|  |  |
| --- | --- |
| 文件编号 | FIL-C-250211-{EID}-{ZZX} |
| 上级文件编号 | FIL-C-240929-901068-MDX |

**PPL接入TPU-MLIR流程COP**

|  |
| --- |
| **注意**  本文件所含的信息均具有保密性质并仅限于内部使用。不得对本文件、本文件的任何部分或本文件所含的任何信息进行未经授权地使用、披露或复制。  ***NOTICE***  *The information contained in this document is confidential and is intended only for internal use. Unauthorized use, disclosure or copying of this document, any part hereof or any information contained herein is strictly prohibited.* |

# 目录

[目录 1](#_Toc17890)

[修订记录 2](#_Toc30379)

[第一章 总则 3](#_Toc28722)

[第1条 目的 3](#_Toc4521)

[第2条 适用范围 3](#_Toc11787)

[第3条 名词定义 3](#_Toc5570)

[第二章 PPL特性介绍 3](#_Toc10869)

[第4条 PPL使用的环境变量 3](#_Toc16775)

[第5条 PPL Local 内存分配机制 3](#_Toc4795)

[第三章 PPL与TPU-MLIR对接流程 4](#_Toc23059)

[第6条 将 pl 文件添加到 TPU-MLIR 项目中 4](#_Toc6307)

[第7条 添加算子封装 cpp 文件 4](#_Toc3803)

[第8条 编译生成libppl\_host.so 8](#_Toc1901)

[第9条 模型Codegen时调用PPL算子 9](#_Toc31032)

[第10条 JIT编译流程 10](#_Toc2494)

[第11条 JIT缓存介绍 10](#_Toc32042)

[第四章 注意事项 10](#_Toc28637)

[第五章 调试算子 11](#_Toc15635)

[第六章 Checklist 12](#_Toc241)

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 修订日期 | 作者 | 修订内容 |
| 1.0 | 2025.02.11 | 陈梁 | 首次制作 |

1. 总则
   1. 目的
      1. 撰写此文档的目的是为了介绍PPL编写的算子添加进TPU-MLIR的流程及原理
   2. 适用范围
      1. 本文档适用于算子开发人员、TPU-MLIR开发人员
   3. 名词定义
      1. **名词**：解释
2. PPL特性介绍
   1. PPL使用的环境变量
      1. PPL编译期会使用到一些环境变量：

|  |  |  |  |
| --- | --- | --- | --- |
| 环境变量 | 作用 | 来源 | 是否可以修改 |
| PPL\_PROJECT\_ROOT | 记录PPL编译器根目录 | envsetup.sh | 否 |
| PPL\_CACHE\_PATH | 指定ppl编译缓存目录，默认为~/.ppl/cache | 用户设置 | 是 |
| CACHE\_ITEMS | 指定ppl编译缓存kernel数量 | 用户设置 | 是 |

