文件编号	FIL-C-241227-901410-SWX
上级文件编号	FIL-C-240929-901068-MDX

PPL 编译器开发

目录

目录 1

修订记录 3

第一章 *总则* 4

第 1 条 目的 4
第 2 条 适用范围 4
第 3 条 名词定义 4

第二章 *环境配置* 4

第 4 条 从 gerrit 上下载并配置 PPL 工程 4
第 5 条 编译方式 5

第三章 *工程目录介绍* 5

第 6 条 根目录简介 5

第四章 *Codegen 流程介绍* 6

第 7 条 类 C/C++ 风格的编程接口 Codegen 流程介绍 7
第 8 条 类 python 风格的编程接口 Codegen 流程介绍 7

第五章 *C/C++ 指令添加* 7

第 9 条 前端添加算子调用函数 7
第 10 条 在 IR 中添加指令的 OP 定义 8
第 11 条 添加前端函数到 IR 的转换 9
第 12 条 添加 IR 到 tpu_kernel 代码的转换 9
第 13 条 正确性验证 10

第六章 *Python 指令添加* 11

第 14 条 前端添加算子 api 函数 11
第 15 条 添加具体实现 13

第 16 条 在 ppl.cc 中实现 create_pool_avg 13
第 17 条 在 example/python 目录下添加对应的测试例 14
第 18 条 测试例加入回归 16

第七章 *Pass 添加* *16*

第 19 条 定位优化 pass 的添加位置 16
第 20 条 Pattern Match 机制概述 17
第 21 条 Pattern Match 的添加与执行 17
第 22 条 Pattern Match 的优点 19
第 23 条 添加优化 Pass 19
第 24 条 构建测试样例 20

第八章 *芯片添加* *21*

第 25 条 添加芯片信息 21
第 26 条 添加芯片 emit 支持 22
第 27 条 添加芯片 runtime 支持 23
第 28 条 正确性检测 24
第 29 条 添加线上回归检测 25

第九章 *checklist* *26*

修订记录

版本号	修订日期	作者	修订内容
1.0	2025.1.10	李胜超、 雷晓军、 郭尚霖、 路韬臣	首次制作

第一章 总则

第1条 目的

1) 介绍 PPL 编译器工程开发的方法。

第2条 适用范围

1) 所有需要开发 PPL 编译器工程的员工。
2) 所有使用 PPL 开发 TPU 算子的人员。

第3条 名词定义

1) **PPL**：Primitive Programming Language，PPL 既是语言也是编译器，作为语言它提供了一套类 C/C++ 和一套类 python 风格的编程接口；作为编译器它会对代码进行内存管理、流水并行等优化。
2) **MLIR**：Multi-Level Intermediate Representation，多层次中间表示，用于优化和转换代码的框架。
3) **IR**：intermediate representation，是在计算机科学和编译器设计中使用的概念，它是一种中间形式的程序表示，用于在不同编译阶段之间传递和处理代码，PPL 会将代码转成 mlir IR。
4) **Pass**：编译器中用于优化或转换 IR 的逻辑单元。
5) **Dialect**：MLIR 中的方言，用于定义特定领域的操作和类型。
6) **OP**：Operation，IR 中的一个具体操作或指令。

第二章 环境配置

第4条 从 gerrit 上下载并配置 PPL 工程

```
# 安装 git-lfs
curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash
sudo apt-get install git-lfs
# 如何使用 git-lfs
https://wiki.sophgo.com/pages/viewpage.action?pageId=127019078

# clone PPL 工程
git config --global http.sslVerify false

git clone https://shengchao.li@gerrit-ai.sophgo.vip:8443/a/ppl && (cd ppl && mkdir -p .git/hooks && curl -Lo `git rev-parse --git-dir`/hooks/commit-msg https://shengchao.li@gerrit-ai.sophgo.vip:8443/tools/hooks/commit-msg; chmod +x `git rev-parse --git-dir`/hooks/commit-msg)
```

```
git config --global http.sslVerify true

# 配置 http
cd ppl
git config lfs.https://gerrit-ai.sophgo.vip/ppl.git/info/lfs.locksverify false
git config --global credential.helper store
git config http.sslVerify false

# 获取第三方库
cd ppl/third_party

scp
guest@172.22.12.22:/data/ppl_third_party/llvm+mlir-17.0.0-x86_64-linux-gnu-ubuntu-18.04-release.tar.gz .
(password:123456)

tar -xvhf llvm+mlir-17.0.0-x86_64-linux-gnu-ubuntu-18.04-release.tar.gz
mv llvm+mlir-17.0.0-x86_64-linux-gnu-ubuntu-18.04-release/ llvm_release/
```

第5条 编译方式

```
# 使用 PPL 提供的 docker 环境
cd docker
# 当前已处于 ppl/docker 目录下
./build.sh
cd ..
# 当前已处于 ppl/ 目录下
docker run --privileged -itd -v $PWD:/workspace --name ppl sophgo/ppl:latest
docker exec -it ppl bash

# 使用 sophgo 通用镜像
docker run --privileged -itd -v $PWD:/workspace --name ppl sophgo/tpuc_dev:latest
docker exec -it ppl bash

# PPL 工程编译
cd /workspace
source envsetup.sh
./build.sh (DEBUG)
```

第三章 工程目录介绍

第6条 根目录简介

1) bin/ 目录存放 PPL 编译工具(ppl-compile)的源码文件和 LLVM 前端 codegen 源码。

2) doc/ 目录存放《PPL快速入门指南.pdf》和《PPL开发参考手册.pdf》的 rst 源文件。
3) docker/ 目录存放 PPL 专用镜像的编译脚本。
4) example/ 目录存放使用 C/C++ 和 python 指令接口开发的示例算子。
5) inc/ 目录存放 PPL 指令头文件。
6) include/ 目录存放 PPL 工程依赖的头文件。
7) lib/ 目录存放 PPL 工程源文件；包含 PPL dialect 和 PPL_Op 定义。
8) python/ 目录存放使用 python 开发的辅助工具、 python 前端支持源码、PPL 生成代码依赖的辅助函数和脚本。

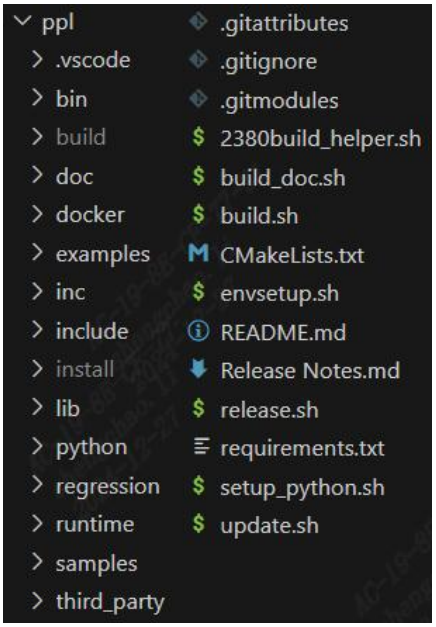

Figure 1

9) regression/ 目录存放 ci 和 daily_regression 验证示例。
10) runtime/ 目录存放不同芯片的 cmodel 运行时库和依赖的头文件。
11) samples/ 目录存放 PPL 代码在不同芯片不同模式下运行的示例。
12) third_party/ 目录存放第三方库。
13) build_doc.sh 为文档编译脚本。
14) build.sh 为 PPL 工程编译脚本。
15) envsetup.sh 为环境变量设置脚本。
16) release.sh 为 release 包编译脚本。
17) update.sh 为 runtime 库更新脚本。
18) setup_python.py 为 whl 包编译脚本。

