函数调用 TBE DSL API tbe.dsl.cast_to 函数将输入数据的数据类型转换成 float32，当计算逻辑结束得到结果后，需要将结果数据的数据类型转换成 float16，具体实现如程序清单 3-4 所示。

程序清单 3-4　数据类型转换

```
def sqrt_compute(input_x, y , kernel_name = "sqrt"):
    dtype = input_x.dtype
    if dtype == "float16":
        input_x = tbe.dsl.cast_to(input_x, "float32")
    log_val = tbe.dsl.vlog(input_x)
    mul_val = tbe.dsl.vmuls(log_val, tvm.const(0.5, "float32"))
    res = tbe.dsl.vexp(mul_val)

    if dtype == "float16":
        res = tbe.dsl.cast_to(res, "float16")
return res
```

由于数据类型转换会有性能开销，因此如果 float16 类型的数据计算精度在可允许范围内，尽量不要转换数据类型。

最后在 Python 代码最下方添加 main 函数调用该算子，通过 MindStudio 编译算子实现文件，用于单算子代码的简单语法校验，代码如程序清单 3-5 所示。

程序清单 3-5　语法校验

```
if __name__ == '__main__':
    input_output_dict = {"shape": (5, 6, 7),"format": "ND","ori_shape": (5, 6, 7),
"ori_format": "ND", "dtype": "float16"}
    sqrt(input_output_dict, input_output_dict, kernel_name = "sqrt")
```

在编译界面右击 tbe/impl/sqrt.py，选择 Run‘sqrt’，编译算子。如果编译没有报错，且在当前目录“tbe/impl”下生成 kernel_meta 文件夹，该文件夹包括算子二进制文件 *.o 和算子描述文件 *.json(用于定义算子属性及运行时所需要的资源)，则表示算子代码能够编译运行。

8. 通过变换公式进行精度优化

在某些场景下，直接使用相关指令可能会出现算子精度不达标的情况，这种情况就需要通过变换公式来避免使用有精度问题的指令。

上面简单介绍将 fp16 数据转换成 fp32 进行计算，用高精度数据进行中间计算提升精度。当算子出现精度不达标的情况时，可以通过变换公式避免直接调用 API 引起的精度问题。下面还是以开平方（sqrt 算子）举例说明。

对于开平方根的近似解法，很自然就想到牛顿迭代法，其思想就是切线是曲线的线性逼近，牛顿迭代公式为：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

根据上式重写 sqrt 算子，核心代码见程序清单 3-6，通过测试发现 3 次迭代就能得到精度满足要求的结果。

程序清单 3-6　重写 sqrt 算子

```
def _sqrt(x, shape, dtype, iter = 3):
    # iter: 迭代次数
    y_last = tbe.dsl.vexp(tbe.dsl.vmuls(tbe.dsl.vlog(x), tvm.const(0.5)))
    for i in range(iter):
        y = tbe.dsl.vmul(x, tbe.dsl.vrec(y_last))
        y = tbe.dsl.vadd(y, y_last)
        y = tbe.dsl.vmuls(y, tvm.const(0.5, dtype))
        y_last = y
    return y
```

当然，牛顿迭代法只是方法之一，对于一些数学算子，常常采用泰勒级数用无限项连加式来拟合，这些相加的项由函数在某一点的导数求得，通过函数在自变量零点的导数求得的泰勒级数又叫作麦克劳林级数。在数学上，对于一个在实数或复数邻域上，以实数作为变量或以复数 α 作为变量、无穷可微的函数 $f(x)$，它的泰勒级数是以下这种形式的幂级数：

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

对于函数 $f(x)$，虽然它们的展开式会收敛，但函数与其泰勒级数也可能不相等。在实际应用中，泰勒级数需要截断，只取有限项，可以根据误差的上限选取泰勒级数展开的阶数。函数 $f(x)$在进行泰勒级数展开的时候，在展开点附近拟合误差比较小，一般定义域离展开点越远，收敛越慢，误差越大。为了简化级数表达式，一般函数往往采用在 $\alpha=0$ 处进行泰勒级数展开（即麦克劳林级数展开）来拟合，例如：

$$\text{指数函数：} e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$\text{自然对数：} \ln(x+1) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} x^n \quad \forall x \in (-1,1]$$

而有些函数在某些区间的收敛十分缓慢，如图 3-8 所示为 arcsin 函数的拟合曲线，当 x 接近 1 的时候拟合误差很大，无法直接采用麦克劳林级数展开来近似表示（由于收敛过慢，即使提高展开阶数也达不到精度要求），因此需要针对这类函数进行分区间拟合。因此针对这种情况，为了达到精度的要求，可以采用分区间在不同的展开点进行泰勒级数展开，或者将精度达标的展开区间的拟合结果通过数学公式映射到其他区间。

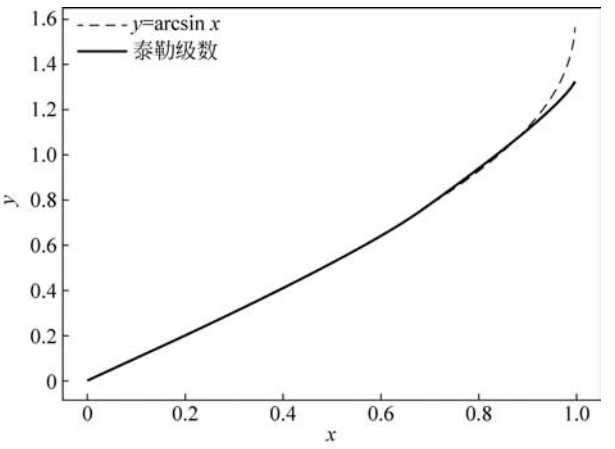

图 3-8　arcsin 函数的拟合曲线

下面以 arcsin$x$ 为例简单介绍三种解决拟合精度问题的方法，反正弦函数 arcsin$x$ 是正弦函数 sin$x$ 将值域限制在[$-\pi/2,\pi/2$]的反函数，定义域区间为[$-1,1$]，关于原点对称，为奇函数。其麦克劳林展开的无穷级数表示为：

$$\arcsin x=\sum_{k=0}^{\infty}\binom{-\frac{1}{2}}{k}(-1)^k\frac{x^{2k+1}}{2k+1}=x+\frac{1}{6}x^3+\frac{3}{40}x^5+\frac{5}{112}x^7+\cdots$$

除此之外，它还满足：

$$\arcsin(2x)=2\arcsin\left(\sqrt{\frac{1-\sqrt{1-4x^2}}{2}}\right)$$

$$\arcsin x=\frac{\pi}{2}-\arcsin\sqrt{1-x^2}$$

方法 1：区间映射。

由于 $y=\arcsin x$ 在零点附近收敛很快，拟合精度很高，因此可以考虑利用公式将零点附近的区间映射到 $x=1$ 附近的区间。经过分析，公式 $\arcsin x=\frac{\pi}{2}-\arcsin\sqrt{1-x^2}$ 可用于进行区间映射。将区间分界点选在 $x=\sqrt{2}/2$ 的时候，正好可以利用上述公式将区间[$0,\sqrt{2}/2$]上的麦克劳林级数展开拟合结果映射到区间[$\sqrt{2}/2,1$]。

但当区间[$0,\sqrt{2}/2$]上的麦克劳林级数展开阶数小于等于 13 阶（7 个系数）时，x 在 0.68～0.73 存在精度问题（即不满足万分之一的相对误差要求）。

方法 2：提高泰勒级数展开阶数。

一般情况下，可以直接通过 MATLAB 或者 OCTAVE 等工具仿真最少需要用多少阶数展开可以达到精度要求，虽然展开阶数越多精度越高，但也会带来更多的乘加操作使得运行性能下降，因此需要在满足精度的条件下选择更小的展开阶数。

在进行 OCTAVE 仿真的时候，可以看到当麦克劳林级数展开 15 阶(8 个系数)时，误差全都小于万分之一。

但针对有些函数在某些区间难以收敛，即使提高泰勒级数展开阶数也无法满足精度要求，可以再次通过区间映射方法将精度较高区间的计算结果映射到精度不达标区间(比如可以通过公式 $\arcsin 2x = 2\arcsin\sqrt{\frac{1-\sqrt{1-4x^2}}{2}}$ 将区间[0，0.5]的麦克劳林级数展开结果映射到区间)[0.5，$\sqrt{2}/2$]，或者在不同的展开点进行泰勒级数展开。接下来就介绍分区间泰勒级数展开的方法。

方法 3：分区间泰勒级数展开。

当对 $\arcsin x$ 在 $a!=0$，即 a 不等于 0 处进行泰勒级数展开时，其泰勒级数展开的无穷级数可以表示为：

$$\begin{aligned}\arcsin x = {} & \arcsin a + (1-a^2)^{-\frac{1}{2}}(x-a) + \frac{1}{2}a(1-a^2)^{-\frac{3}{2}}(x-a)^2 + \\ & \frac{1}{6}[(1-a^2)^{-\frac{3}{2}} + 3a^2(1-a^2)^{-\frac{5}{2}}](x-a)^3 + \\ & \frac{1}{24}[9a(1-a^2)^{-\frac{5}{2}} + 15a^3(1-a^2)^{-\frac{7}{2}}](x-a)^4 + \\ & \frac{1}{120}[9(1-a^2)^{-\frac{5}{2}} + 90a^2(1-a^2)^{-\frac{7}{2}} + 105a^4(1-a^2)^{-\frac{9}{2}}](x-a)^5 + \cdots\end{aligned}$$

可以考虑在区间[0，0.5]继续采用麦克劳林级数展开来拟合，针对直接采用麦克劳林级数展开拟合精度较差的区间[0.5，$\sqrt{2}/2$]，用 $a=0.6$ 处的泰勒级数展开来近似，区间[$\sqrt{2}/2$，1]的结果依旧由将区间[0，$\sqrt{2}/2$]的麦克劳林级数展开结果映射得到。

3.2.4　算子原型定义与算子信息定义

1. 算子原型定义

算子原型定义规定了在昇腾 AI 处理器上可运行算子的约束，主要体现算子的数学含义，包含定义算子输入、输出和属性信息，基本参数的校验和形状(shape)的推导，原型定义的信息会被注册到图编译器的算子原型库中。当网络模型生成时，图编译器会调用算子原型库的校验接口进行基本参数的校验，校验通过后，会根据原型库中的推导函数推导每个节点的输出形状与数据类型(dtype)，进行输出张量的静态内存的分配。

算子原型库在整个网络模型生成流程中的作用如图 3-9 所示。

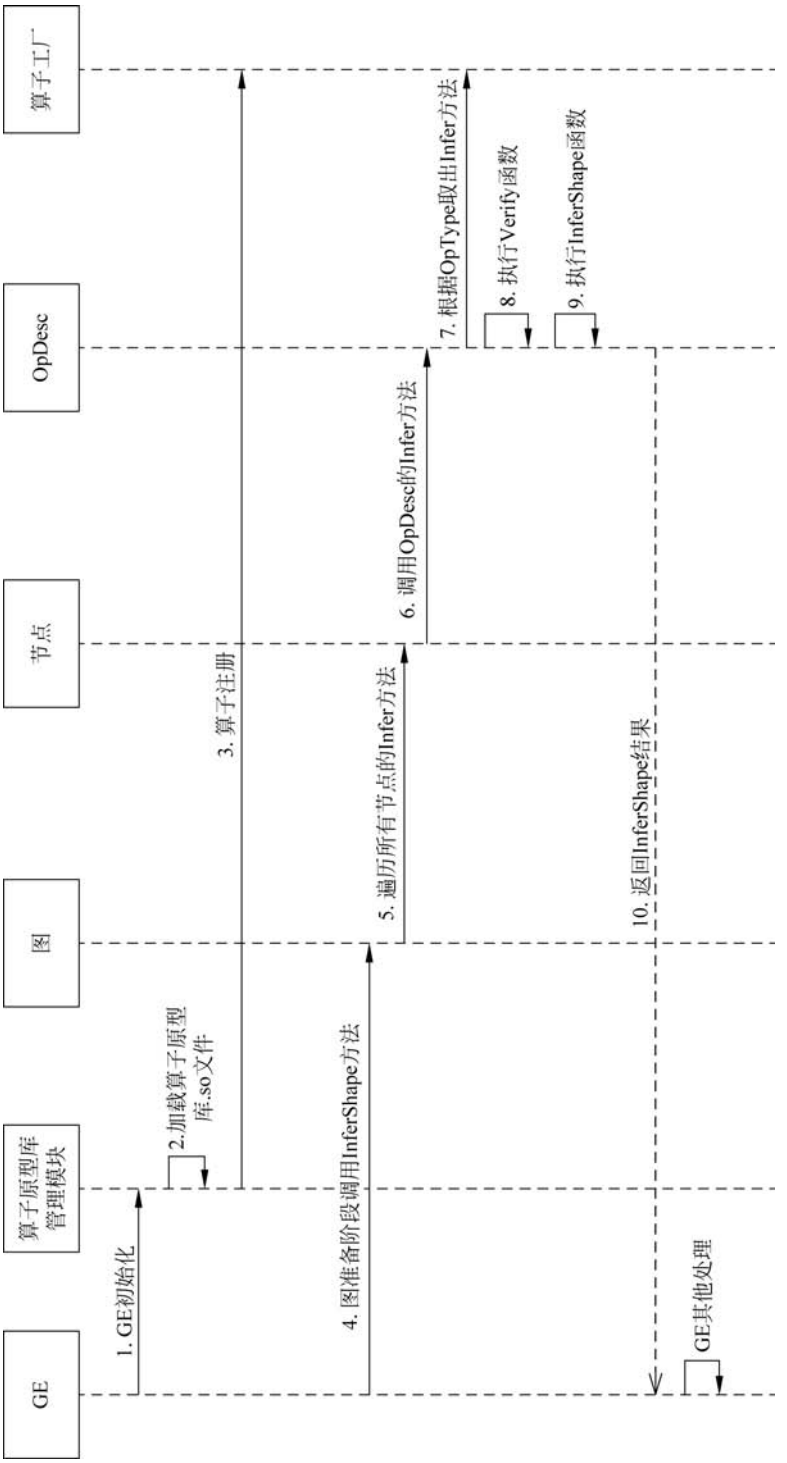

图 3-9　整个网络模型生成流程

算子的原型定义(IR)需要在算子工程目录下的文件,如/op_proto/算子名称.h 和/op_proto/算子名称.cc中实现,MindStudio 会自动生成框架代码。