Table . 环境变量介绍

* 1. PPL Local 内存分配机制
     1. ppl编译器会自动进行 Local/L2 内存分配，原理是，ppl根据用户创建tensor/gtensor(L2)时，传入的shape来计算此tensor需要使用的 Local/L2 内存大小，然后根据tensor的生命周期，指令的bank conflict来分配实际的地址
     2. tensor/gtensor(L2)**构造函数的shape参数**是会用来计算tensor使用的local/L2**内存大小**，调用tensor/gtensor的**view/sub\_view方法，传入的shape不会参与内存分配计算，也不会对原始tensor的内存大小和实际数据产生任何影响**，view/sub\_view操作仅会改变tensor的shape/stride/offset等属性
     3. 当所有的 tensor 或 gtensor（L2）(不包含通过view/sub\_view函数生成的子tensor)在创建时，其shape值为常量时，我们称之为**静态shape**。相反，如果shape在创建时不是常量，则称为**动态shape**
     4. 当使用动态shape时，tensor的shape一定与某个kernel函数参数相关，需要在此**函数参数**定义时，加上**const**进行修饰，从而告诉ppl编译器，此参数与内存分配相关
     5. **静态shape**，内存分配发生在 PL 文件编译时
     6. **动态shape**内存分配在不同使用场景会有区别：  
        与tpu-mlir对接时，ppl的编译是jit编译，即调用算子时才会编译ppl算子代码，此时ppl编译器可以获取到实际的参数值，ppl编译器会将const修饰的函数参数当作常量来处理，此时内存分配是在 PL 文件编译时；ppl编译器会对const修饰的函数参数进行常量折叠优化，如果此函数参数与if的条件相关，ppl编译器会将if进行优化，只保留true分支  
        使用ppl\_compile.py测试时，对kernel的编译是aot编译，此时，PL 文件编译时无法确定tensor的大小，因此内存分配在算子运行时进行。当参数传入后，我们获取到实际的shape值后，再进行内存分配。可以在ppl\_compile.py测试时，添加--desc选项，使其也通过jit方式编译
     7. 内存分配有可能失败，若发生此情况，我们将返回错误码，告知用户失败原因，包括 Local 内存不足、L2 内存不足或其他错误

1. PPL与TPU-MLIR对接流程

PPL与TPU-MLIR对接主要分为以下步骤：

* + 1. 将 .pl 文件添加到 TPU-MLIR 项目中。
    2. 添加一个 cpp 文件，作用是封装算子的调用并为codegen提供入口函数。
    3. 编译生成libppl\_host.so
    4. 在模型codegen过程中调用用户开发的入口函数。

下面将以添加AddConst的ppl后端为例，详细介绍对接的流程细节

* 1. 将 pl 文件添加到 TPU-MLIR 项目中

推荐在ppl的开发环境下进行算子的开发和测试工作，当算子开发完成后，将pl源码直接复制到tpu-mlir/lib/PplBackend/src/下（版本变化可能导致目录变更，请注意）

* 1. 添加算子封装 cpp 文件

需要添加一个cpp文件，功能主要有：  
1.将codegen的参数格式转换为kernel算子的参数格式  
2.如果是动态shape，需要在cpp中进行tiling，寻找合适的切分shape  
3.调用kernel算子

cpp文件开发可以参考tpu-mlir/lib/PplBackend/src/add\_const\_fp.cpp；有以下几个需要注意的地方

* + 1. 添加必要的头文件

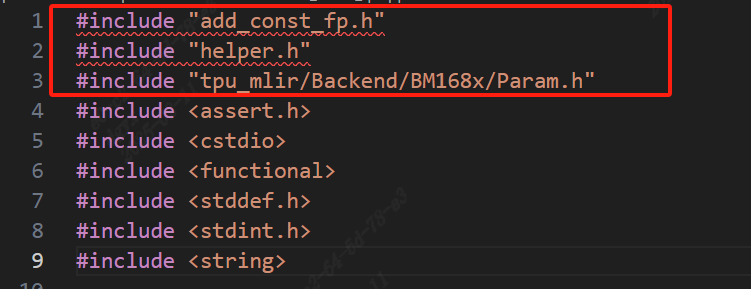


Figure . 示例图片

1.头文件需要引入与pl文件同名的头文件，例如算子写在add\_const\_fp.pl中，则需要添加add\_const\_fp.h  
2.添加helper.h(可选)，此头文件定义了一些公用的函数  
3.tpu\_mlir/Backend/BM168x/Param.h，此头文件定义了codegen的参数定义

