|  |  |
| --- | --- |
| 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](#_Toc119)

[修订记录 3](#_Toc16946)

[第一章 总则 4](#_Toc21962)

[第1条 目的 4](#_Toc23134)

[第2条 适用范围 4](#_Toc29052)

[第3条 名词定义 4](#_Toc21693)

[第二章 PPL特性介绍 4](#_Toc11159)

[第4条 PPL使用的环境变量 4](#_Toc30180)

[第5条 PPL Local 内存分配机制 4](#_Toc30814)

[第三章 python前端与torch-tpu plugin-in 对接流程 5](#_Toc8959)

[第6条 添加python算子源码 5](#_Toc19036)

[第7条 调用python算子 6](#_Toc26171)

[第四章 c++前端与torch-tpu plugin-in 对接流程 7](#_Toc1452)

[第8条 添加c++算子源码 7](#_Toc4955)

[第9条 调用c++算子 7](#_Toc5819)

[第五章 PPL与torch-tpu plugin-in 对接原理 12](#_Toc30214)

[第六章 plugin-in 对接 DEBUG 方式 12](#_Toc4049)

[第七章 c++前端与torch-tpu built-in 对接流程 13](#_Toc32668)

[第八章 使用PPL需要注意的地方 13](#_Toc3106)

[第10条 Checklist 14](#_Toc22671)

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 |
| 1.1 | 2025.06.01 | 向少雄 | 更新流程:ppl调用tpudnn |
| 1.2 | 2025.07.16 | 陈梁 | 新增torch-tpu built-in 模式编译流程 |

1. 总则
   1. 目的
      1. 撰写此文档的目的是为了介绍PPL编写的算子如何与torch-tpu配合使用，并在例如TGI框架中适配
   2. 适用范围
      1. 本文档适用于算子开发人员、torch-tpu开发人员、TGI框架等开发人员
   3. 名词定义
      1. **plugin-in**：ppl算子作为第三方算子与torch-tpu对接，使用者为torch-tpu的用户或开发人员
      2. **built-in**：ppl算子作为torch-tpu的内置算子，使用者为torch-tpu的开发人员
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：

Table . 环境变量介绍

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |
| CHIP | 指定ppl编译算子时，使用的芯片类型 | 用户设置/whl包自动从torch-tpu获取 | 是 |

* 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(c++前端)或pl.constexpr(python前端)\*\*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ppl算子代码编译时
     6. **动态shape**内存分配在不同的前端会有区别：  
        **python前端**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pl.constexpr修饰的函数参数当作常量来处理，此时内存分配还是在算子代码编译期；ppl编译器会对pl.constexpr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。  
        **c++前端**是aot编译，pl文件编译时无法确定tensor的大小，因此内存分配是在算子运行时进行的，当参数传入后，我们获取到实际的shape值后，再进行内存分配。
     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. python前端与torch-tpu plugin-in 对接流程

以TGI框架下的text-generation-inference/server/tests/soph\_test/test\_ppl.py为例，可以使用python test\_ppl.py运行sample

* 1. 添加python算子源码

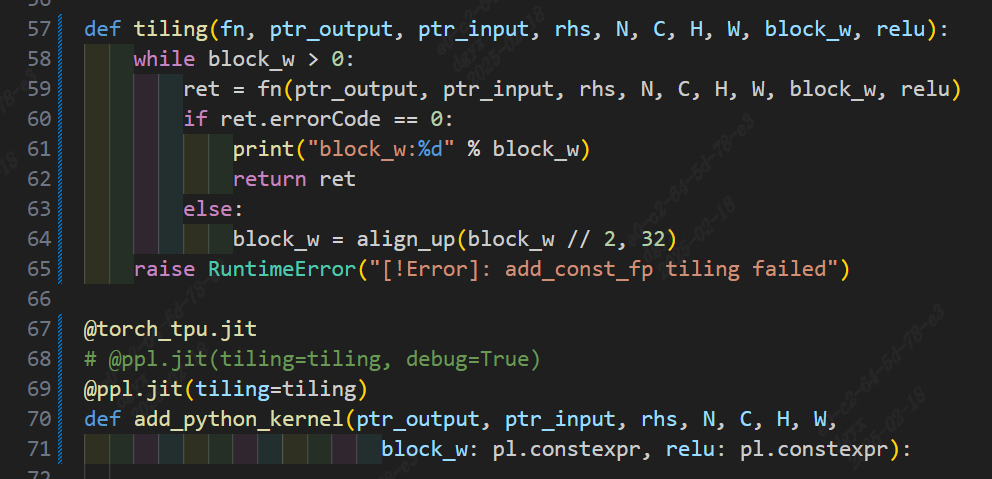


Figure . 示例图片

* + 1. 需要在算子函数前加上@torch\_tpu.jit
    2. 如果是动态shape的算子，编译时可能存在内存不足的情况，因此需要添加一个tiling函数来做切分，以便数据在TPU上可以放下，tiling函数的参数第一位为fn，其他与算子函数一致
    3. 如果需要进行debug，则可以将@ppl.jit加上debug=True