1) 算子 IR 头文件.h 注册代码实现

(1) 宏定义:使用如下语句进行算子 IR 注册宏的定义,宏名称固定为 GE_OP_OPERATORTYPE_H,OPERATORTYPE 为 REG_OP(OpType)语句中 OpType 的大写形式。

```
#ifndef GE_OP_OPERATORTYPE_H                    //条件编译
#define GE_OP_OPERATORTYPE_H                    //进行宏定义
```

(2) 包含头文件:在算子 IR 实现文件的头部使用预编译命令“#include”将算子注册的头文件包含到算子 IR 实现的文件中。

```
#include "graph/operator_reg.h"
```

operator_reg.h 存在于 ATC 工具安装路径/include/graph/下,包含此头文件,可使用算子类型注册相关的函数、宏、结构体等。

(3) 原型注册:提供 REG_OP 宏,以“.”连接 INPUT、OUTPUT、ATTR 等接口注册算子的输入、输出和属性信息,最终以 OP_END_FACTORY_REG 接口结束,完成算子的注册。

其中输入、输出的描述信息顺序需要与算子实现中定义的信息保持一致,ATTR 的顺序可变。

注册代码如程序清单 3-7 所示。

程序清单 3-7　注册代码

```
namespace ge{
    REG_OP(OpType)                          //算子类型名称
    .INPUT(x1, TensorType({ DT_FLOAT, DT_INT32 }))
    .INPUT(x2, TensorType({ DT_FLOAT, DT_INT32 }))
    // .DYNAMIC_INPUT(x, TensorType{DT_FLOAT, DT_INT32})
    // .OPTIONAL_INPUT(b, TensorType{DT_FLOAT})
    .OUTPUT(y, TensorType({ DT_FLOAT, DT_INT32 }))
    // .DYNAMIC_OUTPUT(y, TensorType{DT_FLOAT, DT_INT32})
    .ATTR(x, Type, DefaultValue)
    // .REQUIRED_ATTR(x, Type)
    // .GRAPH(z1)
    // .DYNAMIC_GRAPH(z2)
    .OP_END_FACTORY_REG(OpType)
}
```

下面对上述代码做部分说明。

① REG_OP(OpType):OpType 为注册到昇腾 AI 处理器的自定义算子库的算子类

型，需要与适配开发框架（TensorFlow 框架）中 REGISTER_CUSTOM_OP("OpType") 定义的算子类型名称保持一致。

② INPUT(x,TensorType({DT_FLOAT,DT_UINT32,…}))：注册算子的输入信息。

x：宏参数，算子的输入名称，用户可自定义。

TensorType({ DT_FLOAT,DT_UINT8,…})：“{ }”中为此输入支持的数据类型的列表。

若算子有多个输入，每个输入需要使用一条 INPUT(x,TensorType({DT_FLOAT,DT_UINT32,…}))语句进行描述。

③ DYNAMIC_INPUT(x,TensorType{DT_FLOAT,DT_INT32,…})：算子为动态多输入场景下的输入信息注册。

x：宏参数，算子的输入名称，图运行时，会根据输入的个数自动生成 x0,x1,x2,…，序号依次递增。

④ OPTIONAL_INPUT(x,TensorType{DT_FLOAT,…})：若算子输入为可选输入，可使用此接口进行算子输入的注册。

x：宏参数，算子的输入名称。

⑤ OUTPUT(y,TensorType({DT_FLOAT,DT_UINT32,…}))：注册算子的输出信息。

y：宏参数，算子的输出名称，用户可自定义。

若算子有多个输出，每个输出需要使用一条 OUTPUT(y,TensorType({DT_FLOAT,DT_UINT32,…}))语句进行注册。

⑥ DYNAMIC_OUTPUT(y,TensorType{DT_FLOAT,DT_INT32})：算子为动态多输出场景下的输出信息注册。

⑦ ATTR(x,Type,DefaultValue)：注册算子的属性，包括算子的属性名称、属性类型以及属性值的默认值，当用户不设置算子对象的属性值时需要使用默认值。ATTR 接口中 Type 的取值与对应的属性类型请参见原型定义接口（REG_OP）。

例如：ATTR(mode,Int,1)，表示注册属性名称为 mode，属性类型为整型，默认值为 1。

若算子有多个属性，每个属性需要使用一条 ATTR(x,Type,DefaultValue)语句或者 REQUIRED_ATTR (x,Type)语句进行注册。

⑧ REQUIRED_ATTR (x,Type)：注册算子的属性，包括算子的属性名称与属性类型，无默认值，用户必须设置算子对象的属性值。此接口中 Type 的取值与对应的属性类型请参见原型定义接口（REG_OP）。

⑨ GRAPH(z1)：注册算子中包含的子图信息，输入 z1 为子图名称，一般用于控制类算子（分支算子/循环算子等）。

注册完成后，会自动生成子图相关的接口，用户获取子图名称，获取或设置子图描述信息等，用户可使用生成的相关接口进行 IR 模型的构建。对于同一个算子，注册的算子子图名称需要保持唯一。

⑩ DYNAMIC_GRAPH(z2)：注册动态算子子图信息，输入 z2 为子图名称，一般用于控制类算子(分支算子/循环算子等)。

⑪ OP_END_FACTORY_REG(OpType)：结束算子注册。OpType 与 REG_OP(OpType)中的 OpType 保持一致。

sqrt 算子头文件.h 的注册代码见程序清单 3-8。

程序清单 3-8　sqrt 算子头文件.h 的注册代码

```
#ifndef GE_OP_SQRT_H
#define GE_OP_SQRT_H
#include "graph/operator_reg.h"
namespace ge {

REG_OP(sqrt)
    .INPUT(x, TensorType({DT_FLOAT16,DT_FLOAT,DT_DOUBLE,DT_INT8,
DT_INT16,DT_INT32,DT_INT64}))
    .OUTPUT(y, TensorType({DT_FLOAT16,DT_FLOAT,DT_DOUBLE,DT_INT8,
DT_INT16,DT_INT32,DT_INT64}))
    .OP_END_FACTORY_REG(sqrt)
}
#endif //GE_OP_SQRT_H
```

2) 算子 IR 定义的.cc 文件注册代码实现

算子 IR 定义的.cc 文件主要实现如下两个功能：一是对算子参数的校验，实现程序健壮性并提高定位效率；二是根据算子的输入张量描述、算子逻辑及算子属性，推理出算子的输出张量描述，包括张量的形状、数据类型及数据排布格式等信息。这样算子构图准备阶段就可以为所有的张量静态分配内存，避免动态内存分配带来的开销。

首先是实现 InferShape 方法，算子 IR 中 InferShape 的定义可以使用如下接口：

```
IMPLEMT_COMMON_INFERFUNC(func_name)
```

会自动生成的一个类型为 Operator 类的对象 op，可直接调用 Operator 类接口进行 InferShape 的实现。若 InferShape 方法具有通用性，可被多个算子的原型实现调用，可选择此接口实现。其中的 func_name 是自定义的名称。

如果输出的张量形状、数据类型、排布格式和输入是一样的，可以将输入描述直接赋给输出描述，对于 sqrt 算子来说，就是这样的，其实现代码如程序清单 3-9 所示。

程序清单 3-9　输出描述

```
IMPLEMT_COMMON_INFERFUNC(SqrtInferShape)
{
    TensorDesc tensordesc_output = op.GetOutputDescByName("y");
    tensordesc_output.SetShape(op.GetInputDescByName("x").GetShape());
```

```
    tensordesc_output.SetDataType(op.GetInputDescByName("x").GetDataType());
    tensordesc_output.SetFormat(op.GetInputDescByName("x").GetFormat());

    (void)op.UpdateOutputDesc("y", tensordesc_output);

    return GRAPH_SUCCESS;
}
```

其次是实现 Verify 方法，算子 Verify 函数的实现使用如下接口：

```
IMPLEMT_VERIFIER (OpType, func_name)
```

传入的 OpType 为基于 Operator 类派生出来的子类，会自动生成一个类型为此子类的对象 op，可以使用子类的成员函数获取算子的相关属性。

(1) OpType：自定义算子的类型。

(2) func_name：自定义的 Verify 函数的名称。

Verify 函数主要校验算子内在关联关系，例如对于多输入算子，多个张量的数据类型参数(dtype)需要保持一致，此时需要校验多个输入的参数 dtype，其他情况下，dtype 不需要校验。

比如 Pow 算子，要求输入 x 和 y 的数据类型必须一致，实现样例如程序清单 3-10 所示。

程序清单 3-10　数据类型

```
IMPLEMT_VERIFIER(Pow, PowVerify) {
    DataType input_type_x = op.GetInputDesc("x").GetDataType();
    DataType input_type_y = op.GetInputDesc("y").GetDataType();
    if (input_type_x != input_type_y) {
      return GRAPH_FAILED;
    }
    return GRAPH_SUCCESS;
}
```

最后是注册 InferShape 方法与 Verify 方法。

调用 InferShape 注册宏与 Verify 注册宏完成 InferShape 方法与 Verify 方法的注册，具体代码如下：

```
COMMON_INFER_FUNC_REG(OpType, func_name);
VERIFY_FUNC_REG (OpType, func_name);
```

func_name 即为 IMPLEMT_COMMON_INFERFUNC(func_name)与 IMPLEMT_VERIFIER(OpType,func_name)函数中的 func_name。

2. 算子信息定义

算子信息库作为算子开发的交付件之一，主要体现算子在昇腾 AI 处理器上的具体实现规格，包括算子支持输入、输出的数据类型(dtype)、数据排布格式(format)以及输入形状(shape)等信息。网络运行时，图编译器会根据算子信息库中的算子信息做基本校验，选择参数 dtype、format 等信息，并根据算子信息库中的信息找到对应的算子实现文件进行编译，用于生成算子二进制文件。

算子用户需要通过配置算子信息库文件，将算子在昇腾 AI 处理器上相关实现信息注册到算子信息库文件中。

算子信息库文件的路径为：自定义算子工程目录下的/tbe/op_info_cfg/ai_core/${soc_version}/xx.ini。

其中，${soc_version}表示昇腾 AI 处理器的版本。对于昇腾 910 系列处理器，统一使用“ascend910”。需将"xx.ini"文件的文件名的大写字母转换为小写字母，如 Sqrt 改为 sqrt。

对于开平方算子，它的算子信息定义见程序清单 3-11(MindStudio 会自动生成默认代码)：

程序清单 3-11　算子信息定义

```
[Sqrt]
input0.name = x
input0.dtype = float16,float,int8,int16,int32,int64
input0.paramType = required
input0.format = ND,ND,ND,ND,ND,ND
output0.name = y
output0.dtype = float16,float,int8,int16,int32,int64
output0.paramType = required
output0.format = ND,ND,ND,ND,ND,ND
opFile.value = sqrt
opInterface.value = sqrt
```

3.2.5　算子适配插件开发与算子编译及部署

1. 算子适配插件开发

如果是基于第三方框架 TensorFlow 进行自定义算子开发的场景，开发人员完成自定义算子的实现代码后，需要进行适配插件的开发，将基于第三方框架的算子映射成适配昇腾 AI 处理器的算子，将算子信息注册到图编译器中。基于 TensorFlow 框架开发的神经网络模型在昇腾处理器上运行时，首先会加载并调用图编译器中的插件信息，将原始框架网络中的算子进行解析并映射成昇腾 AI 处理器中的算子。

原始框架为 TensorFlow 的自定义算子注册代码，sqrt 算子的插件代码如程序清单 3-12 所示。

程序清单 3-12　sqrt 算子的插件代码

```
#include "register/register.h"
namespace domi
{
REGISTER_CUSTOM_OP("OpType")
    .FrameworkType(TENSORFLOW)
    .OriginOpType("OriginOpType")
    .ParseParamsByOperatorFn(ParseParamByOpFunc)
    .ImplyType(ImplyType::TVM);
}
```

下面对部分代码说明如下。

（1）REGISTER_CUSTOM_OP：算子注册到图编译器的算子类型。

（2）FrameworkType(TENSORFLOW)：原始框架类型为 TensorFlow。

（3）OriginOpType：算子在 TensorFlow 框架中的类型。

（4）ParseParamsByOperatorFn：用来注册解析模型的函数，若原始 TensorFlow 框架算子属性与昇腾 AI 处理器中算子属性一一对应（属性个数、属性名称与属性含义一致），可直接使用自动映射回调函数 AutoMappingByOpFn 自动实现映射。若原始 TensorFlow 框架算子属性与昇腾 AI 处理器中算子属性无法一一对应，比如针对 Conv2DBackpropInput 算子，strides 属性无法直接使用 TensorFlow 的对应算子中的 strides 属性，需要重新计算。所以需要在回调函数 ParseParamByOpFunc 中进行对应的代码实现。

2. 算子编译

将自定义算子工程编译生成自定义算子安装包 custom_opp_Target OS_Target Architecture.run。具体编译内容包括将算子插件实现文件、算子原型定义文件、算子信息定义文件分别编译成算子插件、算子原型库、算子信息库。编译过程如图 3-10 所示。

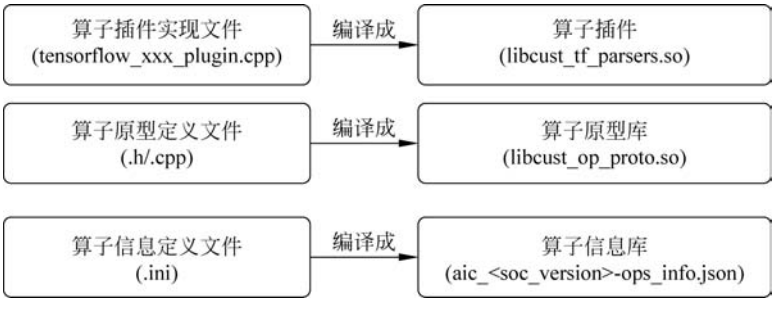

图 3-10　编译过程

编译时，在 MindStudio 工程界面选中算子工程，之后单击顶部菜单栏的 Build→Edit Build Configuration…命令，进入编译配置界面，如图 3-11 所示。