* + 1. 以C语言格式生成动态库符号

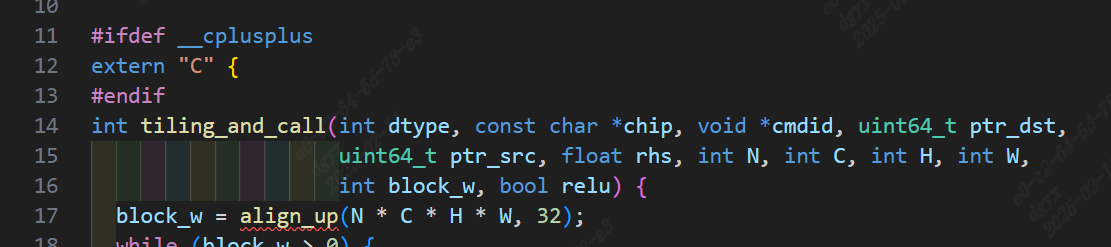


Figure . 示例图片

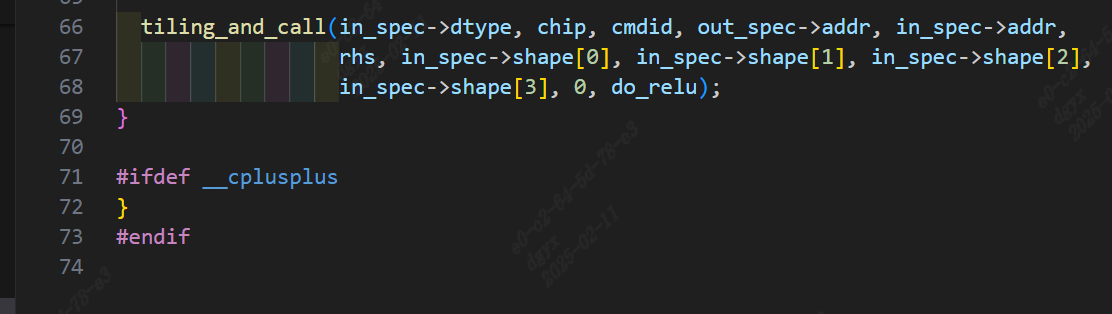


Figure . 示例图片

函数定义需要在extern "C"内

* + 1. 定义codegen入口函数

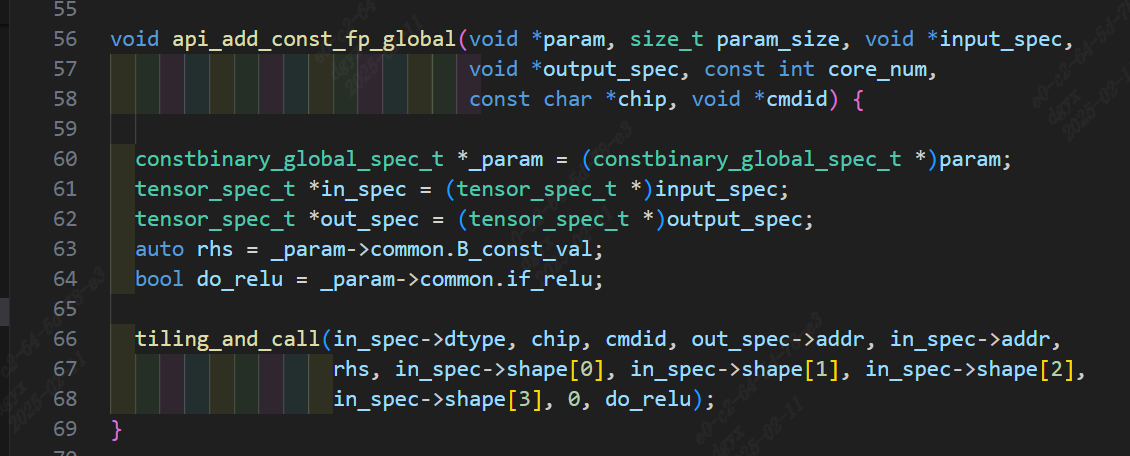


Figure . 示例图片

入口函数是为codegen提供的，用于调用 PPL 算子，此函数的参数是固定格式

|  |  |
| --- | --- |
| 参数名 | 作用 |
| param | codegen传入的算子参数 |
| param\_size | codegen传入的算子参数长度 |
| input\_spec | 输入tensor信息 |
| output\_spec | 输出tensor信息 |
| core\_num | 多核编程时使用的核数 |
| chip | 算子部署的芯片类型 |
| cmdid | 保证生成的指令序号连续 |