第四章 Codegen 流程介绍

第7条 类 C/C++ 风格的编程接口 Codegen 流程介绍

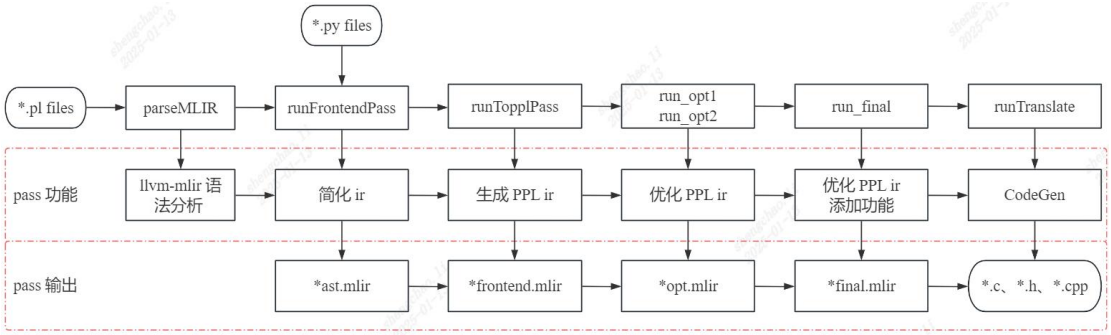

Figure 2

1) 统一使用 .pl 后缀来保存类 C/C++ 风格编程接口开发的算子。
2) 获取到 pl 文件后，通过 parseMLIR 函数进行语法分析和词法分析，得到最初始版本的 IR。
3) runFrontendPass 函数是一个复杂的前端优化步骤，结合了多种优化技术（如 CSE、LICM、Mem2Reg、Affine 替换等），对输入的 IR 进行简化和优化，为后续的 IR 转换和后端优化奠定基础。输出 *ast.mlir。
4) runTopplPass 函数将输入 IR 中的 call func 内联处理，并转换成 PPL op。去除冗余算术操作，输出 *frontend.mlir。
5) run_opt1 和 run_opt2 函数进行图优化,清理无用代码。推导 shape 信息并设置 shape。进一步优化 IR 的计算和内存访问。输出 *opt.mlir。
6) run_final 函数为编译器优化流程的最终阶段，涉及流水并行优化、Tensor 转换、规范化、公共子表达式消除、内存优化与分配、寄存器分配以及动态块处理等关键步骤，输出 *final.mlir。
7) runTranslate 函数将 *final.mlir 翻译为可直接在设备上运行的 *.c、*.h 和 host 端的 *.cpp 代码，

第8条 类 python 风格的编程接口 Codegen 流程介绍

1) 类 python 与类 C/C++ 编程接口 Codegen 流程唯一的区别为：前者使用 python 解释器进行语法和词法分析，后者使用 llvm-clang 进行分析。

第五章 C/C++ 指令添加

以 pool_avg 算子添加为例，介绍 C/C++ 指令添加的流程。

第9条 前端添加算子调用函数

1) PPL 前端函数定义文件在 PPL 工程的 inc/ 目录下,主要包含以下几个头文件：

文件名	说明

ppl_defs.h	PPL 前端使用的宏定义
ppl_types.h	内置结构体、枚举类型等定义
ppl_func.h	不属于 TPU/DMA 操作的函数
ppl_utils.h	对标量操作的辅助函数
ppl_dma_func.h	DMA 操作相关函数
ppl_sdma_func.h	SDMA 操作相关函数
ppl_tiu_func.h	TIU 操作相关函数

2) 根据你要添加的算子类型在 ppl_dma_func.h、ppl_sdma_func.h 或 ppl_tiu_func.h 中添加前端函数。以下是函数定义示例：

```
namespace ppl {
namespace sdma{
    template <typename DataType0, typename DataType1>
    void pool_avg(tensor<DataType0> &dst, tensor<DataType1> &src, dim2 *kernel,
                  padding_t *pad, dim2 *stride, dim2 *dilation, dim2 *ins,
                  int scale, int rshift);
    template <typename DataType0, typename DataType1>
    void pool_avg(tensor<DataType0> &dst, tensor<DataType1> &src, dim2 *kernel,
                  padding_t *pad, dim2 *stride, dim2 *dilation, int scale,
                  int rshift) {
        return pool_avg(dst, src, kernel, pad, stride, dilation, (dim2 *)nullptr, scale, rshift);
    }
}
}
```

3) 该函数定义了一个 pool_avg 计算的函数，有八个输入 src，kernel，pad，stride，dilation，ins，scale，rshift，参数的含义正如名字所代表的含义一样；一个输出 dst，参数类型是 tensor，tensor 的数据类型为 DataType0。

第10条 在 IR 中添加指令的 OP 定义

1) 前端语言中的指令函数会被转成 mlir IR 中的 OP ，因此需要在 IR 中定义对应的 OP；OP 定义在 include/ppl/Dialect/Ppl/IR/PplOps.td 中，OP 在这里名字是 AvgPool2DOp。

```
def PPL_AvgPool2DOp: PPL_Op<"ins_avgPool2D", [MemoryEffects<[MemWrite]>,
                                DeclareOpInterfaceMethods<TiuInterface>,
                                DeclareOpInterfaceMethods<MemInterface>,
                                DeclareOpInterfaceMethods<EmitInterface>,
                                DeclareOpInterfaceMethods<ShapeInferenceOpInterface, ["inferShapes"]>]> {
    let summary = "ins avgPool2D";
    let arguments = (ins PPL_MemRef:$rst,  PPL_MemRef:$input,
                    PPL_IntLike:$kernel, PPL_IntLike:$pad,
                    PPL_IntLike:$stride, PPL_IntLike:$dilation,
                    PPL_IntLikeOrNone:$ins, PPL_FpInt:$scale,
                    PPL_FpInt:$rshift);
    let assemblyFormat = "operands attr-dict `:` type(operands)";
}
```

Figure 3

2) 根据 OP 是属于哪种 engine 选择继承对应的 Interface，例如 pool_avg 属于 tiu engine，则继承 TiuInterface。
3) 如果 OP 需要生成 tpu kernel 代码则添加 EmitInterface，后面加上支持此算子的芯片名称。
4) Tensor、dim 等参数定义为 PPL_MemRef 类型。