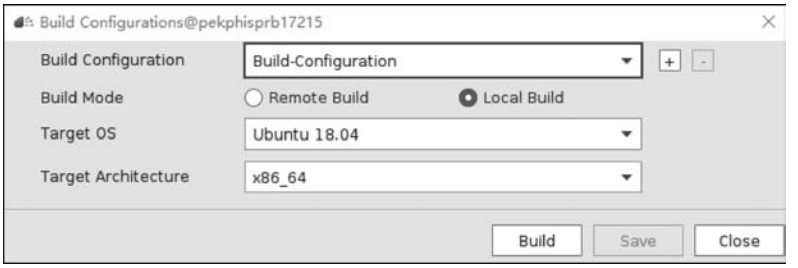

图 3-11　编译配置界面

编译配置界面中的参数说明如表 3-4 所示。

表 3-4　编译配置界面中的参数说明

参　数	说　明
Build Configuration	编译配置名称，默认为 Build-Configuration
Build Mode	编译方式。Remote Build：远端编译(远端编译需要 g++版本为 7.5.0)。 Local Build：本地编译。算子工程在 MindStudio 安装服务器进行编译，方便用户通过编译日志快速定位到 MindStudio 的实现代码所在位置，从而快速定位问题。此种方式下需要配置交叉编译环境
SSH Connection	在 Remote Build 模式下显示该配置。从下拉列表选择 SSH 配置信息，若未添加配置信息，请单击 ⊞ 添加
Target OS	在 Local Build 模式下显示该配置。针对 Ascend EP，选择昇腾 AI 处理器所在硬件环境的 Host 侧的操作系统，比如 CentOS 7.6 或 EulerOS 2.5。 针对 Ascend RC，选择板端环境的操作系统，比如 Ubuntu 18.04
Target Architecture	在 Local Build 模式下显示该配置。选择 Target OS 的操作系统架构，比如 x86_64 或 Arm64

最后单击 Build 按钮进行工程编译。在界面最下方的窗口查看编译结果，并在算子工程的 cmake-build 目录下生成自定义算子安装包 custom_opp_Target OS_Target Architecture.run。

3. 算子部署

算子部署指将算子编译生成的自定义算子安装包(*.run)部署到 OPP 算子信息库中。算子部署可以是本地部署或远程部署。这里先介绍本地部署。在 MindStudio 工程界面菜单栏中依次选择 Ascend→Deploy 命令，随后弹出算子部署界面。在弹出的界面中选择 Deploy Locally，并单击 Deploy 按钮。在下方 Output 选项卡出现如图 3-12 所示的信息，代表自定义算子部署成功。

自定义算子包安装成功后，会将自定义算子部署在 Ascend-cann-toolkit 安装目录

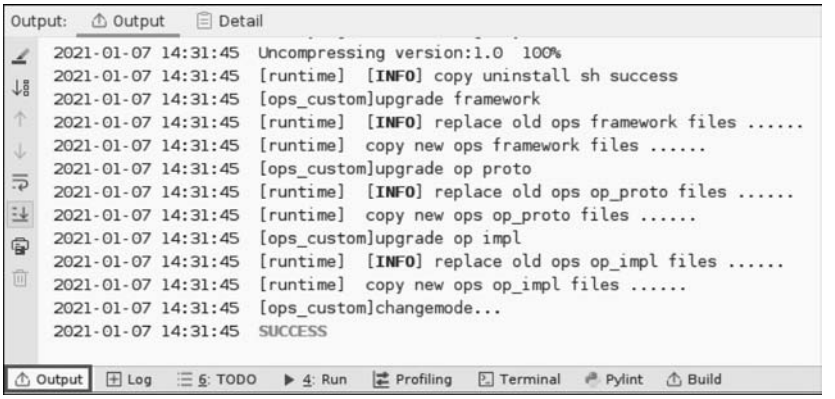

图 3-12 自定义算子部署成功

/ascend-toolkit/{version}/{arch}-linux/opp 下的对应文件夹中。

也可以将自定义算子安装包 custom_opp_Targert OS_Target Architecture.run 部署到昇腾 AI 处理器所在硬件环境的算子库中，为后续算子在网络中运行构造必要条件。

远程部署操作的步骤如下（如图 3-13 所示）。

（1）在 MindStudio 工程界面选中算子工程。

（2）选择顶部菜单栏的 Ascend→Deploy 命令，进入算子打包部署界面。选择 Deploy Remotely 单选按钮，在 SSH Connection 栏从下拉列表中选择 SSH 配置信息。

图 3-13 远程部署操作

配置环境变量时，在 MindStudio 中使用 Host 侧的运行用户在 Host 侧进行算子部署，进行算子部署执行前，需要在 Host 侧进行如下环境变量的配置（以运行用户在 Host 侧的 $HOME/.bashrc 文件中配置为例）。

```
export ASCEND_OPP_PATH=/home/xxx/Ascend/opp
```

/home/xxx/Ascend/为 OPP 组件（算子库）的安装路径，请根据实际情况配置。

执行命令使环境变量生效。

```
source ~/.bashrc
```

（3）选择算子部署的目标服务器，单击 Deploy 按钮。

（4）算子部署过程即算子工程编译生成的自定义算子安装包的安装过程，部署完成

后，算子被部署在 Host 侧算子库 OPP 对应文件夹中。

3.2.6　算子单元测试

基于 MindStudio 进行算子开发，用户可进行算子的单元测试（Unit Test，UT）。单元测试是开发人员进行算子代码验证的手段之一，主要目的是：

（1）单元测试测试算子代码的正确性，验证输入、输出结果与设计的一致性。

（2）单元测试侧重于保证算子程序能够正确运行，选取的场景组合应能覆盖算子代码的所有分支（一般来说覆盖率要达到 100%），从而降低不同场景下算子代码的编译失败率。

1. 接口介绍

1）OpUT 测试类定义

OpUT 为 UT 框架的基类，提供了测试用例定义及测试用例执行的接口，函数原型如下：

```
OpUT(op_type, op_module_name = None, op_func_name = None)
```

其中，op_type：算子的类型。

op_module_name：算子的 module 名称（即算子的实现文件名称和路径），例如 impl.sqrt（文件路径为：impl/sqrt.py）。默认值为 None，可根据 op_type 自动生成。

op_func_name：算子的接口名称，算子实现文件中的算子接口名。默认值为 None，可根据 op_type 自动生成。

（1）OpUT.add_case 接口：函数原型如下。

```
OpUT.add_case(support_soc = None, case = None)
```

support_soc：测试该用例是否支持对应的昇腾 AI 处理器。

case：测试用例，该参数为 dict 类型，示例见程序清单 3-13。

程序清单 3-13　配置信息

```
{
    "params": [
        {
            "shape": (32, 64),
            "ori_shape": (32, 64),
             "format": "ND",
            "ori_format": (32, 64),
            "dtype": "float16"
        },
        {
            "shape": (32, 64),
```

```
                "ori_shape": (32, 64),
                "format": "ND",
                "ori_format": (32, 64),
                "dtype": "float16"
            }
        ],
        "case_name": "test_sqrt_case_1",
        "expect": "success"
    }
```

该 dict 中 key 字段含义如下：

params：该字段在测试用例运行时传递给算子接口。

case_name：测试用例的名称，可选参数。若不设置，测试框架会自动生成用例名称，生成规则是：test_[op_type]_auto_case_name_[case_count]，例如：test_Sqrt_auto_case_name_1。

expect：期望结果。默认为期望“success”，也可以是预期抛出的异常，例如 RuntimeError。

（2）OpUT.add_precision_case 接口：用于添加算子编译＋精度测试的用例，函数原型如下：

```
OpUT.add_precision_case(support_soc = None, case)
```

case 示例如程序清单 3-14 所示。

程序清单 3-14　add_precision_case 接口 case 参数配置信息

```
{
    "params": [ … ],
    "case_name": "test_add_case_1",
    "calc_expect_func": np_add                                                    # 一个函数
    "precision_standard": precision_info.PrecisionStandard(0.001,0.001)    # 可选字段
}
```

该 dic 中字段含义如下，其中 params 与 add_case 接口类似。

calc_expect_func：期望结果生成函数。precision_standard：自定义精度标准，若不配置此字段，按照如下默认精度与期望数据进行比对。float16：双千分之一，即每个数据之间的误差不超过千分之一，误差超过千分之一的数据总和不超过总数据数的千分之一。float32：双万分之一，即每个数据之间的误差不超过万分之一，误差超过万分之一的数据总和不超过总数据数的万分之一。

（3）OpUT.run 接口：用于执行测试用例，使用 MindStudio 运行 UT 用例时，无须用户手工调用 OpUT.run 接口，函数原型如下：

```
OpUT.run(soc, case_name = None, simulator_mode = None, simulator_lib_path = None)
```

soc：执行测试用例的昇腾 AI 处理器。

case_name：指定执行的 case，配置为 add_case 接口及 add_precision_case 接口中的“case_name”。

simulator_lib_path：仿真库所在的路径。该路径结构如下：

```
simulator_lib_path/
  Ascend910/
      lib/
          libpv_model.so
          ...
  Ascend310/
      lib/
          libpv_model.so
          ...
```

2）BroadcastOpUT 测试类定义

BroadcastOpUT 继承了 OpUT 类，包含了 OpUT 类的能力。BroadcastOpUT 主要供双输入、单输出的 Broadcast 类型的算子进行测试用例的定义，例如 Add、Mul 等算子。BroadcastOpUT 为这类算子提供了更加便利的接口，例如，创建算子编译用例时，对于一些简单场景无须输入数据排布格式(format)等信息，函数原型如下：

```
BroadcastOpUT(op_type, op_module_name = None, op_func_name = None)
```

(1) BroadcastOpUT. add_broadcast_case 接口：用于添加算子编译的测试用例，测试算子是否支持相关规格，编译出“.o”文件，函数原型如下：

```
BroadcastOpUT.add_broadcast_case(self, soc, input_1_info, input_2_info,
output_info = None,expect = op_status.SUCCESS, case_name = None)
```

soc：测试该用例是否支持对应的昇腾 AI 处理器，支持的数据类型为 str、tuple 或者 list，tuple 或者 list 表示可以支持多个 SoC。

input_1_info：算子的第一个输入的信息，有两种形式：[dtype，shape，format，ori_shape，ori_format]和[dtype，shape，format]。采取后一种形式，ori_shape 与 ori_format 的取值与 shape、format 的取值相同。

input_2_info：算子的第二个输入的信息，与 input_1_info 含义相同。

output_info：默认为 None，不需要填写。

测试示例见程序清单 3-15。

程序清单 3-15　测试示例

```
ut_case.add_broadcast_case("all", ["float16", (32, 32), "ND"], ["float16", (32, 32), "ND"])
ut_case.add_broadcast_case("all", ["float16", (32, 32), "ND", (32, 32), "ND"], ["float16",
```

```
(32, 32), "ND", (32, 32), "ND"])
# 期望异常的用例
ut_case.add_broadcast_case("all", ["float16", (31, 32), "ND"], ["float16", (32, 32),
"ND"], expect = RuntimeError)
```

（2）BroadcastOpUT. add_broadcast_case_simple 接口，此接口较 BroadcastOpUT. add_broadcast_case 接口更加简化，函数原型为：

```
BroadcastOpUT.add_broadcast_case_simple(self, soc, dtypes, shape1, shape2,
expect = op_status.SUCCESS, case_name = None)
```

dtypes：需要测试的数据类型，填写多个数据，相当于一次添加了多个测试用例。

shape1：算子的第一个输入的形状。

shape2：算子的第二个输入的形状。

测试示例如下：

```
ut_case.add_broadcast_case_simple(["Ascend910", "Ascend310"], ["float16", "float32"],
(32, 32), (32, 32))
```

这个示例就相当于一次添加了 dtype= float16 和 dtype= float32 两类测试用例。

3）ElementwiseOpUT 测试类定义

ElementwiseOpUT 继承了 OpUT 类，包含了 OpUT 类的能力。ElementwiseOpUT 主要供单输入、单输出的 Elementwise 类型的算子进行测试用例的定义，例如 Abs、Square 等算子，函数原型和接口如下：

```
ElementwiseOpUT(op_type, op_module_name = None, op_func_name = None)
ElementwiseOpUT.add_elewise_case(self, soc, param_info,
expect = op_status.SUCCESS, case_name = None)
ElementwiseOpUT.add_elewise_case_simple(self, soc, dtypes, shape,
expect = op_status.SUCCESS, case_name = None)
```

4）ReduceOpUT 测试类定义

ReduceOpUT 继承了 OpUT 类，包含了 OpUT 类的能力。ReduceOpUT 主要供 Reduce 类型的算子进行测试用例的定义，例如 ReduceSum，ReduceMean 等算子。函数原型如下：

```
ReduceOpUT(op_type, op_module_name = None, op_func_name = None)
ReduceOpUT.add_reduce_case(self, soc, input_info, axes, keep_dim = False,
expect = op_status.SUCCESS, case_name = None)
ReduceOpUT.add_reduce_case_simple(self, soc, dtypes, shape, axes, keep_dim = False,
expect = op_status.SUCCESS, case_name = None)
```

2. 创建和运行单元测试用例

1）创建 UT 用例。

创建 UT 用例，有以下入口：右击算子工程根目录，选择 New Cases→UT Case 命

令；若已经存在了算子的 UT 用例，可以右击 testcases 目录或者 testcases→ut 目录，选择 New Cases→UT Case 命令，创建 UT 用例。

在弹出的算子选择界面，选择需要创建 UT 用例的算子 sqrt，单击 OK 按钮。

UT 用例创建完成后，会在算子工程根目录下生成 testcases 文件夹，目录结构如下：

```
├─── tbe_operator_sample                    //工程根目录
|    ├────  testcases
|    |    ├─── libs                         // gtest 框架，为第三方依赖，用户无须关注
|    |    ├────  ut
|    |    |    ├─── ops_test
|    |    |    |    ├─── sqrt
|    |    |    |    |    ├─── CMakeLists.txt        //用于编译可执行文件
|    |    |    |    |    ├─── test_sqrt_impl.py     //算子实现代码的测试用例文件
|    |    |    |    |    ├─── test_sqrt_proto.cc    //算子原型定义代码的测试用例文件
|    |    |    |    ├─── CMakeLists.txt             //用于编译可执行文件
|    |    |    |    ├─── test_main.cc               //测试用例调用总入口
|    |    |    ├ CMakeLists.txt
```

2）编写算子实现代码的 UT Python 测试用例

在 testcases/ut/ops_test/test_sqrt_impl.py 文件中，编写算子实现代码的 UT Python 测试用例，计算出算子执行结果，并取回结果和预期结果进行比较，来测试算子逻辑的正确性。

下面还是以 sqrt 算子为示例编写单元测试用例，开平方属于 Elementwise（元素积）操作，所以可以调用 ElementwiseOpUT 接口，具体参考代码见程序清单 3-16。