|  |
| --- |
| # 静态shape不需要tiling @ppl.jit def add\_python\_kernel()  # 动态shape添加tiling函数 @ppl.jit(tiling=tiling) def add\_python\_kernel()  # 动态shape添加tiling函数，同时编译debug版本算子动态库 @ppl.jit(tiling=tiling, debug=True) def add\_python\_kernel() |

* 1. 调用python算子

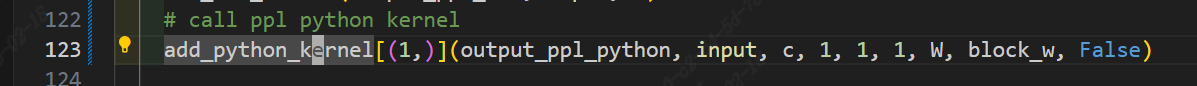


Figure . alt text

* + 1. 调用算子函数与调用普通python函数区别仅在于调用算子函数需要添加[(x,)]或[(x,y)]参数
    2. [(x,)]表示算子使用x个block，[(x,y)]表示算子使用x个group，y个block
    3. 调用python算子时，会使用ppl编译器对算子进行编译，结果存放在${PPL\_CACHE\_PATH}，如果算子代码及pl.constexpr参数没有变化不会重复编译
    4. 调用算子失败时，例如tiling失败，也会被缓存，所以在算子开发时，如果调用失败，先删除缓存目录，避免缓存影响

1. c++前端与torch-tpu plugin-in 对接流程

以TGI框架下的text-generation-inference/server/tests/soph\_test/test\_ppl.py为例，可以使用python test\_ppl.py运行sample

* 1. 添加c++算子源码

将c++算子源码add.pl添加到工作区

* 1. 调用c++算子
     1. 首先使用ppl.pl\_extension.load加载算子，ppl.pl\_extension.load的输出为算子动态库路径；全局可以只加载一次，所以可以在\_\_init\_\_函数中调用；加载后动态库和内存分配函数存储在${PPL\_CACHE\_PATH}，如果pl文件及const参数没有变化不会重复编译

Table . ppl.pl\_extension.load输入参数

|  |  |  |
| --- | --- | --- |
| 参数名字 | 作用 | 是否可以缺省 |
| name (str) | 算子函数名字 | 否 |
| sources (list[str]) | pl和cpp文件路径, 如果是相对路径需要关注当前运行目录 | 否 |
| extra\_cflags (list[str]) | 附加C/ c++编译标志 | 是 |
| extra\_plflags (list[str]) | 附加ppl C/ c++编译标志 | 是 |
| extra\_ldflags (list[str]) | 附加链接器标志 | 是 |
| extra\_include\_paths (list[str]) | 附加include路径 | 是 |

* + 1. 如果是动态shape，需要循环调用内存分配函数，每次减小block尺寸，直到内存分配函数返回0；内存分配函数入参与算子函数一致，内存分配函数主要关注3个返回值，0表示成功，0x11表示Local内存分配失败，0x1A表示L2内存分配失败

Table . 内存分配函数返回值

|  |  |  |
| --- | --- | --- |
| 枚举 | 值 | 说明 |
| PplLocalAddrAssignErr | 0x11 | Local内存分配失败 |
| FileErr | 0x12 |  |
| LlvmFeErr | 0x13 |  |
| PplFeErr | 0x14 | AST转IR失败 |
| PplOpt1Err | 0x15 | 优化pass opt1失败 |
| PplOpt2Err | 0x16 | 优化pass opt2失败 |
| PplFinalErr | 0x17 | 优化pass final失败 |
| PplTransErr | 0x18 | 代码生成失败 |
| EnvErr | 0x19 | 环境变量异常 |
| PplL2AddrAssignErr | 0x1A | L2内存分配失败 |
| PplShapeInferErr | 0x1B | shape推导失败 |
| PplSetMemRefShapeErr | 0x1C |  |
| ToPplErr | 0x1D |  |
| PplTensorConvErr | 0x1E |  |
| PplDynBlockErr | 0x1F |  |

