AutoKernel

OPEN AI LAB

2021 年 09 月 01 日

简介

CHAPTER 1

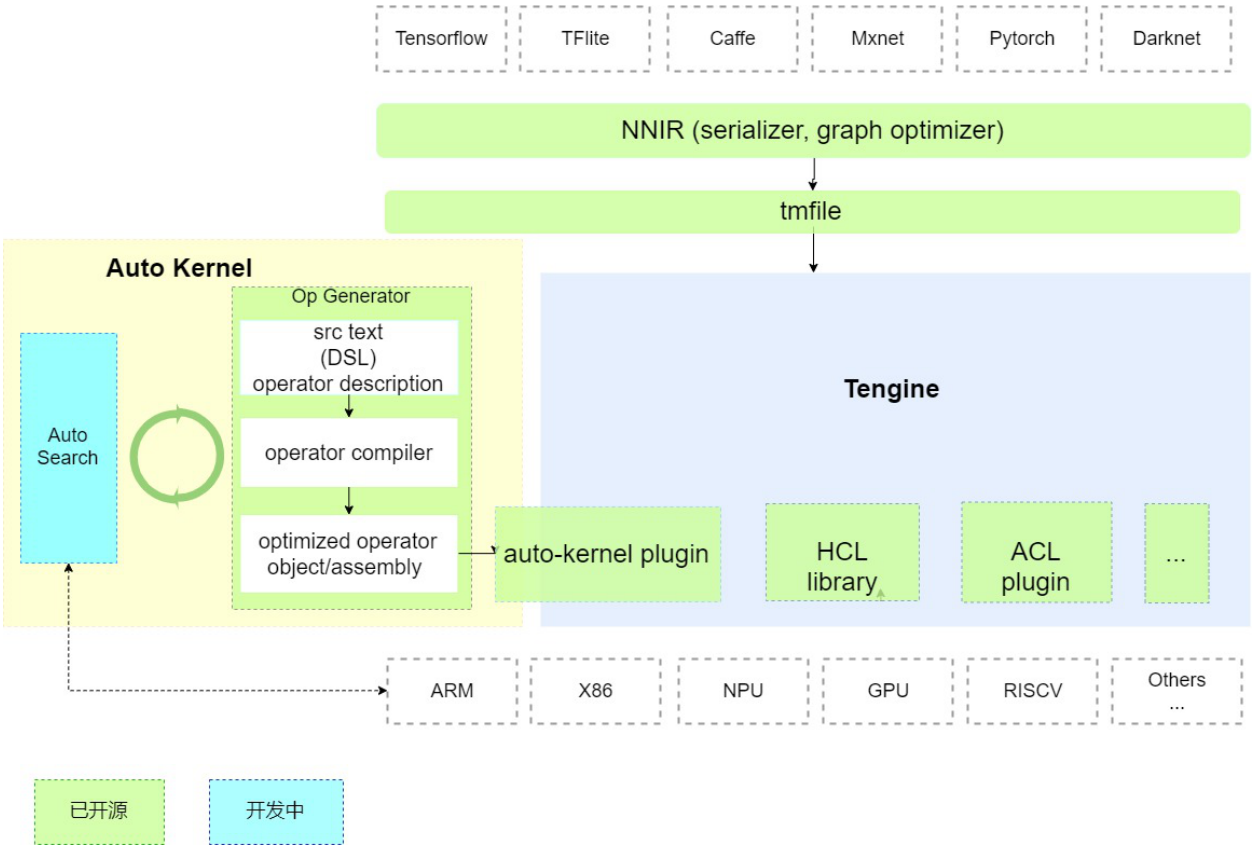
AutoKernel

1.1 简介

随着人工智能的普及，深度学习网络的不断涌现，为了让各硬件 (CPU, GPU, NPU, ⋯) 能够支持深度学习应用，各硬件芯片需要软件库去支持高性能的深度学习张量运算。目前，这些高性能计算库主要由资深 HPC 工程师 (高性能计算优化工程师）进行开发，为了加快开发进程，缩短深度学习应用落地周期，自动化算子优化是一个趋势。

AutoKernel 是由 OPEN AI LAB 提出的高性能算子自动优化工具，可以自动优化调度策略、生成底层优化代码，大幅减少各硬件芯片算子开发成本，提升算子优化效率，让工程师更快实现深度学习算法在各硬件芯片上的高性能部署。

1.2 AutoKernel 架构



AutoKernel 分为三个模块：

- 算子生成器:

  该模块使用了开源项目Halide；Halide 是业界广泛使用的自动代码生成项目，它首次提出将计算和调度分离。该模块的输入是和硬件无关的算子计算描述，输出是相应后端的优化汇编代码/目标文件；