用户需要从param、input\_spec、output\_spec提取出算子需要的参数，然后调用pl中的算子

* + 1. 调用算子

pl文件经过ppl编译后会生成tpu-kernel的c/c++代码，实际要调用的算子函数与pl文件定义的函数同名，函数参数会在开头增加两个：chip和cmdid，然后pl中的**指针类型会变成uint64\_t类型**，示例如下：

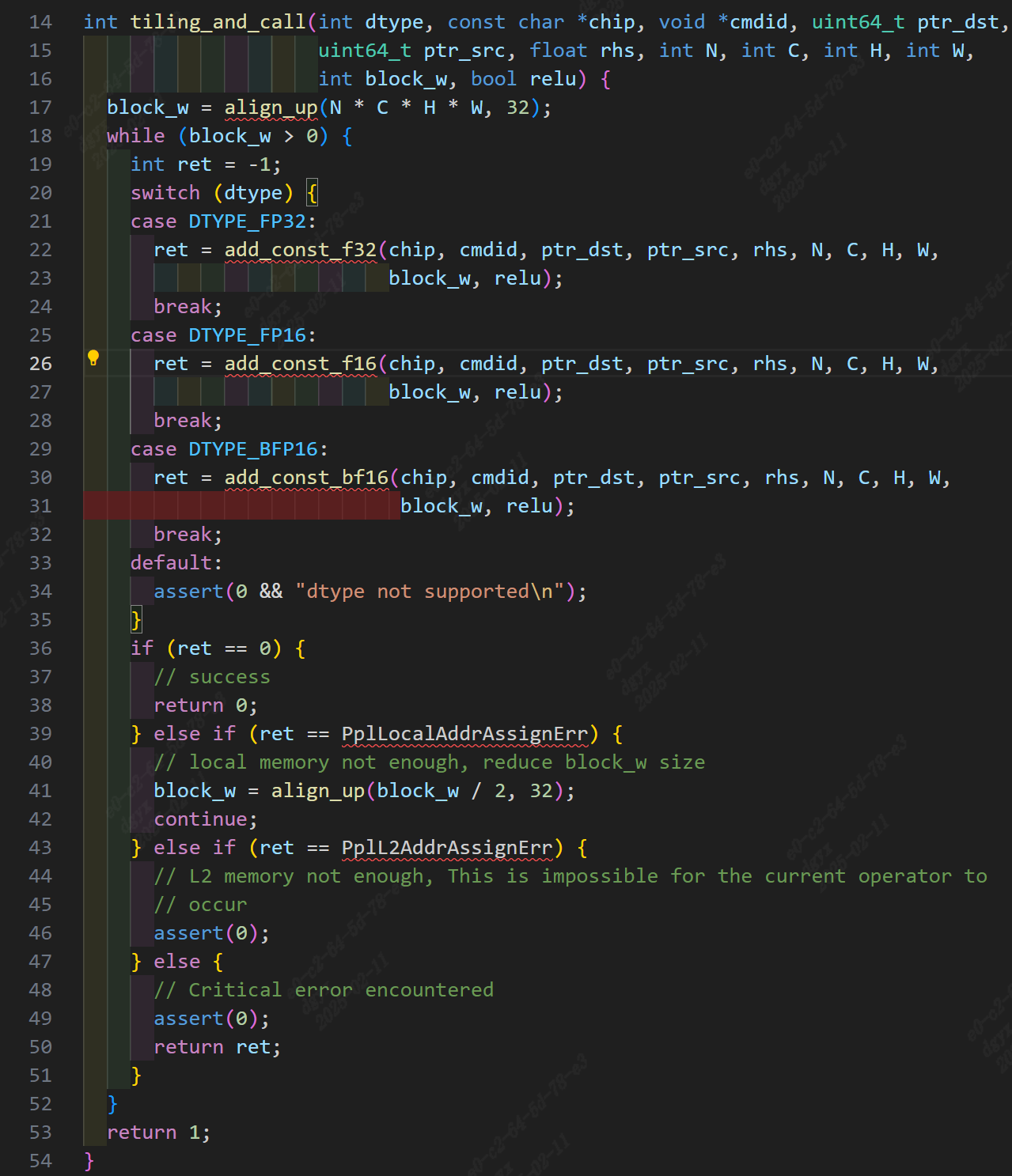


Figure . 示例图片

|  |
| --- |
| // 例如在pl文件中定义了如下kernel函数 \_\_KERNEL\_\_ void add\_const\_f32(float \*ptr\_dst, float \*ptr\_src,  float rhs, int N, int C, int H, int W,  const int block\_w, bool relu) // 经过ppl编译后，生成的实际函数定义如下: // 此函数定义在tpu-mlir/lib/PplBackend/build/host/add\_const\_fp.cpp中 int add\_const\_f32(const char \*chip, void \*pid\_node,  unsigned long long ptr\_dst\_v1,  unsigned long long ptr\_src\_v2,  float rhs\_v3, int32\_t N\_v4,  int32\_t C\_v5, int32\_t H\_v6,  int32\_t W\_v7, int32\_t block\_w\_v8,  uint8\_t relu\_v9) |

1.如果算子是动态shape（动态shape定义参见PPL Local 内存分配机制），则需要进行tiling，即判断算子调用后的返回值。如果是PplLocalAddrAssignErr或PplL2AddrAssginErr，则需要减小tensor尺寸相关参数（通常命名为block\_xxx），直到返回0；如果返回值不为0，且不是PplLocalAddrAssignErr或PplL2AddrAssginErr，则是其他编译错误，需要进行调试解决。