程序清单 3-16　UT Python 用例

```
import numpy as np
from op_test_frame.ut import ElementwiseOpUT  # 导入 UT 测试类，可根据算子类型选择使用
哪个测试类

# 实例化 UT 用例，ut_case 为 UT 框架关键字，不可修改
# 修改 op_func_name 的值，与 sqrt_impl.py 中注册算子函数名相同
ut_case = ElementwiseOpUT ("sqrt", op_func_name = " sqrt")

# 利用 numpy 的开平方函数 sqrt()实现生成期望数据的函数
def calc_expect_func(input_x, output_y):
    res = np.sqrt(input_x["value"])
    return [res, ]

# 添加测试用例
ut_case.add_precision_case("all", {
    "params": [{"dtype": "float16", "format": "ND", "ori_format": "ND",
"ori_shape": (32,), "shape": (32,),"param_type": "input"},
```

```
            {"dtype": "float16", "format": "ND", "ori_format": "ND",
"ori_shape": (32,), "shape": (32,),"param_type": "output"}],
    "calc_expect_func": calc_expect_func
})

# 若定义多个用例,定义多个 ut_case.add_precision_case 函数
ut_case.add_precision_case("all", {
    "params": [{"dtype": "float16", "format": "ND", "ori_format": "ND",
"ori_shape": (16,2), "shape": (16,2),"param_type": "input"},
            {"dtype": "float16", "format": "ND", "ori_format": "ND",
"ori_shape": (16,2), "shape": (16,2),"param_type": "output"},],
    "calc_expect_func": calc_expect_func
})
```

3) 编写算子原型定义的 UT C++测试用例

在 testcases/ut/ops_test/sqrt/test_sqrt_proto.cc 文件中,编写算子原型定义的 UT C++测试用例,用于定义算子实例、更新算子输入/输出并调用 InferShapeAndType 函数,最后验证 InferShapeAndType 函数执行过程及结果的正确性。

(1) 导入 gtest 测试框架和算子 IR 定义的头文件。UT 的 C++用例采用的是 gtest 框架,所以需要导入 gtest 测试框架;算子原型定义在原型定义头文件中,所以需要导入原型定义的 *.h 文件,见程序清单 3-17。

程序清单 3-17　导入原型定义的 *.h 文件

```
//导入 gtest 框架
#include <gtest/gtest.h>
//导入基础的 vector 类库
#include <vector>
//导入算子的 IR 定义头文件
#include "sqrt.h"
```

(2) 定义测试类。UT 的 C++用例采用的是 gtest 框架,所以需要定义一个类来继承 gtest 的测试类。测试类的名称可自定义,以 test 为后缀,MindStudio 会默认生成这个测试类,见程序清单 3-18。

程序清单 3-18　测试类

```
class SqrtTest : public testing::Test {
protected:
    static void SetUpTestCase() {
        std::cout << "sqrt test SetUp" << std::endl;
}
    static void TearDownTestCase() {
```

```
        std::cout << "sqrt test TearDown" << std::endl;
    }
};
```

（3）编写测试用例。每一个场景写一个测试用例函数，该用例中需要构造算子实例，包括算子名称、形状、数据类型。然后调用 InferShapeAndType 函数，并将推导出的参数 shape、dtype 与预期结果进行对比。

sqrt 算子的测试用例代码见程序清单 3-19。

程序清单 3-19　sqrt 算子的测试用例代码

```
TEST_F(SqrtTest, sqrt_test_case_1) {
    // 定义算子实例及输入参数 shape 和 type,以 TensorDesc 实例承载
    ge::op::Sqrt sqrt_op;  //sqrt 为算子的类型,需要与原型定义的 REG_OP(OpType)中的
OpType 保持一致
    ge::TensorDesc tensorDesc;
    ge::Shape shape({2, 3, 4});
    tensorDesc.SetDataType(ge::DT_FLOAT16);
tensorDesc.SetShape(shape);
    // 更新算子输入,输入的名称需要与原型定义 *.h 文件中的名称保持一致,x 为 Sqrt 算子
的输入
    sqrt_op.UpdateInputDesc("x", tensorDesc);
    // 调用 InferShapeAndType 函数,InferShapeAndType()接口为固定接口,用例执行时会自动
调用算子原型定义中的 shape 推导函数
    auto ret = sqrt_op.InferShapeAndType();
    // 验证调用过程是否成功
    EXPECT_EQ(ret, ge::GRAPH_SUCCESS);

    // 获取算子输出并比较参数 shape 和 type,算子输出的名字需要与原型定义 *.h 文件中的
名称保持一致,例如:算子的输出为 y
    auto output_desc = sqrt_op.GetOutputDesc("y");
    EXPECT_EQ(output_desc.GetDataType(), ge::DT_FLOAT16);
    std::vector<int64_t> expected_output_shape = {2, 3, 4};
    EXPECT_EQ(output_desc.GetShape().GetDims(), expected_output_shape);
}
```

若不同输入的参数 shape 不同，请自行定义多个 TensorDesc 对象进行设置，例如：

```
ge::op::Operator1 operator1_op;                    //Operator1 为算子的类型
ge::TensorDesc tensorDesc1;
ge::TensorDesc tensorDesc2;
ge::Shape shape1({2, 3, 4});
ge::Shape shape2({3, 4, 5});
```

```
tensorDesc1.SetDataType(ge::DT_FLOAT16);
tensorDesc1.SetShape(shape1);
tensorDesc2.SetDataType(ge::DT_FLOAT16);
tensorDesc2.SetShape(shape2);
// 更新算子输入
operator1_op.UpdateInputDesc("x1", tensorDesc1);
operator1_op.UpdateInputDesc("x2", tensorDesc2);
```

4）运行算子实现文件的 UT 用例

开发人员可以执行当前工程中所有算子的 UT 用例，也可以执行单个算子的 UT 用例。前者可以通过右击 testcases/ut/ops_test 文件夹，选择 Run Tbe Operator‘All’UT Impl with coverage，执行整个文件夹下算子实现代码的测试用例。后者通过右击“testcases/ut/ops_test/test_sqrt_impl.py”文件夹，选择 Run Tbe Operator‘算子名称’UT Impl with coverage，执行单个算子实现代码的测试用例。

测试用例第一次运行时会弹出运行配置界面，请根据界面提示配置，然后单击 Run。运行完成后，通过界面下方的 Run 日志打印窗口查看运行结果。在 Run 窗口中单击 index.html 的 URL（URL 中的 localhost 为 MindStudio 安装服务器的 IP，建议直接单击打开），查看 UT 用例的覆盖率结果，运行完成后，生成的中间文件与可执行文件目录结构如下：

```
├─── MyOperator                          //工程根目录
|    ├─── out
|    |    ├─── bin
|    |    |    ├─── data
|    |    |    ├─── kernel_meta
|    |    |    ├─── report
|    |    |    ├─── task_scheduler.dump
|    |    ├─── coverage_report
|    |    |    ├─── ut_impl
|    |    ├─── model
|    |    ├─── **.json
|    |    ├─── **.json
```

如果在配置运行信息选择的 Target 为 Simulator_TMModel，还可以查看执行流水线，如图 3-14 所示。

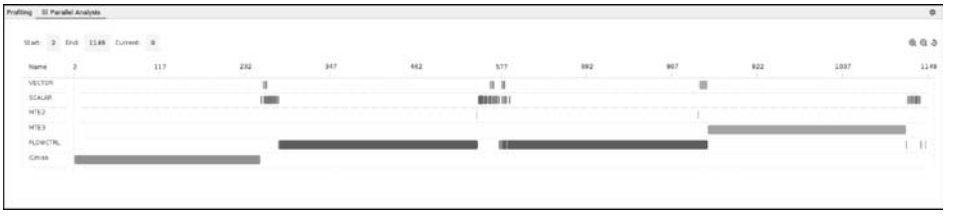

图 3-14　流水线

3.2.7　算子系统测试

MindStudio 提供了算子 ST(System Test,系统测试)框架,可以自动生成测试用例,在真实的硬件环境中,验证算子功能的正确性和计算结果准确性,包括:

(1) 根据算子实现和算子信息文件(*_impl.py)生成算子测试用例定义文件(*.json),作为算子 ST 用例的输入。

(2) 根据算子测试用例定义文件生成 ST 数据及测试用例执行代码,在硬件环境上执行算子测试用例。

ST 的流程为:生成算子测试用例定义文件 *.json,将自定义算子转换成单算子离线模型文件(*.om),然后使用 AscendCL 加载离线模型,并传入算子输入数据,执行算子,通过查看输出结果验证算子功能是否正确,如图 3-15 所示。

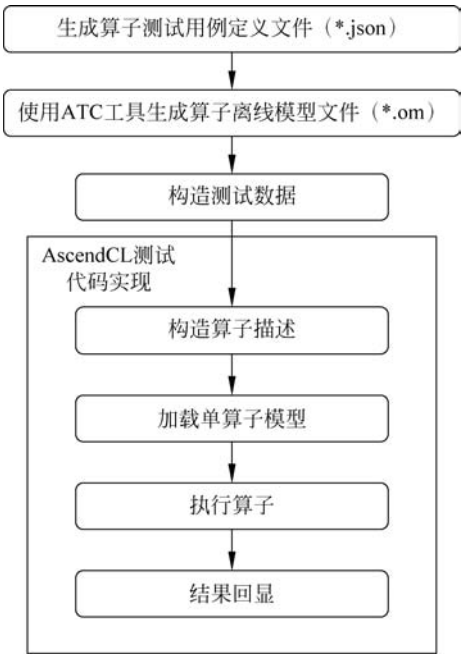

图 3-15　系统测试流程

其中,MindStudio 中的 ST 用例生成工具已进行了测试框架的封装,用户只需定义算子测试用例文件 *.json,即可自动生成测试数据以及 AscendCL 测试代码。

1. 配置算子测试用例定义文件(*.json)

在 MindStudio 工程界面右击算子工程根目录,选择 New Cases→ST Case 命令,在弹出的 Create ST Cases for an Operator 界面中选择需要创建 ST 用例的算子,如图 3-16 所示。

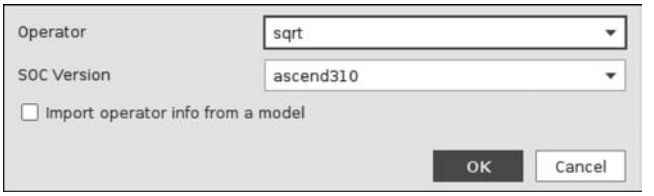

图 3-16　创建 ST 用例的算子

若不勾选 Import operator info from a model，单击 OK 按钮后，会生成 Shape 为空的算子测试用例定义文件，Design 视图如图 3-17 所示。

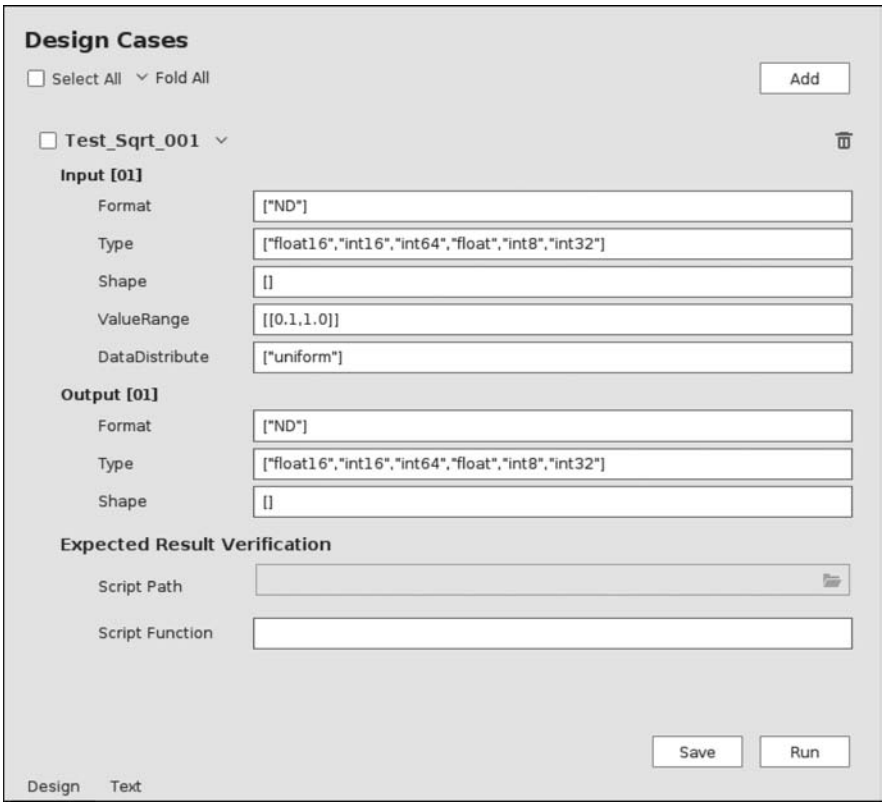

图 3-17　Design 视图

需要进行 Shape 信息的配置，用于生成测试数据及测试用例，也可以根据需要进行其他字段的配置，每个字段的详细说明可以参考网站 https://www.hiascend.com/document 中的《MindStudio 用户指南》文档。若用户勾选 Import operator info from a model，选择包含算子的 TensorFlow 模型文件(*.pb)后，界面会显示获取到的模型文件的首层 Shape 参数信息。

若要将算子与标杆数据对比，需要定义并配置算子期望数据生成函数，算子期望数据

生成函数是用 TensorFlow 或 Caffe 等框架实现的与自定义算子功能相同的函数，其可以在 CPU 上运行并生成标杆数据。标杆数据用来与自定义算子生成数据进行对比，根据对比结果确定自定义算子精度。算子期望数据生成函数用 Python 语言实现，在一个 Python 文件中可以实现多个算子期望数据生成函数。该函数的输入、输出、属性与自定义算子的输入、输出、属性的 Format、Type、Shape 保持一致。

在 Design 视图下，配置方法为：在 Expected Result Verification 下的 Script Path 中选择算子期望数据生成函数的 Python 文件所在路径；在 Script Function 中输入算子期望数据生成函数的函数名。

如果不进行标杆数据对比，那么会自动根据输入数据生成输出数据，并把实际结果返回。

在 Text 视图下，可以看到自定义算子 json 文件用于进行算子的描述，需要按照算子原型定义进行配置，包括算子的输入、输出及属性信息，配置示例如程序清单 3-20 所示。

程序清单 3-20　配置示例

```
[
  {
    "op": "Sqrt",
    "input_desc": [
      {
        "format": "NCHW",
        "shape": [8, 512, 7, 7],
        "type": "float16"
      },
      {
        "format": "NCHW",
        "shape": [512, 512, 3, 3],
        "type": "float16"
      }
    ],
    "output_desc": [
      {
        "format": "NCHW",
        "shape": [8, 512, 7, 7],
        "type": "float16"
      }
    ],
  }
]
```