第11条 添加前端函数到 IR 的转换

1) 前端函数到 IR 的转换在 lib/Dialect/PplFe/Transforms/ToPPL.cpp 中，前端函数在这里是 pool_avg，PPL 中 OP 名字在这里是 AvgPool2DOp。

```
{
  //// pooling
  node.func_map["pool_max"] = createCommon<MaxPool2DOp>;
  node.func_map["fpool_max"] = createCommon<MaxPool2DOp>;
  node.func_map["pool_min"] = createCommon<MinPool2DOp>;
  node.func_map["fpool_min"] = createCommon<MinPool2DOp>;
  node.func_map["fpool_avg"] = createAvgPool2DOp;
  node.func_map["pool_avg"] = createAvgPool2DOp;
}
```

Figure 4

2) 根据指令所属 engine，在 init_ns_ppl_tiu、init_ns_ppl_dma 或 init_ns_ppl_sdma 函数中添加指令到 OP 的映射。
3) 在 func_map 中添加前端函数的名字。
4) 如果前端函数参数与 OP 定义参数一致，并且没有特殊处理，则可以使用 createCommon<OP>，否则需要添加一个自定义的转换函数。

第12条 添加 IR 到 tpu_kernel 代码的转换

1) IR 到 tpu_kernel 代码的转换在 lib/Dialect/Ppl/IR 目录下，根据芯片类型在对应的文件夹下添加 OP 的 emit 函数。

```cpp
#include "mlir/Support/IndentedOstream.h"
#include "ppl/Support/BmHelper.h"
#include "ppl/Support/CEmitter.h"

using namespace mlir;
liang.chen, 2 months ago | 4 authors (liang.chen and others)
namespace mlir {
liang.chen, 2 months ago | 4 authors (liang.chen and others)
namespace ppl {

LogicalResult AvgPool2DOp::emitBM1684x(void *emitter) {
  CEmitter *cEmitter = static_cast<CEmitter *>(emitter);
  raw_indented_ostream &os = cEmitter->ostream();
  auto block = getOperation()->getBlock();
  auto rst = getRst();
  auto in = getInput();
  auto kernel = getKernel();
  auto pad = getPad();
  auto stride = getStride();
  auto dilation = getDilation();
  auto fmt = getDataType(in);
  auto scale = getScale();
  auto rshift = getRshift();
  auto ins = getIns();
  auto rst_shape_name = cEmitter->getTensorShapeName(rst);
  auto rst_addr_name = cEmitter->getTensorAddrName(rst);
  auto in_shape_name = cEmitter->getTensorShapeName(in);
  auto in_addr_name = cEmitter->getTensorAddrName(in);
  std::string op_fmt;

  // check layout
  assertOpError(getTensorAlign(rst) == TPU_ALIGN, getOperation(),
                "Dst's layout must be TPU_ALIGN!!!\n");
  assertOpError(getTensorAlign(in) == TPU_ALIGN, getOperation(),
                "Src's layout must be TPU_ALIGN!!!\n");
```

Figure 5

第13条 正确性验证

```cpp
using namespace ppl;
__KERNEL__ void avg_pool_2d_bf16(bf16 *ptr_rst, bf16 *ptr_inp, int N, int C,
                                 const int kh, const int kw, const int stride_h,
                                 const int stride_w, const int pad_h_t, const int pad_h_b,
                                 const int pad_w_l, const int pad_w_r, const int H, const int W,
                                 const int OH, const int OW,
                                 const int block_c, const int block_oh) {

  const int dilation_h = 1;
  const int dilation_w = 1;

  int effective_kh = kh + (kh - 1) * (dilation_h - 1);
  int effective_kw = kw + (kw - 1) * (dilation_w - 1);
  int new_C = N * C;
  dim4 inp_shape = {1, new_C, H, W};
  dim4 rst_shape = {1, new_C, OH, OW};

  dim2 kernel = {kh, kw};

  dim2 stride = {stride_h, stride_w};
  dim2 dilation = {dilation_h, dilation_w};

  auto inp_gtensor = gtensor<bf16>(inp_shape, GLOBAL, ptr_inp);
  auto out_gtensor = gtensor<bf16>(rst_shape, GLOBAL, ptr_rst);

  int block_ih = (block_oh - 1) * stride_h + effective_kh;
  dim4 inp_local_mem_shape = {1, block_c, block_ih, W};
  dim4 out_local_mem_shape = {1, block_c, block_oh, OW};

  auto inp = tensor<bf16>(inp_local_mem_shape);
  auto res = tensor<bf16>(out_local_mem_shape);
```

Figure 6

1) 在 regression/unittest/pool/ 目录下（也可以是其他文件夹，可以找一个类似的算子程序进行修改），需要增加 .pl 程序验证算子添加是否存在语法错误，如 Figure 6。

2) 在同级目录下，需要增加 .py 程序，将 pl 的结果和 torch 的结果进行对比，验证添加的算子计算结果正确。

```
torch.manual_seed(0)
delta = 0.005
shape = (1, 8, 32, 16)
kernel_size = (2, 2)
stride = (1, 1)
padding = (1, 1)
dilation = (1, 1)
x = torch.rand(shape, device='cpu', dtype=torch.bfloat16)
output_torch = F.avg_pool2d(x, kernel_size=kernel_size, stride=stride, padding=padding)
output_ppl = avg_pool_(x, kernel_size, stride, padding, dilation)
print(output_torch)
print(output_ppl)
max_delta = torch.max(torch.abs(output_torch - output_ppl))
assert  max_delta < delta, \
    f"The maximum difference between torch and ppl avg_pool op is {delta}, but got {max_delta}"
print(
    f'The maximum difference between torch and ppl avg_pool op is '
    f'{max_delta}'
)

x = torch.rand(shape, device='cpu', dtype=torch.bfloat16)
output_torch = F.max_pool2d(x, kernel_size=kernel_size, stride=stride, padding=padding)
output_ppl = max_pool_(x, kernel_size, stride, padding, dilation)
print(output_torch)
print(output_ppl)
max_delta = torch.max(torch.abs(output_torch - output_ppl))
assert  max_delta < delta, \
    f"The maximum difference between torch and ppl max_pool op is {delta}, but got {max_delta}"
print(
    f'The maximum difference between torch and ppl max_pool op is '
    f'{max_delta}'
)
```

Figure 7

第六章 Python 指令添加

以 pool_avg 算子添加为例，介绍 python 指令添加的流程。

第14条 前端添加算子 api 函数

1) Python 前端函数定义文件在 PPL 工程的 python/ppl/language 下，主要包含以下几个头文件：

文件名	说明
tiu/tiu.py	TIU 操作相关函数
dma/dma.py	DMA 操作相关函数
sdma/sdma.py	SDMA 操作相关函数
hau/hau.py	HAU 操作相关函数
tiu、dma、sdma、hau 目录下的__init__.py	对用户暴露定义的 api