- 自动搜索模块：

  该模块可以通过最优化算法/搜索算法/机器学习/强化学习搜索出相应后端的最优算子的调度策略参数（该模块仍在开发中）；

- 算子部署插件（AutoKernel Plugin）：

  Tengine是 OPEN AILAB 开源的深度学习推理框架，实现了 AI 算法在不同硬件的快速高效部署。该模块实现了将自动生成的优化算子代码以 plugin 的形式一键集成到Tengine中，实现自动优化算子的一键部署；

1.3 后端支持

1.4 Supported backends

以下后端 target 已测试:

- x86-64-linux
- x86-64-linux-opencl
- x86-64-linux-cuda
- arm-64-linux
- arm-64-linux-opencl

更多 target archs/features:

- arch:arm, hexagon, mips, powerpc, riscv, wasm, x86.
- bits：32, 64
- os：android, ios, linux, windows…
- features: avx, avx2, avx512, cl_half, cuda, opencl…

CHAPTER 2

源码安装

1. 编译安装 Halide

```
git clone https://github.com/halide/Halide
cd Halide && mkdir build && cd build
export HALIDE_DIR=/path/halide-install # 安装路径，根据实际修改
cmake .. -DTARGET_WEBASSEMBLY=OFF -DCMAKE_INSTALL_PREFIX=${HALIDE_DIR}
make -j `nproc` && make install # 编译安装
```

2. 编译 Tengine

```
git clone https://github.com/OAID/Tengine.git
cd Tengine && mkdir build && cd build
export TENGINE_DIR=/path/tengine-install # Tengine 安装路径
cmake .. -DCMAKE_INSTALL_PREFIX=${TENGINE_DIR}
make -j `nproc` && make install # 编译安装
```

3. 编译 AutoKernel

```
git clone https://github.com/OAID/AutoKernel.git
cd         AutoKernel/autokernel_plugin
find . -name "*.sh" | xargs chmod +x  #给 sh 脚本添加可执行权限
```

修改 ./scripts/generate.sh 中 halide 库的安装路径

```
# 将第一行中的 HALIDE_DIR 路径修改成第一步中的安装路径
export HALIDE_DIR=/path/halide-install
```

修改 Tengine 安装路径, 打开 autokernel_plugin/CMakeLists.txt 中 TENGINE_ROOT

```
set(TENGINE_ROOT /path/Tengine) # 修改为 Tengine 项目所在目录
set(TENGINE_DIR /path/tengine-install) # 修改为第二步中的安装路径
```

编译 AutoKernel

```
./scripts/generate.sh  #自动生成算子汇编文件
mkdir build && cd build
cmake .. && make -j `nproc`
```

将 libautokernel.so 所在的路径加入 LD_LIBRARY_PATH，在 AutoKernel/autokernel_plugin/build/ 目录下执行

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`/src/
```

在 AutoKernel/autokernel_plugin 目录下执行测试程序

```
./build/tests/tm_classification
```

运行结果如下

```
start to run register cpu allocator

......

[INFO]: using halide maxpooling....
[INFO]: using halide maxpooling....
[INFO]: using halide maxpooling....
current 53.934 ms

Model name : squeezenet
tengine model file : models/squeezenet.tmfile
label file : models/synset_words.txt
image file : images/cat.jpg
img_h, imag_w, scale, mean[3] : 227 227 1 104.007 116.669 122.679

Repeat 1 times, avg time per run is 53.934 ms
max time is 53.934 ms, min time is 53.934 ms
--------------------------------------
0.2732 - "n02123045 tabby, tabby cat"
0.2676 - "n02123159 tiger cat"
```

(下页继续)

(续上页)

```
0.1810 - "n02119789 kit fox, Vulpes macrotis"
0.0818 - "n02124075 Egyptian cat"
0.0724 - "n02085620 Chihuahua"
--------------------------------------
ALL TEST DONE
```

CHAPTER 3

基于 Docker

AutoKernel 提供了 docker 镜像，镜像内安装了 Halide 和 Tengine, 方便开发者直接使用:

3.1 Docker Image:

- cpu

```
docker pull openailab/autokernel
```

- cuda:

```
nvidia-docker pull openailab/autokernel:cuda
```

  [NOTE]: 使用 cuda 镜像需要用 nvidia-docker, 安装指南见 nvidia-docker install-guide.

- opencl:

```
docker pull openailab/autokernel:opencl
```

3.2 Dockerfile

具体的 Dockerfile 见

- Dockerfile.cpu
- Dockerfile.cuda
- Dockerfile.opencl

3.3 AutoKernel 安装指引

1. 拉取镜像 (可能需要一段时间，请耐心等待，根据网速，可能需要 10-20mins)