2.进行tiling时，尽量将tensor尺寸相关参数大小与LANE\_NUM/EU\_NUM进行对齐，因为这些参数的变化会导致pl算子进行重新编译，对齐后会极大减少算子的编译次数，并且大多数情况不会导致内存的浪费，这是因为TPU上的数据排布本来也是LANE\_NUM/EU\_NUM对齐的。

3.如果算子是静态shape，则不需要做tiling，直接调用算子函数，然后判断返回值是否为0，不为0则有问题，需要调试解决。

如果是静态shape，则上述函数可以修改为：

|  |
| --- |
| int call(int dtype, const char \*chip, void \*cmdid,  uint64\_t ptr\_dst, uint64\_t ptr\_src, float rhs,  int N, int C, int H, int W, int block\_w, bool relu) {  switch (dtype) {  case DTYPE\_FP32:  ret = add\_const\_f32(chip, cmdid, ptr\_dst, ptr\_src,  rhs, N, C, H, W, block\_w, relu);  break;  case DTYPE\_FP16:  ret = add\_const\_f16(chip, cmdid, ptr\_dst, ptr\_src,  rhs, N, C, H, W, block\_w, relu);  break;  case DTYPE\_BFP16:  ret = add\_const\_bf16(chip, cmdid, ptr\_dst, ptr\_src,  rhs, N, C, H, W, block\_w, relu);  break;  default:  assert(0 && "dtype not supported\n");  }  if (ret != 0) {  // Critical error encountered  assert(0);  }  return ret; } |