修改测试用例信息后，单击 Save 按钮，修改会保存到算子测试用例定义文件。

算子测试用例定义文件存储在算子工程根目录下的 testcases/st/OpType/{SocVersion}文件夹下，命名为 OpType_case_timestamp.json。

json 文件为 OpDesc 的数组，OpDesc 数组参数说明如表 3-5 所示。

表 3-5　OpDesc 数组参数说明

属　性　名	类　　型	说　　明	是 否 必 填
Op	string	算子类型	是
input_desc	TensorDesc 数组	算子输入描述	是
output_desc	TensorDesc 数组	算子输出描述	是
Attr	Attr 数组	算子属性	否

其中，TensorDesc 数组的常用参数说明如表 3-6 所示。

表 3-6　TensorDesc 数组的常用参数说明

属性名	类　　型	说　　明	是否必填
format	string	Tensor 的排布格式，配置为算子原始框架支持的 format。当前支持 NHWC、NCHW、ND、NC1HWC0 等	是
type	string	Tensor 的数据格式	是
shape	int 数组	Tensor 的形状，例如[1,224,224,3]	是

Attr 数组的参数说明如表 3-7 所示。

表 3-7　Attr 数组的参数说明

属性名	类　　型	说　　明	是否必填
Name	string	属性名	是
Type	string	属性值的类型	是
Value	由类型的取值决定	属性值，根据类型的不同，属性值不同	是

2. 使用 ATC 工具生成单算子模型文件

使用 ATC 工具，加载单算子描述的 json 文件，生成单算子的离线模型，命令格式如下（具体使用可参考 6.3 节内容）：

```
atc --singleop=test_data/config/xxx.json --soc_version=${soc_version} --output=op_models
```

（1）singleop：算子描述的 json 文件，为相对于执行 atc 命令所在目录的相对路径。

（2）soc_version：昇腾 AI 处理器的型号，请根据实际情况替换。

（3）output：生成模型文件的存储路径，为相对于执行 atc 命令所在目录的相对路径。

模型转换成功后，在当前目录的 op_models 目录下生成单算子的模型文件 0_Sqrt_3_

2_8_16_3_2_8_16_3_2_8_16.om，命名规范为：序号＋opType ＋ 输入的描述(dateType_format_shape)＋输出的描述。

3. 运行 ST 用例

右击生成的 ST 用例定义文件(路径为“testcases/st/sqrt/{SOC Version}/xxxx.json”)，选择 Run The Operator 'xxx' ST Case，配置界面如图 3-18 所示。

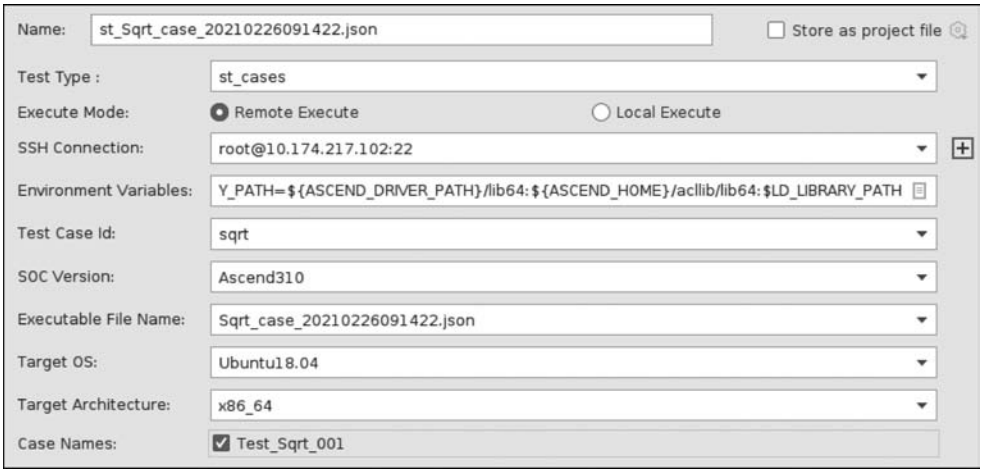

图 3-18　配置界面

其中需要添加如下配置：

(1) SSH Connection：当 Execute Mode 栏选择 Remote Execute 单选按钮时，在 SSH Connection 栏从下拉列表中选择 SSH 配置信息，若未添加配置信息，请单击＋按钮添加。

(2) Environment Variables：设置环境变量。

环境变量设置如下：

```
export install_path = /home/HwHiAiUser/Ascend/ascend - toolkit/latest
export PATH = ${install_path}/atc/ccec_compiler/bin: ${install_path}/atc/bin: $PATH
export ASCEND_OPP_PATH = ${install_path}/opp
```

其中，install_path 为 ATC 工具与 OPP 组件的安装路径。若远程设备为推理环境，则有：

```
LD_LIBRARY_PATH = ${ASCEND_DRIVER_PATH}/lib64: ${ASCEND_HOME}/acllib/lib64: $LD_LIBRARY_PATH;
```

若远程设备为训练环境，则有：

```
LD_LIBRARY_PATH = ${ASCEND_DRIVER_PATH}/lib64/driver: ${ASCEND_DRIVER_PATH}/lib64/common: ${ASCEND_HOME}/lib64: $LD_LIBRARY_PATH;
```

另外，在远程设备上也需要配置环境变量。针对 Ascend EP，需要在硬件设备的 Host 侧配置安装组件路径的环境变量。以 Host 侧运行用户在～/.bashrc 文件中配置 acllib 或 fwkacllib、driver 组件的安装路径。打开运行用户下的.bashrc 文件，在文件最后添加如下信息：

```
export ASCEND_DRIVER_PATH = /usr/local/Ascend/driver
export ASCEND_HOME = /usr/local/Ascend/ascend - toolkit/latest
export ASCEND_AICPU_PATH = ${ASCEND_HOME}/< target architecture >
```

若远程设备为推理环境，则有：

```
export LD_LIBRARY_PATH = ${ASCEND_DRIVER_PATH}/lib64: ${ASCEND_HOME}/acllib/lib64: $LD_
LIBRARY_PATH
```

若远程设备为训练环境，则有：

```
export
LD_LIBRARY_PATH = ${ASCEND_DRIVER_PATH}/lib64/driver: ${ASCEND_DRIVER_PATH}/lib64/
common: ${ASCEND_HOME}/fwkacllib/lib64: $LD_LIBRARY_PATH
```

若已存在如上所述环境变量，请确认为当前运行环境实际安装组件所在路径。

完成配置后，单击 Run 按钮，MindStudio 会根据算子测试用例定义文件在“算子工程根目录/testcases/st/out/OpType”下生成测试数据和测试代码，并编译出可执行文件，在指定的硬件设备上执行测试用例，并将执行结果与标杆数据对比，打印报告到输出窗口中，同时在“算子工程根目录/testcases/st/out/OpType”下生成 st_report.json 文件。

3.3 TBE TIK 算子开发

TIK(Tensor Iterator Kernel，张量迭代内核)是一种基于 Python 语言的动态编程框架，为一个 Python 模块，运行于 Host CPU 上。TIK 算子开发方式灵活，在算子开发效率和算子性能自动优化上有着一定的优势。

3.3.1 TIK 的适用场景

DSL 算子开发方式具有入门容易、易于实现、工具自动调度等优点，但也存在调测不方便、性能调优难度高等缺点，TIK 就是为对性能和调测要求较高的用户准备的，当然这也意味着开发难度的提升。

TIK 算子开发方式有两个优势：一是支持 Debug 模式，支持 TIK Debug 模式实现类似 PDB(Python Debugger，Python 调试器)的调试命令行界面并集成到 MindStudio 中，

能帮助用户快速定位功能问题，极大缩短开发调测时间；二是性能优化，通过手动调度可以更加精确地控制数据搬运和计算流程，从而实现更高的性能，将昇腾 AI 处理器的功能发挥到极致。目前昇腾官网提供在线实验 TIK 算子开发供用户练习，参考链接 https://www.hiascend.com/zh/college/onlineExperiment/codeLabTbeTik/tiks。

3.3.2　TIK 算子开发示例

1. 算子分析

TIK 算子用于实现从全局内存（Global Memory）中的 A、B 处分别读取 128 个 float16 类型的数值并相加，将结果写入全局内存地址 C 中，即两个张量的 Elementwise（元素级）相加，数学表达式为 C＝A＋B。

算子需要定义实现文件名称、实现函数名称以及算子的 OpType。算子 OpType 需要采用大驼峰的命名方式，即采用大写字母区分不同的语义，算子文件名称、算子函数名称则采用全小写；按照规则定义该算子 OpType 为 Add，算子实现文件名称与算子实现函数名称命名为 add。

通过查询 API 列表和分析算子要求，整个实现过程需要调用以下接口：

（1）定义数据：需要使用张量接口。

（2）数据搬运：将输入数据从全局内存搬入统一缓冲区（UB）。需要使用 data_move 接口。

（3）数据运算：将搬入的数据进行 vadd 计算。需要使用 vadd 接口。

（4）数据搬运：将得到的结果从统一缓冲区（UB）搬出到全局内存。需要使用 data_move 接口。

如 DSL 定义新建一个算子一样，OpType 为 Add，MindStudio 会在 tbe/impl 目录下生成 add.py。

首先导入必要的 Python 模块，代码如下：

```
from tbe import tik
import numpy as np
```

2. 定义目标机并构建 TIK DSL 容器

通过 TIK 类构造函数构造 TIK DSL 容器，代码如下：

```
tik_instance = tik.Tik()
```

3. 算子实现

算子实现由数据定义、数据搬运、数据计算三个部分组成，下面分别进行讲解。

1）数据定义

算子要实现 C＝A＋B，需要三个数据，并且它们需要在全局内存（GM）和统一缓冲区（UB）均通过 tik_instance.Tensor 函数进行定义，其中在全局内存中定义输入数据 data_A、data_B 和输出数据 data_C，在统一缓冲区中定义数据 data_A_ub、data_B_ub、data_C_ub。它们均为 128 个 float16 类型的数据。定义数据见程序清单 3-21。

程序清单 3-21　定义数据

```
data_A = tik_instance.Tensor("float16", (128,), name = "data_A", scope = tik.scope_gm)
data_B = tik_instance.Tensor("float16", (128,), name = "data_B", scope = tik.scope_gm)
data_C = tik_instance.Tensor("float16", (128,), name = "data_C", scope = tik.scope_gm)

data_A_ub = tik_instance.Tensor("float16", (128,), name = "data_A_ub", scope = tik.scope_
ubuf)
data_B_ub = tik_instance.Tensor("float16", (128,), name = "data_B_ub", scope = tik.scope_
ubuf)
data_C_ub = tik_instance.Tensor("float16", (128,), name = "data_C_ub", scope = tik.scope_
ubuf)
```

程序中调用 tik_instance.Tensor 函数生成算子需要的张量 data_A，其中元素的数据类型为 float16，地址在全局内存上，其他类似。

TIK 提供标量（Scalar）数据与张量（Tensor）数据两种类型，它们的属性 dtype 标识数据类型，如 int8、uint8、int16、uint16、int32、uint32、float16、float32、uint1（bool）等。TIK 为强类型语言，即不同类型的数据之间无法进行计算。

张量数据对应于存储缓冲区（含 GM 和 UB）中的数据，定义方式如程序清单 3-21 所示。TIK 会自动为每个申请的张量对象分配空间，并且避免各数据块之间的地址冲突。每个张量有四个属性（dtype、shape、scope、name），格式如下：

```
Tensor(dtype, shape, scope, name)
```

- dtype：指定张量对象的数据类型。
- shape：指定张量对象的形状。
- scope：指定张量对象所在缓冲区的空间（scope_gm 表示全局内存中的数据；scope_ubuf 表示统一缓冲区中的数据）。
- name：指定张量名字，不同张量名字需要保持唯一。

张量的生命周期自申请时被创建，若跳出所在代码块，则张量被释放，张量创建与释放之间的状态称为活跃状态，张量只有在活跃状态时才能被访问；同时由于存储空间的限制，任何时刻活跃状态的张量所占用的缓冲区的总大小不超过对应物理缓冲区的总大小，如图 3-19 所示。

标量数据对应于存储寄存器或者标量缓冲区中的数据，定义为

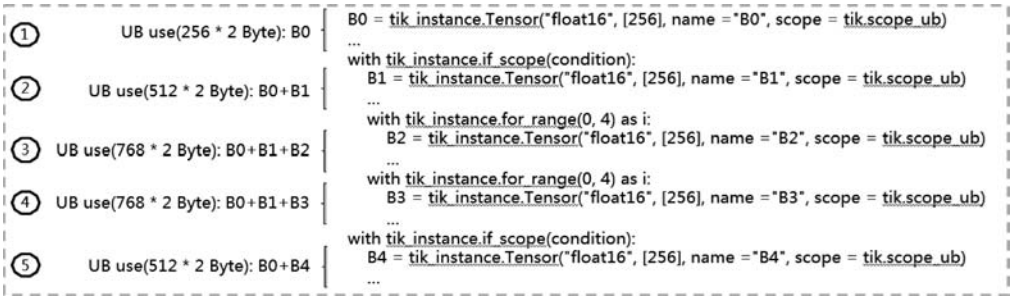

图 3-19　缓冲区使用

```
data_Sample = tik_instance.Scalar(dtype = "float32")
```

标量的生命周期自申请时被创建，若跳出所在代码块，则标量被释放，标量创建与释放之间的状态称为活跃状态，标量只有在活跃状态时才能被访问。如图 3-20 显示 S0、S1、S2 三个标量的活跃状态。

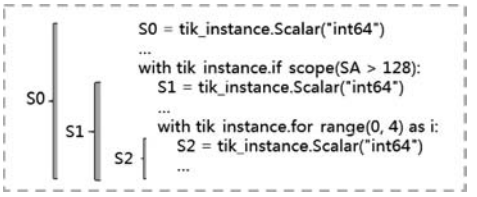

图 3-20　标量活跃状态

2）数据搬运与数据计算

数据搬运、数据计算等属于算子的计算主体，主要靠指令 API 实现。指令 API 与 AI Core 的指令集基本一一对应，分为数据搬运接口、矢量运算接口、矩阵运算接口和特定算法接口。示例算子主要用的是前两个接口。

（1）数据搬运接口：AI Core 数据运算在片上缓冲区中进行，因此需要将数据从全局内存中搬运到片上缓冲区中；同时计算完成后需要将计算结果从片上缓冲区搬运到全局内存中。数据搬运接口就是用于在不同的内存之间相互搬运数据，与 DSL 不同，TIK 需要用户自己根据数据大小来实现数据搬运和数据计算，同时对于不同类型数据有不同的指令。示例算子中用到的数据搬运接口指令如下：

```
data_move (dst, src, sid, nburst, burst, src_stride, dst_stride, *args, **argv)
```