```
docker pull openailab/autokernel
```

2. 创建容器，进入开发环境

```
docker run -ti openailab/autokernel /bin/bash
```

3. docker 里面已经安装好 Halide, Tengine

```
/workspace/Halide         # Halide
/workspace/Tengine   # Tengine
```

4. AutoKernel

```
git clone https://github.com/OAID/AutoKernel.git
```

给 sh 脚本添加可执行权限, 自动生成算子汇编文件

```
cd          AutoKernel/autokernel_plugin
find . -name "*.sh" | xargs chmod +x
./scripts/generate.sh
```

编译

```
mkdir build && cd build
cmake .. && make -j `nproc`
```

执行测试程序

```
cd AutoKernel/autokernel_plugin
./build/tests/tm_classification -n squeezenet
```

3.4 Halide in Docker

Autokernel docker 里已安装好 Halide，并且配置好了 Python 的 API。

Halide 相关的文件都在/workspace/Halide/文件夹下，Halide 的安装文件都在/workspace/Halide/halide-build 文件夹下。

```
cd /workspace/Halide/halide-build
```

- Halide 相关头文件在/workspace/Halide/halide-build/include

```
root@bd3faab0f079:/workspace/Halide/halide-build/include# ls

Halide.h                        HalideRuntimeHexagonDma.h
HalideBuffer.h                  HalideRuntimeHexagonHost.h
HalidePyTorchCudaHelpers.h      HalideRuntimeMetal.h
HalidePyTorchHelpers.h          HalideRuntimeOpenCL.h
HalideRuntime.h                 HalideRuntimeOpenGL.h
HalideRuntimeCuda.h             HalideRuntimeOpenGLCompute.h
HalideRuntimeD3D12Compute.h     HalideRuntimeQurt.h
```

- 编译好的 Halide 库在/workspace/Halide/halide-build/src 目录下, 可以看到 libHalide.so

```
root@bd3faab0f079:/workspace/Halide/halide-build/src# ls
CMakeFiles            autoschedulers       libHalide.so.10
CTestTestfile.cmake   cmake_install.cmake  libHalide.so.10.0.0
Makefile              libHalide.so         runtime
```

- 运行 Halide 小程序

```
cd /workspace/Halide/halide-build
./tutorial/lesson_01_basics
```

运行结果

```
Success!
```

- 运行 Halide 的 Python 接口首先查看 Python 的系统路径

```
python
>>>import sys
>>> sys.path
['', '/root', '/workspace/Halide/halide-build/python_bindings/src', '/usr/lib/
↪python36.zip', '/usr/lib/python3.6', '/usr/lib/python3.6/lib-dynload', '/usr/
↪local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages']
```

可以看到 Python 的系统路径已经有 Halide 的编译后的 python 包路径'/workspace/Halide/halide-build/python_bindings/src'

```
python
>>> import halide
```

直接 import halide 成功!

3.5 Tengine in Docker

Autokernel docker 里已安装好 Tengine，相关文件都在/workspace/Tengine/目录下

```
cd /workspace/Tengine/build
```

- Tengine 相关头文件在/workspace/Tengine/build/install/include

```
root@bd3faab0f079:/workspace/Tengine/build/install/include# ls

tengine_c_api.h
tengine_cpp_api.h
```

- 编译好的 Tengine 库在/workspace/Tengine/build/install/lib 目录下，可以看到 libtengine-lite.so

```
root@bd3faab0f079:/workspace/Tengine/build/install/lib# ls

libtengine-lite.so
```

- 运行 Tengine 小程序

  该示例跑了 Tengine 在目标电脑上各个网络模型的性能 benchmark

```
cd /workspace/Tengine/benchmark
../build/benchmark/tm_benchmark
```

运行结果

```
start to run register cpu allocator
loop_counts = 1
num_threads = 1
power       = 0
tengine-lite library version: 1.0-dev
    squeezenet_v1.1  min =   32.74 ms   max =   32.74 ms   avg =   32.74 ms
        mobilenetv1  min =   31.33 ms   max =   31.33 ms   avg =   31.33 ms
```

(下页继续)

(续上页)