* 1. 编译生成libppl\_host.so
     1. 如果用户没有修改pl文件，则这一步通常不需要用户手动进行，如果修改了pl文件，则建议重新编译
     2. pl文件需要在编译模型前预先编译生成libppl\_host.so，然后在编译模型过程进行codegen时，会从libppl\_host.so中寻找算子入口
     3. 用户可以到tpu-mlir/lib/PplBackend/目录下，运行./build.sh进行手动编译

libppl\_host.so的编译内部流程如下：

* + 1. 删除ppl缓存目录，默认为~/.ppl/cache，可以通过PPL\_CACHE\_PATH环境变量进行修改
    2. 使用ppl-compile编译pl文件，生成同名的cpp文件（文件路径为tpu-mlir/lib/PplBackend/build/host/xxx.cpp），里面包含算子的入口、IR和一些辅助函数，**注意此时并未实际生成算子的tpu-kernel代码**，算子的tpu-kernel代码是在算子被tpu-mlir codegen调用时，jit生成的
    3. 将ppl自动生成的cpp文件与用户添加的cpp文件编译生成libppl\_host.so（文件路径为tpu-mlir/lib/PplBackend/build/libppl\_host.so）
  1. 模型Codegen时调用PPL算子

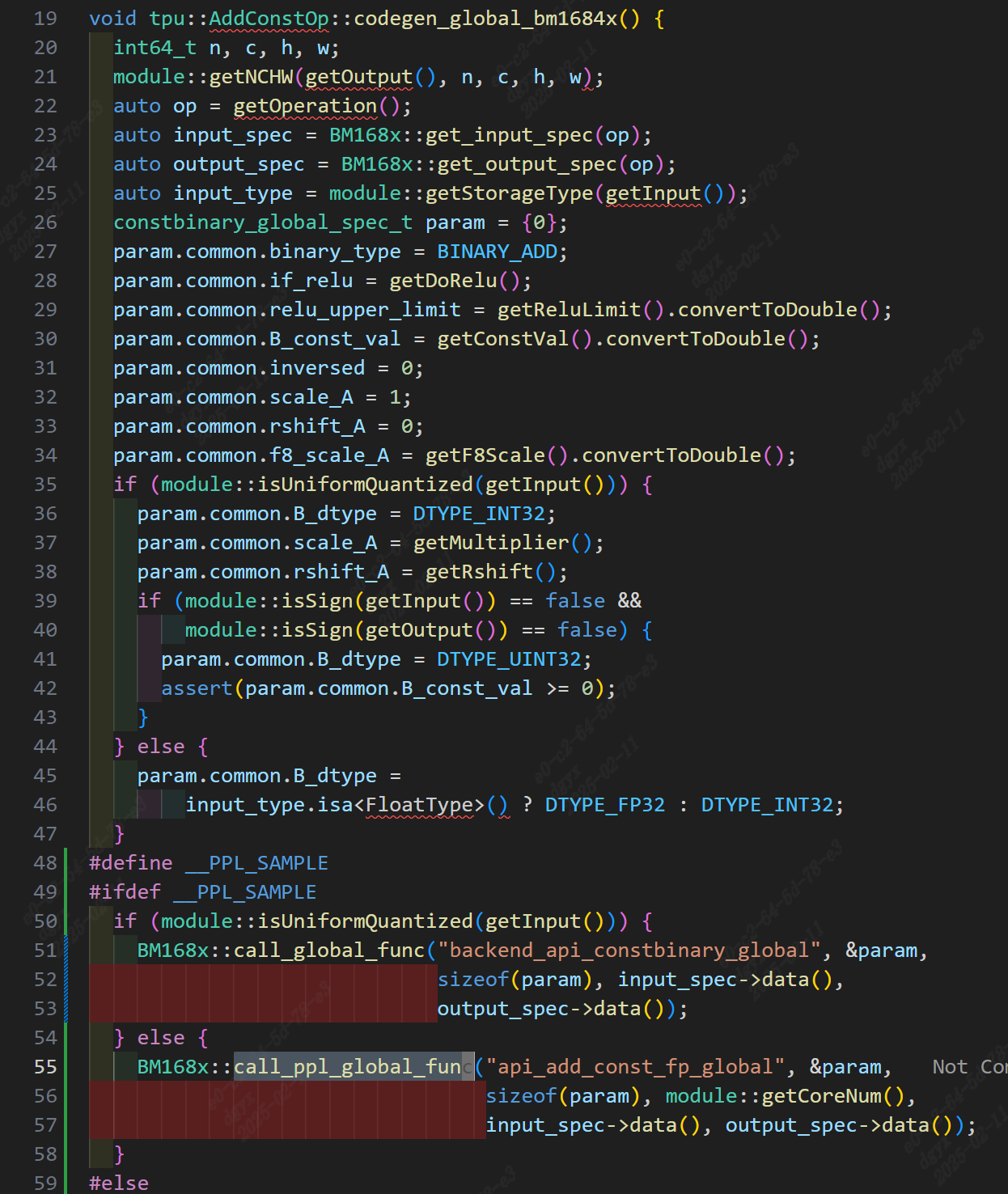


Figure . 示例图片

|  |
| --- |
| // 在Op的codegen代码中，添加调用PPL算子 BM168x::call\_ppl\_global\_func("用户自定义入口函数名字",  &param, sizeof(param),  module::getCoreNum(),  input\_spec->data(),  output\_spec->data()); |

以add\_const\_fp.pl为例，这其中的步骤有：

* + 1. 调用call\_ppl\_global\_func
    2. 通过dlopen动态加载libppl\_host.so，查找api\_add\_const\_fp\_global，并调用
    3. 调用pl文件编译后生成的add\_const\_f32函数