其中，dst、src 是源地址与目标地址，用前面定义的 data_A、data_A_ub 等来表示；sid 是 SMMU ID，为硬件保留接口，输入 0 即可；burst 是指一次搬运连续的块数量，取值范围是[1，65535]，在 UB 里面一个块是 32B，比如搬运的数据量是 32KB，且都是地址连续搬运，那么 burst＝32×1024/32＝1024；nburst 就是搬运的次数，即 burst 的数量，取值范围是[1，4095]。如果是连续搬运，那么 nburst＝1，如果不连续，那么 nburst 就是搬运总量

除以 burst。

src_stride,dst_stride 是张量相邻连续数据片段间隔,即前 burst 尾与后 burst 头的间隔(前后两个块的间隔)。AI Core 中的向量计算单元要求所有计算的源数据以及目标数据存储在统一缓冲区中,并要求 32 字节对齐,它覆盖各种基本的计算类型和许多定制的计算类型,主要包括 FP16/FP32/int32/Int8 等数据类型的计算。固定型号芯片的每个缓冲区都有自己的大小和最小访问粒度,如表 3-8 所示。

表 3-8 缓冲区的大小和最小访问粒度

缓冲区(Buffer)名称	大小/KB	最小访问粒度/B
L1 Buffer	1024	32
L0A/L0B Buffer	64	512/128
L0C Buffer	256	512/1024
Unified Buffer(UB)	256	32/2
Scalar Buffer(SB)	16	2

块即在空间维度上运算单元一次访问数据的最小粒度,对于不同的计算单元,块的大小也不一样,向量运算从 UB 读取数据,一次最小 32B,也就是一个块为 32B,包括 16 个 float16/uint16/int16、8 个 float32/uint32/int32 或 32 个 int8/uint8 的数据。由于支持连续寻址和间隔寻址方式,因此需要指定寻址跨度(stride,即块的头与头之间距离),以块为单位;如果读取的数据是连续的块,那么 src_stride=0,如果读取的数据需要间隔一个块,那么 src_stride=1,dst_stride 也是如此。

* args, ** argv 是预留参数,暂不使用。

(2) 矢量运算接口:矢量运算使用的计算单元是 Vector 计算单元,使用的片上内存是统一缓冲区,遵循的执行模型是 SIMD(SingleInstruction Multiple Data,单指令多数据流指令,指单条指令可以完成多个数据操作)。指令操作分布在空间和时间两个维度,空间上即按照块为单位进行分组,时间上会以 repeat 参数为单位进行迭代,通过这两个维度可以最大限度地实现一行代码操作多个数据。

示例算子使用的是 vadd,如下:

```
vec_add(mask, dst, src0, src1, repeat_times, dst_rep_stride, src0_rep_stride, src1_rep_stride)
```

- mask 参数有的时候取一个块,只想其中的部分数据参与运算,该如何处理? 1 次迭代内部的数据,计算哪些数据,不计算哪些数据,由 mask 参数决定。针对 float16 数据,Vector 计算单元一次最多计算 128 个元素,如 mask=16,表示前 16 个元素参与计算。128 表示计算所有元素;
- dst,src 是源地址与目标地址,用前面定义的 data_A、data_A_ub 等表示;
- repeat_time,dst_rep_stride,src0_rep_stride,src1_rep_stride 是在当前版本的 TIK API 中,Vector 计算单元每次读取连续的 8 个块(256 字节)数据进行计算;为完成对输入数据的处理,Vector 计算单元必须通过多次迭代(repeat)才能完成

所有数据的读取与计算。参数 repeat_times 表示单次 API 调用中执行的迭代次数。考虑到每次 API 启动都有固定时延，将多次迭代放入单次 API 调用中，可大幅减少不必要的启动开销，从而提升整体执行效率。考虑到 AI Core 本身的硬件限制，repeat_times 取值上限为 255。参数 dst_rep_stride、src_rep_stride 分别表示目的操作数和源操作数在相邻迭代间相同块间的地址步长，为方便起见，这类参数以下记为 * _rep_stride。如图 3-21 所示，一个数字方框表示一个块，一次取 8 个块计算，* _rep_stride 为 8，代表每次迭代之间为 8 个块。

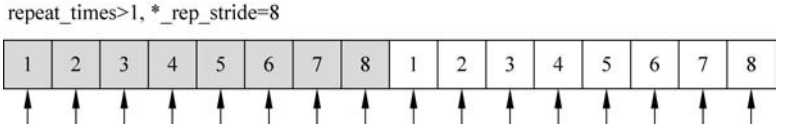

图 3-21　repeat_times 与 * _rep_stride 实例

3）算子实现及解读

示例算子实现见程序清单 3-22。

程序清单 3-22　示例算子

```
tik_instance.data_move(data_A_ub, data_A, 0, 1, 128 //16, 0, 0)
tik_instance.data_move(data_B_ub, data_B, 0, 1, 128 //16, 0, 0)
tik_instance.vec_add(128, data_C_ub[0], data_A_ub[0], data_B_ub[0], 1, 8, 8, 8)
tik_instance.data_move(data_C, data_C_ub, 0, 1, 128 //16, 0, 0)
```

首先是数据从全局内存搬运到统一缓冲区中，需要搬运的数据为 128 个 float16 类型的数据，占 128×2 字节，等于统一缓冲区的大小(256KB)，因此搬一次就可以把输入数据全部搬到统一缓冲区，搬运的次数 nburst 为 1；由于 burst 单位为 32B，每次搬运的数据大小 burst 为 128×2/32B，间隔跳跃式搬运时需要设置参数 stride，示例中为连续搬运，因此两个参数都设置为 0；

TIK 的向量指令每个周期能处理 256B 的数据，并提供 mask 参数调整计算的数据，同时在时间上支持迭代操作，完成一连串的数据计算。TIK 指令操作分布在空间和时间两个维度，其中空间上最多处理 256B 数据(包括 128 个 float16/uint16/int16、64 个 float32/uint32/int32 或 256 个 int8/uint8 的数据)，时间上支持迭代操作。1 次迭代内部的数据计算由 mask 参数决定。针对 float16 数据，Vector 计算单元一次计算 128 个元素，如 mask=128，表示前 128 个元素参与计算。

在 vec_add 中，所有元素参与运算，mask 设置为 128；然后是三个操作数的地址，本次为连续寻址运算 128 个元素，对于 128 个 float16 的数据，通过 1 次迭代可以完成计算，因此 repeat_times 为 1；一次迭代内，因为块大小是 16 个 FP16，那么 128 个 FP16 元素需要 8 个块，所以 * _rep_stride =8，表示一次迭代连续处理 8×32B 数据。

最后，通过 date_move API 把计算结果从统一缓冲区搬运到全局内存中。

4. 算子编译

算子实现后，需要通过 BuildCCE 函数将上述定义的 TIK DSL 容器编译成昇腾 AI 处理器上可执行的二进制代码，即算子的.o 文件和算子描述.json 文件。BuildCCE 属于前述的控制函数，例如：

```
tik_instance.BuildCCE(kernel_name = "add", inputs = [data_A,data_B], outputs = [data_C])
```

其中，kernel_name 指明编译产生的二进制代码中的 AI Core 核函数名称，inputs、outputs 分别指明程序的输入和输出张量。

最后结束程序，代码如下：

```
return tik_instance
```

至此，示例程序已经完成，代码见程序清单 3-23。

程序清单 3-23　示例程序

```
from tbe import tik
import numpy as np

def add():
    tik_instance = tik.Tik()

    data_A = tik_instance.Tensor("float16", (128,), name = "data_A", scope = tik.scope_gm)
    data_B = tik_instance.Tensor("float16", (128,), name = "data_B", scope = tik.scope_gm)
    data_C = tik_instance.Tensor("float16", (128,), name = "data_C", scope = tik.scope_gm)
    data_A_ub = tik_instance.Tensor("float16", (128,), name = "data_A_ub", scope = tik.
scope_ubuf)
    data_B_ub = tik_instance.Tensor("float16", (128,), name = "data_B_ub", scope = tik.
scope_ubuf)
    data_C_ub = tik_instance.Tensor("float16", (128,), name = "data_C_ub", scope = tik.
scope_ubuf)

    tik_instance.data_move(data_A_ub, data_A, 0, 1, 128 //16, 0, 0)
    tik_instance.data_move(data_B_ub, data_B, 0, 1, 128 //16, 0, 0)
    tik_instance.vec_add(128, data_C_ub[0], data_A_ub[0], data_B_ub[0], 1, 8, 8, 8)
    tik_instance.data_move(data_C, data_C_ub, 0, 1, 128 //16, 0, 0)
    tik_instance.BuildCCE(kernel_name = "simple_add", inputs = [data_A,data_B], outputs =
[data_C])

    return tik_instance
```

5. TIK 调试

到目前为止，TIK 算子基本编写完成。与张量加速引擎(TBE)相比，TIK 给予用户更多

的自主权，也比 TBE 更加复杂。而实际上 TIK 优势很多，比如调试，TIK 功能调试是基于功能仿真的一个调试工具，与开源 GDB 一样，可进行断点设置、单步调试、变量打印等操作。TIK 提供 start_debug 和 debug_print 接口，方便用户进行调用，见程序清单 3-24。

程序清单 3-24　调试代码

```
if __name__ == "__main__":

    #调用 TIK 算子函数
    tik_instance = add()
    #初始化输入数据。调用 numpy 在 data 中初始化，生成一个具有 128 个 float16 类型的数字 1 的一维矩阵，并通过描述一个字典类型的格式，将 data_A、data_B 的数据与 data 进行绑定。初始化的数据需要和对应的 tensor 定义的 shape 和 dtype 保持一致，否则会报错。feed_dict 字典里面的 key 需要和对应的输入 tensor 的 name 保持一致
    data = np.ones((128,), dtype = np.float16)
    feed_dict = {"data_A": data, "data_B": data}
    #启动 TIK 调试，传入字典类型的数据，并在调试结束后返回输出结果 data_C。interactive = True，表示调试器会进入交互模式
    data_C, = tik_instance.tikdb.start_debug(feed_dict = feed_dict, interactive = True)
    #打印输出数据
    print("data_C:\n{}".format(data_C))
```

在需要设置断点的行的左边单击，即设置断点（再次单击即取消断点）；使用 Shift＋F9 或者在 RUN 菜单中选择 Debug‘add’命令即可进入 Debug 模式，如图 3-22 所示。

此时程序会停在断点处，并在下方 Debugger 选项卡中显示当前变量值的输出；按 F8 键则进入单步执行模式，同时每条语句执行完后在语句后面会出现此时对应的值，方便定位，如图 3-23 所示。

执行到程序的最后会跳到 console（控制台）中，此时可以输入 L 查看正在执行处的语句，输入 n 则执行到下一条 TIK 语句，输入 p 变量名则可以打印当前存在的变量值，如图 3-24 所示。

输入 p（print）对表达式求值并打印结果。

表达式（expression）可以是任意 Python 表达式。表达式可以使用的变量有 TIK DSL 当前作用域的 Tensor（张量）和 Scalar（标量）。其中 Tensor 会被替换为与 Tensor 等价的 numpy.ndarray，这个 numpy 对象的形状、类型和数据都与 Tensor 一致，Scalar 会被求值并替换为 Python 的 float 或 int 类型的数值。表达式也可以传入纯字符串或者字符串与表达式的复合情况，如图 3-25 所示。