2) 根据你要添加的算子类型在 tiu.py、dma.py、sdma.py、或 hau.py 中添加对应的前端函数。以下是 pool_avg 函数定义示例：

```
@builtin
def pool_avg(output, pointer, kernel, padding, stride, dilation, scale, rshift=0, ins=None,
             _builder=None):
```

a) 整数数据类型 2D 均值池化,可自定义均值 scale 值,对结果做算术移位，结果有 saturation。

```
pool_avg(output, pointer, kernel, padding, stride, dilation, scale, rshift, ins)
```

b) 浮点数据类型 2D 均值池化，可自定义均值 scale 值来代替传统的 1 / (kernel->h * kernel->w)。

```
pool_avg(output, pointer, kernel, padding, stride, dilation, scale, ins)
```

c) 各个参数的含义与类型如下表所示：

参数名	类型	说明
output	ppl.language.tensor	output 张量
pointer	ppl.language.tensor	input 张量
kernel	dim2	kernel 大小
padding	dim4	padding 大小
stride	dim2	stride 大小
dilation	dim2	dilation 大小
scale	int or float	scale 值
rshift	int	右移位数，在浮点数据类型中不需要
ins	dim2	insert 大小

d) pool_avg 无返回值，使用示例如下：

```
scale = _to_tensor(scale, _builder)
rshift = _to_tensor(rshift, _builder)
return semantic.pool_avg(output, pointer, kernel, padding, stride, dilation, ins, scale, rshift,
                         _builder)
```

3) 该函数定义了一个使用 TIU 做 pool_avg 的函数，有 7 个必选参数：output, pointer, kernel, padding, stride, dilation, scale，2 个可选参数：rshift, ins。

4) output、pointer 这 2 个参数类型都是 Tensor，可选参数通过关键字参数进行传递，scale、rshift 是标量参数，在定义体中需要调用 _to_tensor() 将标量参数转为 tensor 类型， kernel、padding、stride、dilation、ins 参数属于 list 类型。

5) 为了减少接口个数及用户困扰,尽量通过关键字参数去复用同一个 api。Api 使用说明在函数定义体中实现，包括多种使用情况、参数说明、返回值、注意事项、使用示例。

6) 在对应目录下的 __init__.py 中声明下定义的 api，示例：

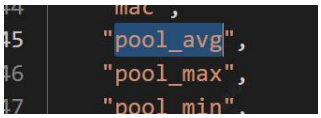

Figure 8

第15条 添加具体实现

1) 在 semantic.py 中添加具体的实现：

```
def pool_avg(output: pl.tensor,
             input: pl.tensor,
             kernel: List[int],
             padding: List[int],
             stride: List[int],
             dilation: List[int],
             ins: List[int],
             scale: pl.tensor,
             rshift: pl.tensor,
             builder: ir.builder) -> pl.tensor:
    kernel = _convert_to_ir_values(builder, kernel, require_i64=False)
    stride = _convert_to_ir_values(builder, stride, require_i64=False)
    padding = _convert_to_ir_values(builder, padding, require_i64=False)
    dilation = _convert_to_ir_values(builder, dilation, require_i64=False)
    if ins is None:
        ins = [builder.get_null_value(pl.int8.to_ir(builder))]
    else:
        ins = _convert_to_ir_values(builder, ins, require_i64=False)
    return pl.tensor(builder.create_pool_avg(output.handle, input.handle, kernel, padding,
                                             stride, dilation, ins, scale.handle,
                                             rshift.handle), pl.void)
```

2) 对 list 参数调用 _convert_to_ir_values 转为 std::vector<Value>。

3) 可选 list 参数传参为 None 时调用 [builder.get_null_value(pl.int8.to_ir(builder))] 转为 std::vector<Value>。

4) 对 tensor、标量参数，直接调用对应的 handle 传参就可以。

5) 当 tensor、标量参数传参为 None 时，可以在 tiu.py 或 semantic.py 中调用 _to_tensor 将 None 创建为 Tensor。

第16条 在 ppl.cc 中实现 create_pool_avg

1) 在 ppl.cc 中定义 create_pool_avg，它的作用是创建 IR，mlir 创建 IR 的通用方法。

```
.def("create_pool_avg",
```

```
    [](TritonOpBuilder &self, mlir::Value &output,
                       mlir::Value &input,
                       std::vector<mlir::Value> &kernel,
                       std::vector<mlir::Value> &padding,
                       std::vector<mlir::Value> &stride,
                       std::vector<mlir::Value> &dilation,
                       std::vector<mlir::Value> &ins,
                       mlir::Value &scale,
                       mlir::Value &rshift) -> void {
                     std::vector<mlir::NamedAttribute> attrs;
                     mlir::Type i32Type = self.getBuilder().getIntegerType(32);
                     attrs.emplace_back(self.getBuilder().getNamedAttr(
                           "struct", mlir::StringAttr::get(self.getBuilder().getContext(),
                                                           "dim2")));
                     auto kernel_ = self.create<mlir::ppl::ShapeOp>(
                           mlir::MemRefType::get({1, 2}, i32Type), kernel, attrs);
                     auto stride_ = self.create<mlir::ppl::ShapeOp>(
                           mlir::MemRefType::get({1, 2}, i32Type), stride, attrs);
                     auto dilation_ = self.create<mlir::ppl::ShapeOp>(
                           mlir::MemRefType::get({1, 2}, i32Type), dilation, attrs);
                     mlir::Value ins_;
                     if (ins.size() == 2)
                       ins_ = self.create<mlir::ppl::ShapeOp>(
                           mlir::MemRefType::get({1, 2}, i32Type), ins, attrs);
                     else
                       ins_ = self.create<mlir::ppl::NoneOp>(self.getBuilder().getNoneType());
                     std::vector<mlir::NamedAttribute>().swap(attrs);
                     attrs.emplace_back(self.getBuilder().getNamedAttr(
                           "struct", mlir::StringAttr::get(self.getBuilder().getContext(),
                                                           "padding_t")));
                     auto padding_ = self.create<mlir::ppl::ShapeOp>(
                           mlir::MemRefType::get({1, 4}, i32Type), padding, attrs);
                     self.create<mlir::ppl::AvgPool2DOp>(output, input, kernel_, padding_, stride_,
                                                         dilation_, ins_, scale, rshift);
                     return;
                                                           })
```