|  |
| --- |
| def add\_cxx\_kernel(output, input, c):  # load ppl code  # ppl.pl\_extension.load  # Load and compile an extension module.   # Args:  # name (str, required): Module name. (Choose wich kernel\_func in .pl should be load)  # sources (List[str], required): Source files (eg. ["a.pl", "a\_tilling.cpp"])  # chip (str, optional): chip\_type (eg. ppl.Chip.bm1690)  # extra\_cflags (List[str], optional): Additional C/C++ flags. (eg. ['-g', '-DDEBUG'])  # extra\_plflags (List[str], optional): Additional ppl-specific C/C++ flags.  # extra\_ldflags (List[str], optional): Additional linker flags. (eg. ["-L{lib\_path}", "-l{lib\_name}"])  # extra\_include\_paths (List[str], optional): Additional include paths. (eg. ["a.h", "b.h"])   # Returns:  # module: The compiled Torch extension module.   # Example:  # ppl.pl\_extension.load(  # name="add",  # sources=["add\_dyn\_block.pl","add\_dyn\_block\_tiling.cpp"],  # chip=ppl.Chip.bm1690,  # extra\_plflags=['-g', '-DDEBUG'])  # )  module = ppl.pl\_extension.load("add\_const\_mc\_f32", ["add.pl", "add\_tiling.cpp"],  ppl.Chip.bm1690, extra\_plflags=['-g', '-DDEBUG'])  # add\_const\_f32\_torch is defined in tilling.cpp  module.add\_const\_f32\_torch(output, input, c, 1,1,1,input.numel(), False,  torch\_tpu.tpu.current\_stream().tpudnn\_handle) |

add\_tiling.cpp示例（文件名请勿使用add.cpp）

|  |
| --- |
| #include <stdexcept> #include "add.h" // header file's name is same as add.pl  #ifdef \_\_cplusplus extern "C" { #endif // tilling func int add\_const\_tiling(gaddr\_t ptr\_dst, gaddr\_t ptr\_src, float rhs, int N, int C, int H, int W,  int relu) {  int ret = -1;  int core\_num = 8;  int block\_w = W;  while (block\_w > 1) {  int core\_num = (W + block\_w - 1) / block\_w;  ret = add\_const\_mc\_f32(ptr\_dst, ptr\_src, rhs, N, C, H, W, block\_w, core\_num, relu);  if (ret == 0) {  break;  } else if (ret == PplLocalAddrAssignErr) {  block\_w = block\_w / 2;  continue;  } else if (ret == PplL2AddrAssignErr) {  throw std::runtime\_error("[!Error]:L2 memory not enough\n");  } else {  throw std::runtime\_error("[!Error]:ppl compilation failed\n");  }  }  return ret; }  #ifdef \_\_cplusplus } #endif // bind func int add\_const\_f32\_torch(torch::Tensor &ptr\_res, torch::Tensor &ptr\_inp,  float rhs, int N, int C, int H, int W,  int relu, uint64\_t handle\_t) {  tpudnnHandle\_t handle = tpudnnGetHandle(handle\_t); // Fixed writing style  tpudnnTensor\_t res = TPUGenerateTpudnnTensor(handle, ptr\_res); // Fixed writing style  tpudnnTensor\_t inp = TPUGenerateTpudnnTensor(handle, ptr\_inp); // Fixed writing style  return add\_const\_tiling((gaddr\_t)res.addr, (gaddr\_t)inp.addr, rhs, N, C, H, W,  relu); }  PYBIND11\_MODULE(TORCH\_EXTENSION\_NAME, m) { // Fixed writing style  m.def("add\_const\_f32\_torch", &add\_const\_f32\_torch,  py::arg("ptr\_res"),  py::arg("ptr\_inp"),  py::arg("rhs"),  py::arg("N"),  py::arg("C"),  py::arg("H"),  py::arg("W"),  py::arg("relu"),  py::arg("handle\_t") = 0,  "add\_const\_f32\_torch"); } |

* + 1. 使用module.xxxx调用算子，算子名为.cpp中定义的算子wrap后的名字，函数最后一个参数为tpudnn\_handle，其他为算子函数参数；一定要保证传入参数的浮点类型和整型与算子函数定义时一致