    4. JIT编译生成算子的tpu-kernel代码
  1. JIT编译流程

JIT编译是ppl内部自动进行，无需用户参与

* + 1. 将const修饰的算子函数参数填入算子IR;
    2. 调用PPL编译器生成tpu-kernel代码;
    3. 调用gcc生成kernel动态库
    4. 动态加载kernel动态库
    5. 如果为动态shape，则会调用内存分配函数
    6. 调用kernel函数
  1. JIT缓存介绍
     1. JIT生成的算子动态库、tpu-kernel代码存放在~/.ppl/cache目录下
     2. JIT时，会将chip、pl文件名、函数名、const修饰的参数值，拼接为一个字符串，计算hash值，并以hash值生成目录存放动态库，如果目录存在，则直接使用，不会重复编译
     3. 在 JIT 调用过程中，如果调用失败（例如，内存分配失败），相关信息会被缓存到 ~/.ppl/cache/flag.txt 文件中。因此，在调试过程中，如果修改了 PL 文件，建议手动删除 ~/.ppl/cache 目录。这可以避免 JIT 返回失败的情况，是因为之前的失败被缓存了

1. 注意事项
   * 1. 在 PPL 中，如果 kernel 函数的参数与控制流相关（例如，作为 if 条件），可以将其定义为 const 类型，这样可以使ppl编译器优化掉不会实际进入的if分支；这样做的优点是优化掉无用分支后，Local 内存的总使用量会降低，tensor可分配的内存可能会增加；缺点是，当const修饰的参数发生变化时，算子的tpu-kernel代码会重新生成，可能增加模型编译耗时
     2. 开发过程中，如果对接时存在异常，建议先清理cache目录（${PPL\_CACHE\_PATH}默认为~/.ppl/cache），然后重新编译libppl\_host.so
     3. 使用动态shape时，ppl\_compile.py测试与tpu-mlir对接内存分配结果可能不一样，tpu-mlir对接的block大小可能大于ppl\_compile.py测试时
2. 调试算子
   * 1. 如果ppl算子加入到tpu-mlir中后，运算结果不正确，或者出现core dump，可以先确认下 tpu-kernel 代码算子函数入参是否正确，可以通过gdb断点打到 tpu-kernel 函数，然后查看参数是否正确，流程如下