第17条 在 example/python 目录下添加对应的测试例

```
import torch
import torch.nn.functional as F
import ppl
import ppl.language as pl
```

```
@ppl.jit
def avg_pool(
        x_ptr,
        output_ptr,
        n:pl.constexpr,
        c:pl.constexpr,
        h:pl.constexpr,
        w:pl.constexpr,
        oh:pl.constexpr,
        ow:pl.constexpr,
        kh:pl.constexpr,
        kw:pl.constexpr,
        stride_h:pl.constexpr,
        stride_w:pl.constexpr,
        pad_h_up:pl.constexpr,
        pad_h_down:pl.constexpr,
        pad_w_l:pl.constexpr,
        pad_w_r:pl.constexpr,
        dilation_h:pl.constexpr,
        dilation_w:pl.constexpr,
        scale:pl.constexpr
):
        pid = pl.get_core_index()
        in_shape = [n, c, h, w]
        out_shape = [n, c, oh, ow]
        offset = [0, 0, 0, 0]
        kernel = [kh, kw]
        padding = [pad_h_up, pad_h_down, pad_w_l, pad_w_r]
        stride = [stride_h, stride_w]
        dilation    = [dilation_h, dilation_w]
        x_global = pl.gtensor(in_shape, pl.GLOBAL, x_ptr)
        o_global = pl.gtensor(out_shape, pl.GLOBAL, output_ptr)
        output = pl.make_tensor(out_shape, x_ptr.dtype)
        x = pl.dma.load(x_global[:,:,:,:])
        pl.tiu.pool_avg(output, x, kernel, padding, \
                        stride, dilation, scale)
        pl.dma.store(o_global[:,:,:,:], output)

def avg_pool_(x: torch.Tensor, kernel_size, stride, padding, dilation):
        n, c, h, w = x.shape
        oh = (h + padding[0] * 2 - kernel_size[0]) // stride[0] + 1
        ow = (w + padding[1] * 2 - kernel_size[1]) // stride[1] + 1
        scale = 1 / (kernel_size[0] * kernel_size[1])
        output = torch.empty((n, c, oh, ow), device=x.device, dtype=x.dtype)
```

```
    avg_pool[(1,)](x, output, n, c, h, w, oh, ow, kernel_size[0], kernel_size[1], stride[0],
                   stride[1], padding[0], padding[0], padding[1], padding[1], dilation[0],
                   dilation[1], scale)
    return output

torch.manual_seed(0)
delta = 0.005
shape = (1, 8, 32, 16)
kernel_size = (2, 2)
stride = (1, 1)
padding = (1, 1)
dilation = (1, 1)
x = torch.rand(shape, device='cpu', dtype=torch.bfloat16)
output_torch = F.avg_pool2d(x, kernel_size=kernel_size, stride=stride, padding=padding)
output_ppl = avg_pool_(x, kernel_size, stride, padding, dilation)
max_delta = torch.max(torch.abs(output_torch - output_ppl))
assert  max_delta < delta, f"The maximum difference between torch and ppl avg_pool op is
{delta}, but got {max_delta}"
print(
    f'The maximum difference between torch and ppl avg_pool op is '
    f'{max_delta}'
)
```

1） 测试例中 PPL 运算结果要和 torch cpu 计算结果进行比较。

2） 测试例要完备，如果对应的 api 支持多种运算，都需要有对应的测试例。

第18条 测试例加入回归

1） 在 python/tool/example.py 中加入测试例，已经支持的芯片型号。

```
python_list = {
    # (filename,                    bm1684x, bm1688, bm1690, sg2380)
    "examples/python":
      [("01-element-wise.py",          Y,       Y,       Y,       N),
       ("01-element-wise-bm1684x.py",  Y,       N,       N,       N),
       ("02-avg-max-pool.py",          Y,       N,       N,       N),
```

Figure 9

第七章 Pass 添加

第19条 定位优化 pass 的添加位置

1） 在 PPL 编译器中，执行命令 ppl_compile --src *.pl --chip * 时，编译过程从 .pl 文件到生成目标 C/C++ 代码的主要优化路径参考第 7 条。

2） 以下是各优化步骤及其对应的添加位置的总结：

步骤	代码入口	Pass 实现位置
runFrontendPass	/ppl/bin/ppl_compile.cpp	/ppl/lib/Dialect/PplFe/Transforms
runTopplPass	/ppl/bin/ppl_compile.cpp	/ppl/lib/Dialect/PplFe/Transforms
run_opt1	/ppl/bin/ppl_compile.cpp	/ppl/lib/Dialect/Ppl/Transforms
run_opt2	/ppl/bin/ppl_compile.cpp	/ppl/lib/Dialect/Ppl/Transforms
run_final	/ppl/bin/ppl_compile.cpp	/ppl/lib/Dialect/Ppl/Transforms
runTranslate	/ppl/lib/Target/Common/ConvertToC.cpp	

第20条 Pattern Match 机制概述

1) 在相应的 Pass 实现位置(例如 /ppl/lib/Dialect/Ppl/Transforms 或 /ppl/lib/Dialect/PplFe/Transforms) 中，创建一个新的 OpRewritePattern。
2) 在 MLIR 中，Pattern Match 是一种用于对操作进行匹配和重写的机制，主要用于优化和转换 IR。以下是其工作流程的核心步骤：
   a) 定义 Rewrite Pattern：使用 mlir::OpRewritePattern 或其他相关基类定义模式匹配和重写逻辑。matchAndRewrite() 方法是核心，负责匹配操作并执行重写。
   b) 注册 Pattern：将定义的 Pattern 添加到 RewritePatternSet 中。RewritePatternSet 是一个容器，用于存储多个 Rewrite Pattern。
   c) 应用Pattern：使用 applyPatternsAndFoldGreedily() 方法，将 Pattern 应用到目标操作或模块中。该方法会在整个模块中逐步匹配和重写操作，直到无法再进行优化。

第21条 Pattern Match 的添加与执行

1) 定义 Rewrite Pattern
   a) 通过继承 mlir::OpRewritePattern 类，定义一个 Rewrite Pattern。以下是通用的 Rewrite Pattern 定义示例：

```
template <typename TargetOp>
class GeneralRewritePattern : public mlir::OpRewritePattern<TargetOp> {
public:
  GeneralRewritePattern(mlir::MLIRContext *context)
      : mlir::OpRewritePattern<TargetOp>(context) {}

  mlir::LogicalResult
  matchAndRewrite(TargetOp targetOp, mlir::PatternRewriter &rewriter) const override {
    // 1. 检查操作类型，确认是否需要重写
    if (!isValidOperation(targetOp)) {
      return mlir::failure();
    }
```

```
    // 2. 构造新的操作
    auto newOp = rewriter.create<NewOperationType>(targetOp.getLoc(), ...);

    // 3. 替换旧操作
    rewriter.replaceOp(targetOp, newOp->getResults());
    return mlir::success();
  }

private:
  bool isValidOperation(TargetOp op) const {
    // 检查目标操作是否满足特定条件
    return true; // 示例：始终返回 true
  }
};
```