1. PPL与torch-tpu plugin-in 对接原理
   * 1. python前端与c++前端都是先生成算子动态库，然后通过torch\_tpu内的tpudnn使用指令缓存方式调用算子动态库
     2. python前端与c++前端都会将算子动态库缓存，缓存目录在${PPL\_CACHE\_PATH}
     3. 算子torch bind函数与算子函数具有相同的参数列表, 并额外接收来自于tpu-train的handle
     4. 算子函数入参会从python端传递到torch bind c++端，再经过handle注册与地址转换后传递给tilling函数，注意tiling函数与pl中定义的算子函数参数列表保持一致
     5. 算子函数入参会从python端传递到ppl tilling c++端，这个过程会将参数做转换，int和bool类型等整型会转成int64，float/double会转成double，同时将参数的顺序保存；tpudnn接收到的参数为std::vector<uint64\_t>(global地址)/std::vector<int64>(整型参数)/std::vector<double>(浮点型参数)以及std::vector<int64\_t>的参数序号，tpudnn再将参数按顺序组成一块内存，将首地址传递给入口函数，所以在python端调用算子时，一定要注意参数类型不要传错
2. plugin-in 对接 DEBUG 方式
   * 1. 如果算子结果不正确，首先删除缓存目录，避免缓存影响
     2. 如果想在TGI中，gdb调试算子tpu-kernel代码，例如确认传递到算子的参数是否正确，可以如下操作

|  |
| --- |
| # python前端可以使用@ppl.jit(debug=True) # c++前端可以使用ppl.pl\_extension.load(extra\_plflags=['-g', '-DDEBUG']) # 编译debug版本算子动态库 # # 可以使用gdb运行python，然后断点打到算子函数名字，然后进入kernel函数 # test\_ppl.py运行过程会打印生成的算子动态库存储路径，可以在这里找到算子的tpu-kernel代码 gdb --args python test\_ppl.py b 算子名字 r  # 如果需要调试c++前端的内存分配代码，则可以断点打在算子函数名字\_check\_mem # 例如算子为add\_const\_f32，则内存分配函数名字为add\_const\_f32\_check\_mem gdb --args python test\_ppl.py b add\_const\_f32\_check\_mem r |

1. c++前端与torch-tpu built-in 对接流程

此对接方式，ppl算子会在torch-tpu编译时，与torch-tpu的算子动态库一起编译，生成的算子动态库会在torch-tpu内部。由于torch-tpu需要算子接口提供runtime需要的stream和module，因此在编译ppl算子时，需要设置mode=6，使ppl输出对应格式的host端代码，此时编译命令如下：

|  |
| --- |
| ppl-compile add\_dyn\_block.pl --chip sg2260e --O3 --mode=6 --o build |

运行命令后，会在build目录下生成以下文件：

Table . 编译生成文件

|  |  |
| --- | --- |
| 文件名 | 作用 |
| device | 编译后生成的device端代码存放目录 |
| host | 编译后生成的host端代码存放目录 |
| include | 编译后生成的头文件存放目录 |

1. 使用PPL需要注意的地方
   * 1. 尽量在ppl开发环境进行算子调试，确认无误，再添加到项目中
     2. 对于**python前端**，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
     3. 对于**c++前端**，if分支无法优化，需要尽量减少算子中的if，可以写成不同的算子，这是因为内存分配时会认为if的所有分支都会进入，这样每个分支使用的内存是累加的，这样会导致内存分配的结果偏小
     4. 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 ，python前端缓存位于 ${PPL\_CACHE\_PATH}/python 下，c++前端缓存位于 ${PPL\_CACHE\_PATH}/cxx 下
     5. 如果使用ppl whl包则可以不管环境变量 ${CHIP} ，如果使用ppl\_v1.xxx.tar.gz Release包，则需要手动设置CHIP类型，或者使用**CHIP=bm1690 python xxx.py**方式运行
     6. 一定要注意调用算子时传入的参数类型，特别是c++前端，标量如果期望使用float类型，则在python端传递参数一定要是float类型，例如期望在算子函数使用float型的1，那么传递参数一定是1.0而不是1，否则可能出错
   1. Checklist
      1. Checklist 表格示例：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cat. | 编号 | Check 项目 | Check 方法与参考值 | Check 结果 | PR & 时间 |
| 添加ppl算子 |  | python前端装饰器 | 检查是否添加@torch\_tpu.jit和@ppl.jit 参考值：是。 |  |  |
|  | 添加tiling函数 | 动态shape是否有添加tiling函数 参考值：是。 |  |  |
|  | 核数量是否正确 | 检查算子代码中设置的核数与运行时设置的核数是否一致 参考值：是。 |  |  |
|  | 参数传递是否正确 | 通过gdb检查python调用算子的参数与tpu-kernel函数接收参数是否一致 参考值：是。 |  |  |