在命令行输入 C 结束整个程序。

通过断点设置和单步调试、打印等功能，TIK 让用户定位问题更加简单，大大提升了开发效率。

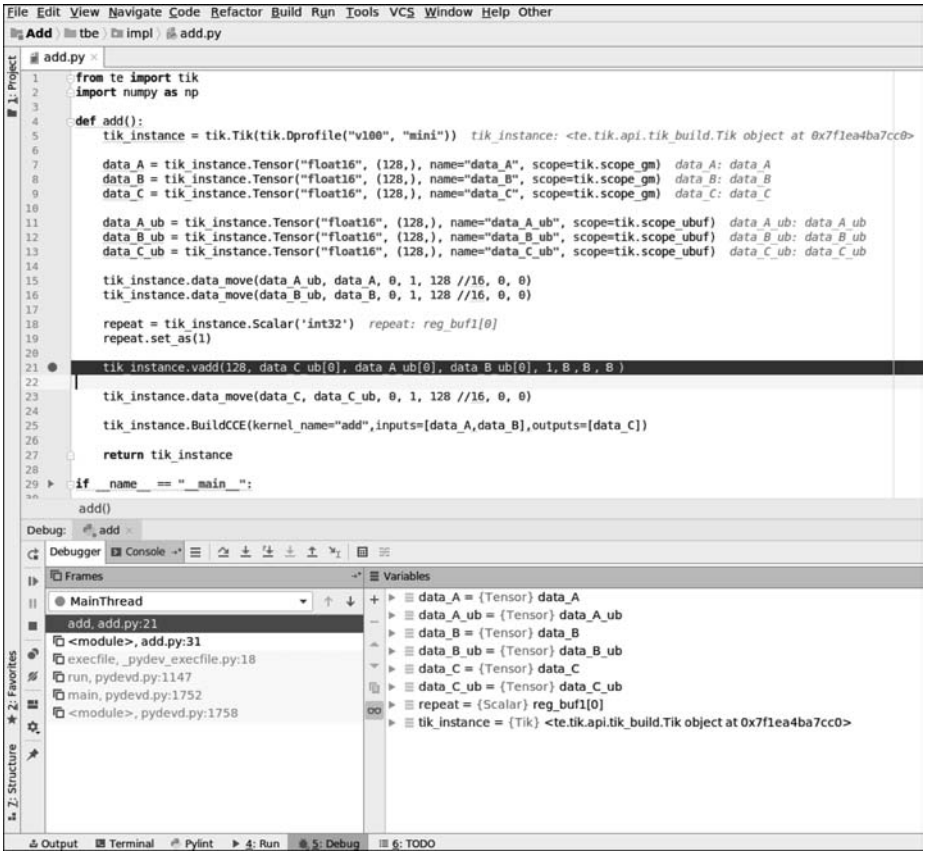

图 3-22　Debug 模式

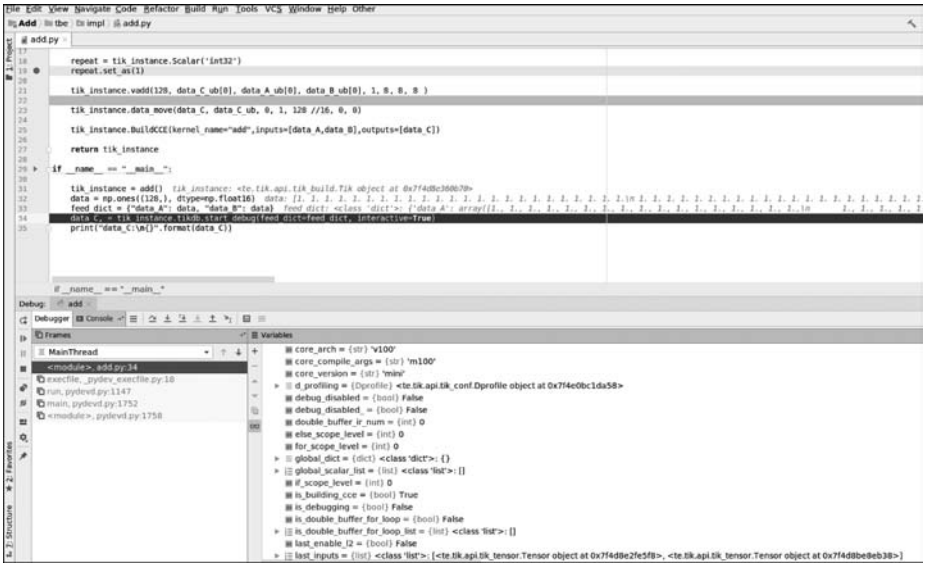

图 3-23　断点界面

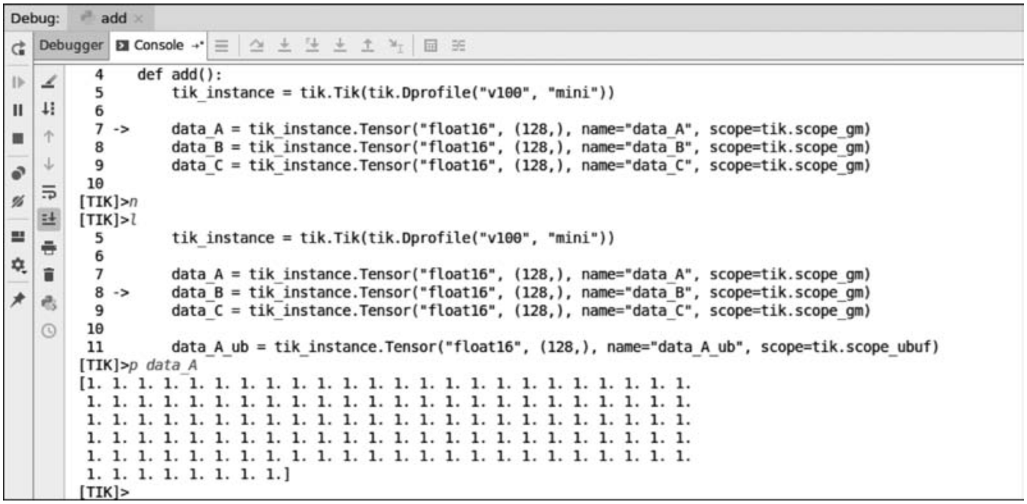

图 3-24　打印结果

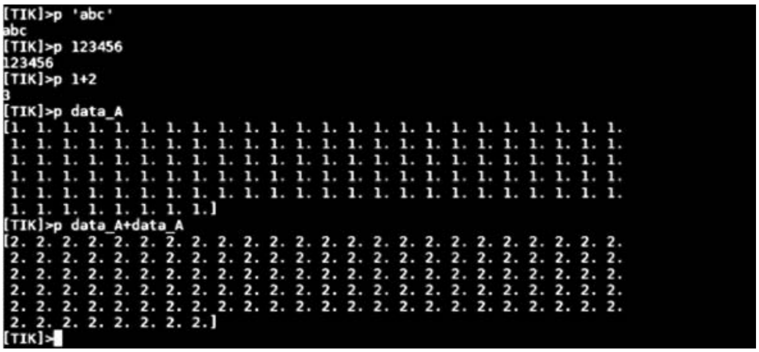

图 3-25　复合情况

3.3.3　算子的性能优化

1. 计算分片

由于统一缓冲区(简称 UB)空间有限,在数据量很大的情况下,无法完整放入输入数据和输出结果,需要对输入数据分片搬入、计算和再搬出。在进行计算分片时,主要考虑如下因素:

(1) 在切分输入数据时要合理地利用 UB 空间,减少搬运次数,从而提高性能。

(2) 由于 UB 上的物理限制,要求数据存储必须保持 32B 数据对齐。

(3) 计算时相同指令运算数据尽可能连续存储,充分利用迭代操作,提高向量利用率。

(4) 计算尾块的处理。对于向量指令计算,单条向量指令支持最大的迭代次数,为

255，以 float16 类型数据举例，每次计算 128 个数。因此，向量计算需要分三步判断：

① 如果数据量大于 255×128 个数，可以设置 repeat＝255，mask＝128，处理 N 次，把这部分数据处理完。

② 剩下的数据量在小于 255×128 个数，大于 128 个数之间，此时设置 mask＝128，求出迭代次数，通过一条指令处理完。

③ 剩下的数据量小于 128 个数，通过设置 mask 参数使用一条指令处理完即可。

下面以张量加法为例，给出较大数据场景下的计算分片方案，见程序清单 3-25。

程序清单 3-25　计算分片方案

```
# 给定数据类型,data_each_block 表示一个块能够存放的数据数目
# 向量指令每次迭代最多计算 8 个块,vector_mask_max 为 mask 的最大值
vector_mask_max = 8 * data_each_block

# move_num 表示搬入的数据数目
# 计算 repeat_times 取得最大值 255 时,需要循环调用 vec_add 多少次
vadd_loop = move_num // (vector_mask_max * 255)
add_offset = 0

if vadd_loop > 0:
with tik_instance.for_range(0, vadd_loop) as add_index:
        add_offset = add_index * vector_mask_max * 255
        tik_instance.vec_add(vector_mask_max, input_x_ub[add_offset],
                             input_x_ub[add_offset], input_y_ub[add_offset], 255, 8, 8, 8)

        # 对剩余数据,当向量计算单元满载运行时,计算单次调用 vec_add 需要多少次迭代
        repeat_time = (move_num % (vector_mask_max * 255) // vector_mask_max)
        if repeat_time > 0:
           add_offset = vadd_loop * vector_mask_max * 255
           tik_instance.vec_add(vector_mask_max, input_x_ub[add_offset],
                                input_x_ub[add_offset],
                                input_y_ub[add_offset], repeat_time, 8, 8, 8)

       # 数据尾巴,此时最后一次调用 vec_add,计算需要多少向量计算单元参与计算
       last_num = move_num % vector_mask_max
       if last_num > 0:
           add_offset += repeat_time * vector_mask_max
           tik_instance.vec_add(last_num, input_x_ub[add_offset],
                                input_x_ub[add_offset], input_y_ub[add_offset], 1, 8, 8, 8)
```

2. 双缓冲

如前面介绍，并行计算可以划分为时间并行和空间并行计算。时间并行计算即指令流水化，空间并行计算即使用多个处理器或者多个计算单元执行并发计算。

执行于 AI Core 上的指令队列主要包括如下几类，即向量指令队列(V)、矩阵指令队列(M)和存储移动指令队列(MTE2、MTE3)。不同指令队列间的相互独立性和可并行性是双缓冲优化机制的基石。

考虑一个完整的数据搬运和计算过程，如图 3-26 所示。MTE2 将数据从全局内存搬运到统一缓冲区，向量计算单元完成计算后将结果写回统一缓冲区，最后由 MTE3 将计算结果搬回全局内存。

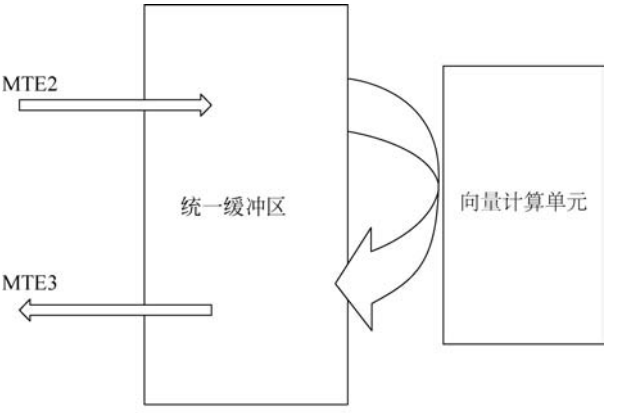

图 3-26　数据搬运和计算过程

在此过程中，数据搬运与向量计算串行执行，向量计算单元无可避免存在资源闲置问题。举例说明，若 MTE2、向量、MTE3 三阶段分别耗时 t，则向量计算单元的时间利用率仅为 1/3，等待时间过长，向量利用率严重不足。

为减少向量计算单元等待时间，双缓冲机制将统一缓冲区一分为二，即 UB_A、UB_B。如图 3-27 所示，当向量计算单元对 UB_A 中数据进行读取和计算时，MTE2 可将下一份数据搬入 UB_B 中；而当向量计算单元切换到计算 UB_B 时，MTE3 将 UB_A 的计算结果搬出，而 MTE2 则继续将下一份数据搬入 UB_A 中。由此，数据的进出搬运和向量计算实现并行执行，向量计算单元闲置问题得以有效缓解。

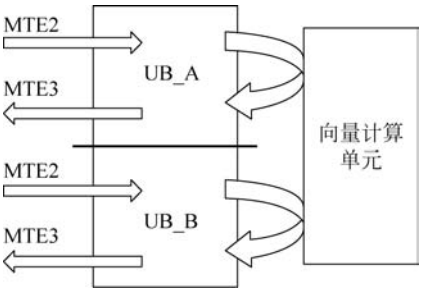

图 3-27　数据搬运和计算优化过程

总体来说，双缓冲机制是基于 MTE 指令队列与向量指令队列的独立性和可并行性，通过将数据搬运与向量计算单元并行执行以隐藏数据搬运时间并降低向量指令的等待时间，最终提高向量计算单元的利用效率。用户可以通过在 for_range 函数中设置参数 thread_num 来实现数据并行，简单代码示例如下：

```
with tik_instance.for_range(0, 10, thread_num = 2) as i:
```

考虑到算子通常具有计算简单、计算快速的特性，向量计算时间与数据搬运时间往往相差较小。多数情况下，采用双缓冲机制能有效提升向量的时间利用率，缩减算子执行时间。然而双缓冲机制缓解向量闲置问题并不代表它总能带来整体的性能提升。当数据搬运时间较短，而向量计算时间显著较长时，由于数据搬运在整个计算过程中的时间占比较低，双缓冲机制带来的性能收益会偏小。又如，当原始数据较小且向量可一次性完成所有计算时，强行使用双缓冲机制会降低向量计算资源的利用率，最终效果可能适得其反。

因此，双缓冲机制的性能收益需综合考虑向量算力、数据量大小、搬运与计算时间占比等多种因素。

3. 多核

多核并行计算属于空间并行，TIK 绑定多核的方式比较简单，TIK 所有的循环都是在代码中通过 for_range 函数创建，只需要设置最外层循环的参数 block_num 即可。代码如下：

```
with tik_instance.for_range( 0, 10, block_num = 10) as i:
```

其中，for_range 循环的表达式会被作用在 10 个执行实例上，最终 10 个执行实例会被分配到 10 个 AI Core 上并行运行，每个 AI Core 拿到一个执行实例和一个不同的块 ID。如果当前可用的 AI Core 小于 10，则执行实例会在当前可用的 AI Core 上分批调度执行；如果当前可用的 AI Core 大于等于 10，则会根据实际情况调度执行，实际运行的核数可能小于等于 10。

需要注意的是，block_num 默认取值为 1，即不分核；而采用分核并行时，其取值上限为 65535，用户需要保证 block_num 的实际值不超过阈值。采用分核并行时，L2/HBM/DDR(统称全局内存)对每个 AI Core 均可见，因而位于全局内存中的张量必须定义在 for_range 循环外；其他存放在标量缓冲区和统一缓冲区中的张量，只对其所在的 AI Core 可见，其张量定义必须放到多核循环内部。

用户可以通过如下接口函数获取 AI Core 的个数：

```
tbe.platform.get_soc_spec("CORE_NUM")
```

为保证负载均衡，block_num 一般尽量设置为 AI Core 的倍数。以昇腾 910 处理器为例，该处理器内含 32 个 AI Core。假如一个张量的形状为(16,2,32,32,32)，如果以张量的第一维度(最外层)进行分核，则只能绑定 16 个核。此时，可通过将张量的第一维度和第二维度合并，使得最外层的长度变成 32，以此将任务均摊到 32 个 AI Core 上。需要注意的是，顾及后端内存自动分配机制的限制，用户实施分核并行时必须从最外层开始做维度合并。

此外，分核并行特别需要注意非 32B 大小的数据对齐写入全局内存的情况，此时存在多余的数据尾巴，导致不同 AI Core 往全局内存写数据存在数据覆盖。

3.4　AI CPU 算子开发

AI CPU 算子是运行在昇腾 AI 处理器中 AI CPU 计算单元上的表达一个完整计算逻辑的运算。AI CPU 算子包括控制算子、标量和向量等通用计算逻辑。对于如下情况，用户需要自定义 AI CPU 算子。

(1) 在 NN 模型训练或者推理过程中，将第三方开源框架转换为适配昇腾 AI 处理器的模型时遇到了昇腾 AI 处理器不支持的算子。此时，为了快速打通模型执行流程，用户可以通过自定义 AI CPU 算子进行功能调测，提升调测效率。功能调通之后，后续性能调测过程中再将 AI CPU 自定义算子转换成 TBE 算子实现。

(2) 某些场景下，无法通过 AI Core 实现自定义算子(比如部分算子需要 int64 类型，但 AI Core 指令不支持)，且该算子不是网络的性能瓶颈，此时可以通过开发 AI CPU 自定义算子实现昇腾 AI 处理器对此算子的支持。

这里结合实例介绍 AI CPU 算子开发的流程。关于 AI CPU 引擎的实现不属于本章讨论的范围，具体可参考前面介绍的 AI CPU 引擎。

1. 算子分析

使用 AI CPU 方式开发算子前，需要确定算子功能、输入和输出参数及数据类型、算子开发方式、算子类型以及算子实现函数名称等。

明确算子的功能以及数学表达式。以 Add 算子为例，Add 算子的数学表达式为：

$$z = x + y$$

计算过程是：将两个输入参数 x，y 相加，得到最终结果 z 并将其返回。

明确输入和输出参数及数据类型。例如 Add 算子有两个输入：x 与 y，输出为 z。本样例中算子的输入支持的数据类型为 int64，算子输出的数据类型与输入数据类型相同。算子输入支持所有形状(shape)，要求输出形状与输入形状相同。算子输入支持的格式(format)为：NCHW、NC1HWC0、NHWC、ND。