b) TargetOp 是目标操作的类型，例如 MyOp。

c) matchAndRewrite() 是核心方法，用于描述匹配逻辑和重写规则。

d) rewriter 是 PatternRewriter 的实例，用于插入新操作和替换旧操作。

2) 注册 Pattern

a) 在 Pass 的实现中，将定义的 Pattern 注册到 RewritePatternSet 中。例如：

```
void runOnOperation() override {
  mlir::MLIRContext *context = &getContext();
  mlir::RewritePatternSet patterns(context);
  mlir::ModuleOp module = getOperation();

  // 注册通用的 Pattern
  patterns.add<GeneralRewritePattern<MyTargetOp>>(context);
  patterns.add<GeneralRewritePattern<AnotherTargetOp>>(context);

  // 应用 Pattern
  if (applyPatternsAndFoldGreedily(module, std::move(patterns)).failed()) {
    signalPassFailure();
  }
}
```

b) 使用 patterns.add<PatternType>() 将 Pattern 注册到 RewritePatternSet。

c) applyPatternsAndFoldGreedily() 是核心方法，用于在模块中应用所有注册的 Pattern。

3) 应用 Pattern : applyPatternsAndFoldGreedily() 方法会在模块中递归地匹配和重写操作，直到所有 Pattern 都无法匹配为止。以下是其执行流程：

a) 遍历模块中的所有操作。

b) 对每个操作，尝试匹配已注册的 Pattern。

c) 如果匹配成功，则执行 matchAndRewrite() 方法中的重写逻辑。

d) 替换旧操作，并继续匹配其他操作。

e) 如果某些操作在重写后可以再次匹配 Pattern，则重复上述过程。

第22条 Pattern Match 的优点

1) 模块化：每个 Pattern 只关注特定的操作，逻辑清晰且易于维护。

2) 自动化： applyPatternsAndFoldGreedily() 自动处理操作的匹配和重写，开发者无需手动遍历 IR。

3) 可扩展性：可以轻松添加新的 Pattern 以处理更多操作。

第23条 添加优化 Pass

1) 定义 Pass 类

a) 在相应的 Pass 实现位置(例如/ppl/lib/Dialect/Ppl/Transforms 或 /ppl/lib/Dialect/PplFe/Transforms)中，创建一个新的 Pass 类。

b) 继承自 mlir::Pass 或者 mlir::OpRewritePattern。

c) 实现 runOnOperation() 或 matchAndRewrite() 方法。

d) 示例代码：

```
class MyOptimizationPass : public PassWrapper<MyOptimizationPass, OperationPass<ModuleOp>> {
public:
  void runOnOperation() override {
    // 实现优化逻辑
  }
};
```

2) 注册 Pass

a) 在 /ppl/include/ppl/Dialect/Ppl/Transforms/Passes.h 或 /ppl/include/ppl/Dialect/PplFe/Transforms/Passes.h 中声明 Pass 的创建函数：

```
std::unique_ptr<Pass> createMyOptimizationPass();
```

b) 在 /ppl/include/ppl/Dialect/Ppl/Transforms/Passes.td 中定义 Pass 的元信息：

```
def MyOptimizationPass : Pass<"my-optimization", "mlir::ModuleOp"> {
  let summary = "My custom optimization pass";
  let constructor = "mlir::ppl::createMyOptimizationPass()";
}
```

3) 实现 Pass 创建函数

a) 在 Pass 实现文件中，定义 Pass 的具体逻辑，并实现创建函数：

```
std::unique_ptr<mlir::Pass> createMyOptimizationPass() {
  return std::make_unique<MyOptimizationPass>();
}
```

4) 将 Pass 添加到优化 Pipeline

a) 在 /ppl/bin/ppl_compile.cpp 中找到对应的优化步骤（如 run_opt1()、run_opt2() 或 run_final()）。

b) 使用 mlir::PassManager 添加新定义的 Pass：

```
mlir::PassManager _pm(&context);
_pm.addPass(mlir::ppl::createMyOptimizationPass());
```

5) 测试 Pass

a) 为新添加的 Pass 编写测试用例。

b) 在 /ppl/test 目录下添加相关测试文件。

c) 使用 .pl 文件作为输入，通过 ppl_compile.py 命令验证优化效果。

第24条 构建测试样例

1) 创建测试输入

a) 在 /ppl/test 目录下创建一个名为 test.pl 的文件作为测试输入：

```
#include "ppl.h"   // PPL 代码依赖的头文件
using namespace ppl;
__KERNEL__ void function(fp32 *a, fp32 *b, int C) {
    // add code here
}
```

```
__TEST__ void add() {
  int C = ...;
  dim4 shape = {..., ..., ..., ...};
  fp16 *a = rand<fp32>(&shape);
  fp16 *b = rand<fp32>(&shape, ..., ...);
  function(a, b, C);
}
```

2) 运行编译器

a) 使用 ppl_compile 工具编译 pl 文件，生成目标代码或 IR：

```
ppl_compile --src test.pl --chip bm1690 --gen_ref
```

b) 运行后，test_test/ 是生成的可执行文件以及IR、C/C++文件的目录，可以直接运行以验证功能。

3) 验证优化效果

a) 对比 IR 文件： 对比优化前后的IR文件，验证优化逻辑是否符合预期： 如果优化 Pass 工作正常，应能看到优化后的IR中移除了冗余操作或进行了性能优化。

b) 运行优化后的代码： 如果生成了优化后的代码（如 test_case），可以直接运行 ./test_case 来验证功能。

c) 检查运行结果是否正确，并与优化前的结果进行对比。

第八章 芯片添加

以 Mars3 芯片的添加为例，介绍 PPL 工程的芯片添加流程。

第25条 添加芯片信息

1) 于 lib/Support/Arch.cpp 文件添加如下芯片的相关信息。

```
87+ class ArchMARS3 : public Arch {
88+ protected:
89+   ArchMARS3() {
90+     npu_num = 8;
91+     eu_bytes = 16;
92+     lmem_bytes = 1 << 16; // 512KB
93+     lmem_banks = 16;
94+     lmem_bank_bytes = lmem_bytes / lmem_banks;
95+     l2_size = 0;
96+     l2_start_addr = 0;
97+     core_num = 1;
98+   }
99+   friend Arch* archCreator(Chip chip);
100+  friend class Singleton<ArchMARS3>;
101+ };
102
103  Arch* archCreator(Chip chip) {
104    switch (chip) {
105    case mlir::ppl::Chip::BM1684X:
106      return (Arch *)(Singleton<ArchBm1684x>::GetInstance());
107    case mlir::ppl::Chip::SG2260:
108      return (Arch *)(Singleton<ArchSg2260>::GetInstance());
109    case mlir::ppl::Chip::BM1688:
110      return (Arch *)(Singleton<ArchBm1688>::GetInstance());
111    case mlir::ppl::Chip::SG2380:
112      return (Arch *)(Singleton<ArchSg2380>::GetInstance());
113    case mlir::ppl::Chip::SG2262:
114      return (Arch *)(Singleton<ArchSg2262>::GetInstance());
115+   case mlir::ppl::Chip::MARS3:
116+     return (Arch *)(Singleton<ArchMARS3>::GetInstance());
117    default:
118      llvm_unreachable("arch not supported\n");
119    }
```