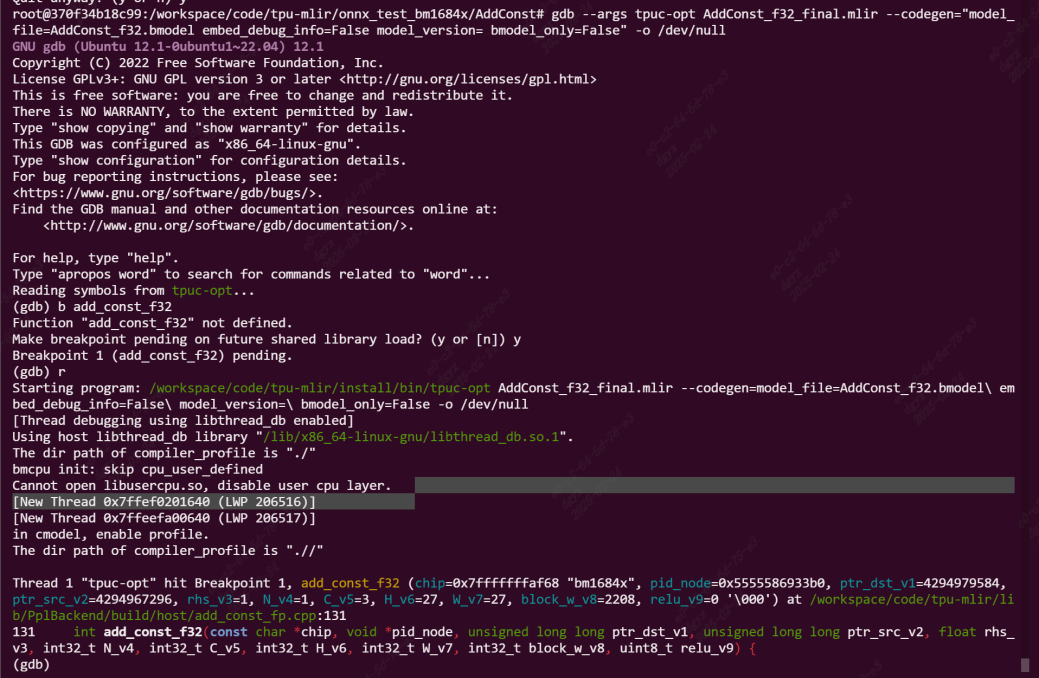


Figure . 示例图片

|  |
| --- |
| # 目前ppl算子动态库是编译的DEBUG版本 # 可以在转模型codegen这一步使用gdb进行调试 # tpu-kernel函数名与pl文件中算子函数名一致 gdb --args tpuc-opt AddConst\_f32\_final.mlir --codegen="model\_file=AddConst\_f32.bmodel embed\_debug\_info=False model\_version= bmodel\_only=False" -o /dev/null b add\_const\_f32 y r  # 如果函数入参与codegen传入不一致，可能是cpp文件函数参数数量错误、参数位置错误、传参错误 |

1. Checklist
   * 1. Checklist 表格：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cat. | 编号 | Check 项目 | Check 方法与参考值 | Check 结果 | PR & 时间 |
| 添加算子封装 cpp 文件 |  | 必要的头文件是否添加 | 检查是否添加pl文件同名的头文件 参考值：是。 |  |  |
|  | 必要的头文件是否添加 | 检查是否添加codegen参数结构体定义头文件param.h 参考值：是。 |  |  |
|  | 函数定义是否在extern "C" 中 | 检查是否添加extern "C" 参考值：是。 |  |  |
|  | 动态shape是否有添加tiling函数 | 检查pl中kernel函数是否有const类型，如果有，则检查cpp文件是否有tiling过程 参考值：是。 |  |  |
|  | codegen入口函数参数是否正确 | 与add\_const\_fp.cpp对照 参考值：是。 |  |  |