明确算子实现文件名称以及算子的类型(OpType)。

2. 新建算子

右击工程名 tbe_operator_sample，选择 New→Operator 命令，弹出如图 3-28 所示界面，在 Operator Type 栏输入算子类型名称：Add，在 Plugin Framework 栏选择 TensorFlow，即算子所在模型文件的 AI 框架类型，在 Compute Unit 栏选择 AI CPU 算子，如图 3-28 所示。

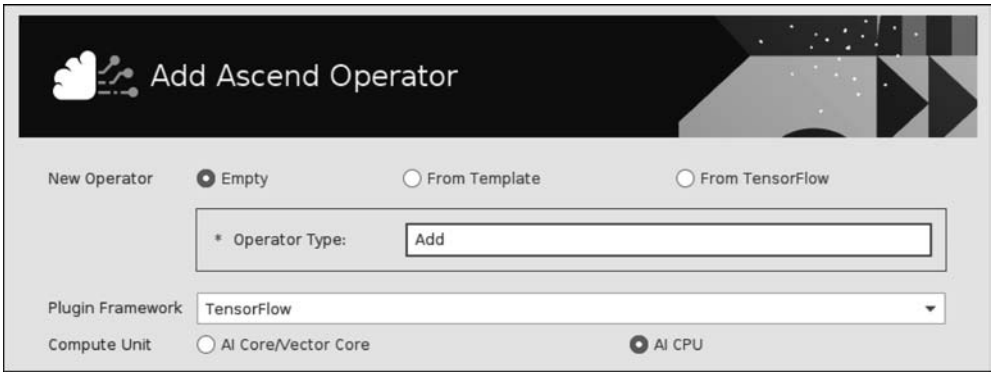

图 3-28　工程创建界面

3. 算子实现

首先是头文件的实现，需要在算子工程的 cpukernel/impl/add_kernels.h 文件中进行算子类的声明，如程序清单 3-26 所示。

程序清单 3-26　头文件

```
#ifndef _ADD_KERNELS_H_
#define _ADD_KERNELS_H_
#include "cpu_kernel.h"

namespace aicpu {
class AddCpuKernel : public CpuKernel {
public:
    ~AddCpuKernel() = default;
    virtual uint32_t Compute(CpuKernelContext &ctx) override;
};
} // namespace aicpu
#endif
```

具体说明如下：

（1）头文件 cpu_kernel.h 中包含了 AI CPU 算子基类 CpuKernel 的定义，以及 Kernels 的注册宏的定义；

（2）声明算子类，此类为 CpuKernel 类的派生类，并需要声明重载函数 Compute，Compute 函数需要在算子实现文件中实现。

（3）在算子工程的 cpukernel/impl/add_kernels.cc 文件中进行算子计算逻辑的实现，AI CPU 算子实现的关键是 Compute 函数的实现。

（4）定义命名空间 aicpu，并在命名空间 aicpu 中实现自定义算子的 Compute 函数，定义算子的计算逻辑。命名空间的名称 aicpu 为固定值，基类及相关定义都在 aicpu 命名

空间中。

(5) Compute 函数声明。例如：

```
uint32_t AddCpuKernel::Compute(CpuKernelContext &ctx)
```

AddCpuKernel 为头文件中定义的自定义算子类，形参 CpuKernelContext 为 CPU Kernel 的上下文，包括算子的输入/输出 Tensor 以及属性等相关信息。

Compute 函数体中，根据算子开发需求，编写相关代码实现获取输入张量相关信息，并根据输入信息组织计算逻辑，得出输出结果，并将输出结果设置到输出张量中。

Add 算子的参考实现如程序清单 3-27 所示。

程序清单 3-27　Add 算子的参考实现

```
#include "add.h"
#include "Eigen/Core"
#include "unsupported/Eigen/CXX11/Tensor"
#include "cpu_kernel_utils.h"
#include "utils/eigen_tensor.h"
#include "utils/kernel_util.h"
namespace {
constexpr uint32_t kOutputNum = 1;
constexpr uint32_t kInputNum = 2;
const char *kAdd = "Add";

#define ADD_COMPUTE_CASE(DTYPE, TYPE, CTX)                          \
  case (DTYPE): {                                                   \
    uint32_t result = AddCompute<TYPE>(CTX);                        \
    if (result != KERNEL_STATUS_OK) {                               \
      KERNEL_LOG_ERROR("Add kernel compute failed [%d].", result);  \
      return result;                                                \
    }                                                               \
    break;                                                          \
  }
}

namespace aicpu {
uint32_t AddCpuKernel::Compute(CpuKernelContext &ctx) {
  // check params
  KERNEL_HANDLE_ERROR(NormalCheck(ctx, kInputNum, kOutputNum),
                      "Check Add params failed.");

  auto data_type = ctx.Input(0)->GetDataType();
  switch (data_type) {
    ADD_COMPUTE_CASE(DT_INT8, int8_t, ctx)
    ADD_COMPUTE_CASE(DT_INT16, int16_t, ctx)
```

```
      ADD_COMPUTE_CASE(DT_INT32, int32_t, ctx)
      ADD_COMPUTE_CASE(DT_INT64, int64_t, ctx)
      ADD_COMPUTE_CASE(DT_UINT8, uint8_t, ctx)
      ADD_COMPUTE_CASE(DT_FLOAT16, Eigen::half, ctx)
      ADD_COMPUTE_CASE(DT_FLOAT, float, ctx)
      ADD_COMPUTE_CASE(DT_DOUBLE, double, ctx)
      ADD_COMPUTE_CASE(DT_BOOL, bool, ctx)
      ADD_COMPUTE_CASE(DT_COMPLEX64, std::complex<float>, ctx)
      ADD_COMPUTE_CASE(DT_COMPLEX128, std::complex<double>, ctx)
      default:
        KERNEL_LOG_WARN("Add kernel data type [%u] not support.", data_type);
        return KERNEL_STATUS_PARAM_INVALID;
    }
    return KERNEL_STATUS_OK;
  }

  template<typename T>
  uint32_t AddCpuKernel::AddCompute(CpuKernelContext &ctx) {
    BCalcInfo calc_info;
    calc_info.input_0 = ctx.Input(0);
    calc_info.input_1 = ctx.Input(1);
    calc_info.output = ctx.Output(0);
    DataType input0_type = calc_info.input_0->GetDataType();
    DataType input1_type = calc_info.input_1->GetDataType();
    DataType output_type = calc_info.output->GetDataType();

    Bcast bcast;
    KERNEL_HANDLE_ERROR(bcast.GenerateBcastInfo(calc_info), "Generate broadcast info
  failed.")
    (void)bcast.BCastIndexes(calc_info.x_indexes, calc_info.y_indexes);
    (void)bcast.GetBcastVec(calc_info);

    return AddCalculate<T>(ctx, calc_info);
  }

  template<typename T>
  uint32_t AddCpuKernel::AddCalculate(CpuKernelContext &ctx, BCalcInfo &calc_info) {
    auto input_x1 = reinterpret_cast<T *>(calc_info.input_0->GetData());
    auto input_x2 = r
  einterpret_cast<T *>(calc_info.input_1->GetData());
    auto output_y = reinterpret_cast<bool *>(calc_info.output->GetData());

    size_t data_num = calc_info.x_indexes.size();
    auto shard_add = [&](size_t start, size_t end) {
      for (size_t i = start; i < end; i++) {
```

```
        auto x_index = input_x1 + calc_info.x_indexes[i];
        auto y_index = input_x2 + calc_info.y_indexes[i];
        output_y[i] = (*x_index + *y_index);
      }
    };
    KERNEL_HANDLE_ERROR(CpuKernelUtils::ParallelFor(ctx, data_num, 1, shard_add),
                        "Add calculate failed.")
    return KERNEL_STATUS_OK;
}

REGISTER_CPU_KERNEL(kAdd, AddCpuKernel);
}  // namespace aicpu
```

REGISTER_CPU_KERNEL(kAdd,AddCpuKernel)用于注册算子的 Kernel 实现，第一个参数 kAdd 为定义的指向算子 OpType 的字符串指针，第二个参数 AddCpuKernel 为自定义算子类的名称。

4. 算子原型定义

算子原型定义(IR)用于描述算子，包括算子输入信息、输出信息、属性信息等，用于把算子注册到算子原型库中。算子原型定义需要在算子的工程目录的“op_proto/算子名称.h”和“op_proto/算子名称.cc”文件中实现。与 TBE 算子的原型定义是类似的，这里只以 Add 算子为例进行简单介绍。

Add.h 实现如程序清单 3-28 所示。

程序清单 3-28　Add.h 实现

```
#ifndef GE_OP_ADD_H
#define GE_OP_ADD_H
#include "graph/operator_reg.h"
namespace ge {

REG_OP(Add)
    .INPUT(x, TensorType({DT_FLOAT16,DT_FLOAT,DT_DOUBLE,
DT_INT8,DT_INT16,DT_INT32,DT_INT64}))
    .OUTPUT(y, TensorType({DT_FLOAT16,DT_FLOAT,DT_DOUBLE,
DT_INT8,DT_INT16,DT_INT32,DT_INT64}))
    .OP_END_FACTORY_REG(Add)
}
#endif //GE_OP_ADD_H
```

add.cc 实现如程序清单 3-29 所示。

程序清单 3-29　add.cc 实现

```
#include "./add.h"
#include <string>
#include <vector>

namespace ge {
bool InferShapeAndTypeAdd (Operator &op, const string &inputName1, const string
&inputName2, const string &outputName)
{
    TensorDesc vOutputDesc = op.GetOutputDescByName(outputName.c_str());

    DataType inputDtype = op.GetInputDescByName(inputName1.c_str()).GetDataType();
    Format inputFormat = op.GetInputDescByName(inputName1.c_str()).GetFormat();
    // 针对 shape 参数维度大小进行交换
    ge::Shape shapeX = op.GetInputDescByName(inputName1.c_str()).GetShape();
    ge::Shape shapeY = op.GetInputDescByName(inputName2.c_str()).GetShape();
    std::vector<int64_t> dimsX = shapeX.GetDims();
    std::vector<int64_t> dimsY = shapeY.GetDims();
    if (dimsX.size() < dimsY.size()) {
        std::vector<int64_t> dimsTmp = dimsX;
        dimsX = dimsY;
        dimsY = dimsTmp;
    }

    // 对小的 shape 进行 1 补齐
    if (dimsX.size() != dimsY.size()) {
        int dec = dimsX.size() - dimsY.size();
        for (int i = 0; i < dec; i++) {
            dimsY.insert(dimsY.begin(), (int64_t)1);
        }
    }

    // 设置输出的 shape 维度
    std::vector<int64_t> dimVec;
    for (size_t i = 0; i < dimsX.size(); i++) {
        if ((dimsX[i] != dimsY[i]) && (dimsX[i] != 1) && (dimsY[i] != 1)) {
            return false;
        }

        int64_t dims = dimsX[i] > dimsY[i] ? dimsX[i] : dimsY[i];
        dimVec.push_back(dims);
    }
    ge::Shape outputShape = ge::Shape(dimVec);

    vOutputDesc.SetShape(outputShape);
```

```
    vOutputDesc.SetDataType(inputDtype);
    vOutputDesc.SetFormat(inputFormat);
    op.UpdateOutputDesc(outputName.c_str(), vOutputDesc);

    return true;
}

// ----------------- Add --------------------
IMPLEMT_VERIFIER(Add, AddVerify)
{
   if (op.GetInputDescByName("x1").GetDataType()!= op.GetInputDescByName("x2").
GetDataType())
{
        return GRAPH_FAILED;
   }
   return GRAPH_SUCCESS;
}

// Obtains the processing function of the output tensor description.
IMPLEMT_COMMON_INFERFUNC(AddInferShape)
{
    if (InferShapeAndTypeAdd(op, "x1", "x2", "y")) {
        return GRAPH_SUCCESS;
    }
    return GRAPH_FAILED;
}

// Registered inferfunction
COMMON_INFER_FUNC_REG(Add, AddInferShape);

// Registered verify function
VERIFY_FUNC_REG(Add, AddVerify);
// ----------------- Add --------------------
} // namespace ge
```

5. 算子信息定义

算子信息库文件路径(在自定义算子工程目录下)为：

cpukernel/op_info_cfg/aicpu_kernel/xx.ini

Add 算子的信息定义如程序清单 3-30 所示。

程序清单 3-30　Add 算子的信息定义

```
[Add]
opInfo.engine = DNN_VM_AICPU
opInfo.flagPartial = False
opInfo.computeCost = 100
opInfo.flagAsync = False
opInfo.opKernelLib = CUSTAICPUKernel
opInfo.kernelSo = libcust_aicpu_kernels.so
opInfo.functionName = RunCpuKernel
opInfo.workspaceSize = 1024
```

其中

(1) opInfo. engine 为配置算子调用的引擎。AI CPU 自定义算子的引擎固定为 DNN_VM_AICPU。

(2) opInfo. opKernelLib 是配置算子调用的 kernelLib。AI CPU 自定义算子调用的 kernelLib 固定为 CUSTAICPUKernel。

(3) opInfo. kernelSo 是配置 AI CPU 算子实现文件编译生成的动态库文件的名称，建议使用 libcust_aicpu_kernels. so。

(4) opInfo. functionName 是配置自定义算子调用的 kernel 函数接口名称。自定义算子的 kernel 函数接口固定为 RunCpuKernel。

(5) opInfo. workspaceSize 是配置内存空间，用于分配算子临时计算的内存(单位为 KB)，建议配置为 1024。

(6) opInfo. flagPartial/ opInfo. computeCost/ opInfo. flagAsync 为预留字段，默认值见程序清单 3-30 中设置的值。

另外推荐的 inputx. name 与 inputx. type 字段配置，可以在算子信息库中做校验，具体可参考相关开发文档。算子的适配插件开发和 TBE 是一样的，这里就不赘述了。

3.5　本章小结

本章介绍了 TBE 工具三种自定义算子开发方式(TBE DSL、TIK、AI CPU)，并为每种开发方式提供了简单实现示例。TBE 工具提供了多层灵活的算子开发方式，可以根据对硬件的理解程度自由选择，利用工具的优化和代码生成能力，生成昇腾 AI 处理器的高性能可执行算子。TBE 极大地满足了多样化需求，对昇腾 CANN 软件栈支持多样化的场景增加了灵活性。