Figure 10

第26条 添加芯片 emit 支持

1) PPL 工程的 emit 是以 OP 为单位，如果 OP 需要生成 tpu-kernel 代码则添加 EmitInterface，后面加上支持此算子的芯片名称。
2) 所以先于 include/ppl/Dialect/Ppl/IR/PplOpInterfaces.td 文件中的 EmitInterface 下添加 Mars3 芯片的 emit 函数。

```
102+      InterfaceMethod<
103+        /*desc=*/[{}],
104+        /*retType=*/"::mlir::LogicalResult",
105+        /*methodName=*/"emitMARS3",
106+        /*args=*/(ins "void *":$emitter),
107+        [{}],
108+        [{
109+          return $_op.emitBM1684x(emitter);
110+        }]
111+      >,
```

Figure 11

3) 由于各个芯片的 tpu-kernel 代码差异较小，所以大部分 OP 可以直接复用 BM1684x 芯片的代码生成逻辑。
4) 当芯片的 tpu-kernel 代码之间存在差异时，于 lib/Dialect/Ppl/IR/目录下以芯片名称新建一个目录，再到新目录中添加当前芯片 OP 的代码生成逻辑。
5) 例如，Mars3 芯片的 sqrt 算子与 BM1684x 芯片存在差异，于

include/ppl/Dialect/Ppl/IR/PplOpInterfaces.td 文件中，sqrt OP 定义时添加 Mars3 芯片的 emit 函数重写机制。

```
def PPL_Fp32SqrtOp: PPL_Op<"fp32_sqrt",
                           [MemoryEffects<[MemWrite]>,
                           DeclareOpInterfaceMethods<TiuInterface>,
                           DeclareOpInterfaceMethods<MemInterface>,
                           DeclareOpInterfaceMethods<EmitInterface, ["emitSG2260", "emitSG2262", "emitBM1688", "emitMARS3"]>,
                           DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
    let summary = "fp32 sqrt";
    let arguments = (ins PPL_MemRef:$dst, PPL_MemRef:$src, PPL_Int:$num_iter);
    let assemblyFormat = "operands attr-dict `:` type(operands)";
}
```

Figure 12

6) 于 lib/Dialect/Ppl/IR/MARS3/ 目录下新建 Sqrt.cpp，重写 emitMARS3 函数。

```
LogicalResult Fp32SqrtOp::emitMARS3(void *emitter) {
  CEmitter *cEmitter = static_cast<CEmitter *>(emitter);
  raw_indented_ostream &os = cEmitter->ostream();
  auto dst = getDst();
  auto src = getSrc();
  auto num_iter = getNumIter();

  auto dst_fmt = getDataType(dst);
  auto src_fmt = getDataType(src);
  auto dst_addr_name = cEmitter->getTensorAddrName(dst);
  auto src_addr_name = cEmitter->getTensorAddrName(src);
  auto shape_name = cEmitter->getTensorShapeName(src);
  auto num_iter_name = cEmitter->getOrCreateName(num_iter);
```

Figure 13

Figure 14

7) 在 emit 阶段，还有一些 runtime 接口、CMakelists、函数调用等细节的修改（如 Figure 14），请参考如下提交进行修改：

Commit Id ： a56cf1949083e579e0d53b316a72388572173758
https://gerrit-ai.sophgo.vip:8443/#/c/133051/

第27条 添加芯片 runtime 支持

1) 需要进入 TPU1686 和 libsophon 工程，编译芯片的 runtime 库，存放

到 PPL 工程的 runtime/ 目录下。

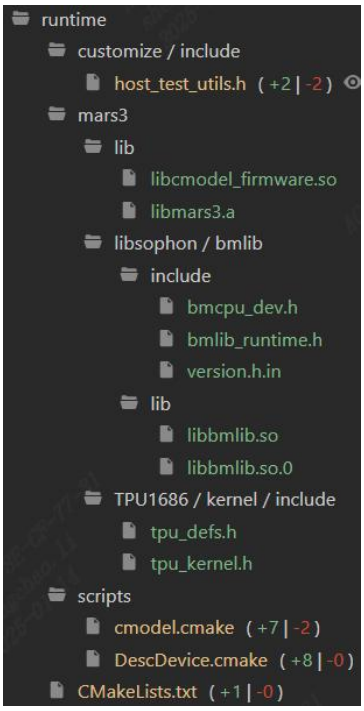

Figure 15

2) 在 runtime/scripts 目录中添加 cmodel.cmake 和 DescDevice.cmake 文件对芯片 runtime 的包含路径。

第28条 正确性检测

1) 在 python/tool/ppl_compile.py 编译脚本中添加芯片编译运行支持。

```
elif args.gen_ref:
    ret = tool.gen_all_file()
    if ret != 0:
        print("[!Error]: gen all file failed")
        return ret
    if args.chip == "bm1684x" or args.chip == "bm1690" \
        or args.chip == "bm1688" or args.chip == "sg2262" or args.chip == "mars3":
        ret = tool.build_code()
        if ret != 0:
            print("[!Error]: build code failed")
            return ret
        ret = tool.validate()
        if ret != 0:
            print("[!Error]: validate failed")
            return ret
```

```
parser.add_argument("--chip",
                    required=True,
                    type=str.lower,
                    choices=['bm1684x', 'bm1690', 'bm1688', 'sg2380', 'sg2262', 'mars3'],
                    help="chip platform name")
```

Figure 16

```
def validate(self):
    os.environ["PPL_DUMP_IR"] = "1"
    os.environ["PPL_CACHE_PATH"] = os.path.join(self.target_path, "cache")
    if self.desc:
        os.environ["PPL_SRC_PATH"] = os.environ["PPL_PROJECT_ROOT"]
        os.environ["PPL_WORK_PATH"] = self.target_path
    if self.chip == "bm1684x" or self.chip == "bm1690" or self.chip == "bm1688" or self.chip == "sg2262" or self.chip == "mars3":
        if self.chip == "bm1684x":
            runtime_lib = "libsophon/bmlib"
        elif self.chip == "bm1688":
            runtime_lib = "libsophon/bmlib"
        elif self.chip == "mars3":
            runtime_lib = "libsophon/bmlib"
```

```
def profiling_cmodel(self, autotune):
    if self.chip == "bm1684x" or self.chip == "bm1690" or self.chip == "bm1688" or self.chip == "sg2262" or self.chip == "mars3":
        os.environ["FILE_DUMP_CMD"] = self.out_name
        if self.chip == "bm1684x":
            runtime_lib = "libsophon/bmlib"
        elif self.chip == "bm1688":
            runtime_lib = "libsophon/bmlib"
        elif self.chip == "mars3":
            runtime_lib = "libsophon/bmlib"
```

```
def profiling_pcie(self):
    if self.chip == "bm1684x" or self.chip == "bm1690" or self.chip == "bm1688" or self.chip == "sg2262" or self.chip == "mars3":
        os.environ["BMLIB_ENABLE_ALL_PROFILE"] = "1"
        os.environ["PPL_DATA_PATH"] = self.data_path
        if self.chip == "bm1690" or self.chip == "sg2262":
            os.environ["PPL_KERNEL_PATH"] = os.path.join(
                self.target_path, "lib/libcmodel.so")
```

Figure 17

2) 修改已存在的 .pl 和 .py 文件或者重新编写 .pl 和 .py 文件，使用 ppl_compile.py 进行测试。

```
ppl_compile.py --src regression/unittest/func/sqrt_mars3_bf16.pl --chip mars3 --gen_ref
python regression/unittest/func/sqrt.py
```

第29条 添加线上回归检测

1) 于 python/tool/regression.py 文件添加芯片回归支持。

```
22      def __init__(self, top_dir, chips, file_list, mode, is_full, time_out):
23          self.result_message = ""
24          self.top_dir = top_dir
25          self.file_list = file_list
26          self.chips = chips
27          self.test_failed_list = []
28          self.vali_failed_list = []
29          self.time_out_list = []
30          self.dubious_pl_list = {}
31          self.file_not_found_list = {}
32          self.chip_index_map = {
33              'bm1684x': 1,
34              'bm1688': 2,
35              'bm1690': 3,
36              'sg2380': 4,
37              'sg2262': 5,
38+             'mars3': 6,
39          }
```

Figure 18

2) 于 python/tool/example.py 文件添加芯片回归时需要执行的示例。其中，full_list 中的 full 用于日常回归测试；base 用于提交代码回归测试。

```
Y, N = True, False

full_list = {
    # (filename,                                  bm1684x, bm1688, bm1690, sg2380, sg2262, masr3)
    #                                             [base, full]
    "regression/unittest/arith":
      [("arith_int.pl",                           Y,       Y,      Y,      N,      N,      N),
       ("add_pipeline_bm1688.pl",                 N,       N,      N,      N,      Y,      N),],
    "regression/unittest/arith_int":
      [("c_sub_int.pl",                           N,       N,      N,      N,      Y,      N),],
    "regression/unittest/attention":
      [("rotary_embedding_static.pl",             N,       N,      N,      N,      N,      N),
       ("mlp_left_trans_multicore.pl",            N,       N,      Y,      N,      N,      N)],
    "regression/unittest/cmp":
      [("greater_fp16.pl",                        Y,       N,      N,      N,      Y,      N),
       ("equal_int16.pl",                         Y,       N,      N,      N,      N,      N),],
    "regression/unittest/conv":
```

Figure 19

3） 于本地环境运行如下命令，进行回归测试。

```
regression.py --reg_mode full
```

第九章 checklist

Cat.	编号	Check 项目	Check 方法与参考值	Check 结果	PR & 时间
C++ 指令添加	1.	添加到的头文件是否正确	检查指令所属 engine 与头文件对应 engine 是否一致。 参考值：是。		
	2.	检查函数参数是否正确	对照 tpu kernel 文档核对参数类型。 参考值：是。		
	3.	检查OP参数是否正确	对照前端指令函数参数核对 OP 参数。 参考值：是。		
	4.	检查生成的 tpu kernel 代码是否正确	编写指令测试代码，检查结果是否正确。 参考值：是。		
	5.	检查添加的	编写 pl 和 py		

		算子逻辑和计算结果是否正确	测试代码，检查结果是否正确。 参考值：是		
Python 指令添加	6.	添加到的文件是否正确	检查指令所属 engine 与文件对应 engine 是否一致。 参考值：是。		
	7.	检查函数参数是否正确	对照 tpu kernel 文档核对参数类型。 参考值：是。		
	8.	检查函数参数是否正确	对照前端指令函数参数核对参数。 参考值：是。		
	9.	检查添加的 api 是否正确	编写指令测试代码，检查结果是否正确。 参考值：是。		
Pass 添加	10.	优化路径是否清晰	是否完整了解了从 runFrontendPass() 到 convertToC() 的优化路径及每个步骤的输入、输出和作用。 参考值：是。		
	11.	Pass 实现位置是否准确	是否明确了每个 Pass 的代码入口和实现位置（如 /ppl/bin/ppl_compile.cpp 和 /ppl/lib/Dialect/Ppl/Transf		

			orms）。 参考值：是。		
	12.	Pattern 定义是否规范	是否基于 mlir::OpRewritePattern 定义了 matchAndRewrite() 方法，且逻辑清晰、符合需求。 参考值：是。		
	13.	Pattern 注册是否正确	是否将定义的 Pattern 注册到 RewritePatternSet 中，并在 Pass 中正确应用。 参考值：是。		
	14.	Pattern 应用是否合理	是否使用了 applyPatternsAndFoldGreedily() 方法递归匹配和重写操作，优化结果是否符合预期。 参考值：是。		
	15.	Pass 定义是否规范	是否继承自 mlir::Pass 或 mlir::OpRewritePattern，并实现了 runOnOperation() 或 matchAndRewrite() 方法。 参考值：是。		
	16.	Pass 注册是否正确	是否在 Passes.h 和 Passes.td 中正确声明和		

			定义了 Pass 信息。 参考值：是。		
	17.	Pass 是否添加到 Pipeline 中	是否在 /ppl/bin/ppl_compile.cpp 中将 Pass 添加到优化 Pipeline 中（如 run_opt1() 或 run_final()）。 参考值：是。		
	18.	测试输入是否合理	是否在 /ppl/test 目录或者其它目录下创建了 .pl 文件，并包含了覆盖优化逻辑的测试用例。 参考值：是。		
	19.	测试输出是否符合预期	是否对比了优化前后的 IR 文件，验证优化逻辑是否符合预期。 参考值：是。		
	20.	优化后的代码是否正确运行	是否运行了优化后的代码，并验证结果与优化前一致或性能得到提升。 参考值：是。		
芯片添加	21.	芯片信息是否正确	是否对照芯片文档确定芯片信息的正确性。 参考值：是。		
	22.	EmitInterface 是否添加	是否存在定义默认 emit		

		正确	函数。 参考值：是。		
	23.	是否可以复用 BM1684x 芯片的 emit 函数	是否对照 tpu-kernel 文档和 TPU1686 工程，确定指令是差异。 参考值：是。		
	24.	是否正确编译芯片的 runtime 库	是否按照 libsophon 和 TPU1686 工程的 Readme 文档正确编译 runtime 库。 参考值：是。		
	25.	是否完成正确性检测	是否使用 ppl_compile.py 工具检测芯片添加的正确性。 参考值：是。		
	26.	是否完成本地回归检测	是否使用 regression.py 脚本在本地完成回归检测。 参考值：是。		