```
        mobilenetv2  min =   35.55 ms   max =   35.55 ms   avg =   35.55 ms
        mobilenetv3  min =   37.65 ms   max =   37.65 ms   avg =   37.65 ms
        shufflenetv2  min =   10.93 ms   max =   10.93 ms   avg =   10.93 ms
            resnet18  min =   74.53 ms   max =   74.53 ms   avg =   74.53 ms
            resnet50  min =  175.55 ms   max =  175.55 ms   avg =  175.55 ms
        googlenet  min =  133.23 ms   max =  133.23 ms   avg =  133.23 ms
        inceptionv3  min =  298.22 ms   max =  298.22 ms   avg =  298.22 ms
            vgg16  min =  555.60 ms   max =  555.60 ms   avg =  555.60 ms
                mssd  min =   69.41 ms   max =   69.41 ms   avg =   69.41 ms
        retinaface  min =   13.14 ms   max =   13.14 ms   avg =   13.14 ms
        yolov3_tiny  min =  132.67 ms   max =  132.67 ms   avg =  132.67 ms
    mobilefacenets  min =   14.95 ms   max =   14.95 ms   avg =   14.95 ms
ALL TEST DONE
```

CHAPTER 4

AutoSearch

AutoSearch 是一个对 Halide 算子进行策略搜索和自动优化的模块，它支持生成 cpu 和 gpu 的 schedule，并且可以生成运行在不同平台（x86 或 arm）的代码文件。除了自动优化算子策略外，我们还增加了对输入数据排布方式的优化 (data transform)。

4.1 安装

在使用 AutoSearch 前必须先安装 Halide，可以直接使用 AutoKernel 提供的 docker（里面已安装了 Halide）

```
docker pull openailab/autokernel
# /workspace/Halide
```

执行以下终端指令：

```
export HALIDE_HOME=<path>/<to>/Halide
cd <path>/<to>/AutoSearch
mkdir build && cd build
cmake ..
make -j `nproc`
```

4.2 快速使用

以下 matmul 测试的 shape config 均为 M=N=K=512。

4.2.1 Manuel schedule 的编译和自动测试耗时

1. CPU: x86-64-linux

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target x86-64-linux -
↪compute_time
```

在 Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz 上测试时间为：

```
autokernel time:        1.79585 ms
```

2. Nvida GPU: CUDA

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target x86-64-linux-cuda -
↪compute_time
```

在 GeForce GTX 1080 Ti 上测试时间为：

```
autokernel time:        0.6114 ms
```

3. ARM CPU 该步骤依赖 aarch64 交叉编译工具链 `aarch64-linux-gnu-g++`

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target arm-64-linux -
↪compute_time --num_iterators 10
```

把可执行文件复制到 RK3399 板子上，并执行

```
scp samples/demo_run firefly@xx.xx.xx.xx:/home/firefly
ssh firefly@xx.xx.xx.xx
cd /home/firefly
./demo_run
```

在 RK3399 上测试时间为：

```
autokernel time:        245.7393 ms
```

4. ARM Mali GPU: Opencl

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target arm-64-linux-opencl␣
↪ -compute_time --num_iterators 10
```

在 RK3399 Mali-T860 上测试时间为：

```
autokernel time:        126.4644 ms ms
```

4.2.2 AutoTune: 自动生成调优策略并自动测试时间

1. CPU: x86-64-linux

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target x86-64-linux -
↪autotune -compute_time
```

在 Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz 上测试时间为：

```
autokernel time:        0.880885 ms
```

2. Nvida GPU: CUDA

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target x86-64-linux-cuda -
↪autotune -compute_time
```

在 GeForce GTX 1080 Ti 上测试时间为：

```
autokernel time:        0.604405 ms
```

3. ARM CPU

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target arm-64-linux-opencl␣
↪-autotune -compute_time --num_iterators 10
```

在 RK3399 上测试时间为：

```
autokernel time:         470.75 ms ms
```

该生成的策略在 arm-cpu 上性能一般，因为这个自动生成步骤基于 baseline.weights, 并没有在 arm cpu 上进行重训练

4. ARM Mali GPU

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target arm-64-linux -
↪autotune -compute_time  --num_iterators 20
```

在 RK3399 Mali-T860 上测试时间为：

```
autokernel time:          87.249 ms
```

4.2.3 DataTransform

在自动调优的过程，可以使用-datatransform 来自动选择合适的数据排布，目前仅支持 matmul 这个算子

```
cd toolkit
python3 tools.py --gen ../generator/batch_matmul.cpp --target x86-64-linux -autotune -
↪compute_time -datatransform
```

得到的性能结果如下：

4.2.4 正确性校验模板

用户可以参考 toolkit/template/demo_eval.cpp 文件编写对应的校验代码，在 matmul(shape 为 1_512_512_512) 的例子中，生成的.h.s 文件在 toolkit/samples/目录下，执行如下指令生成可执行文件：

```
export HALIDE_BUILD=/workspace/Halide/halide_build
# For x86-64-linux:
g++ demo_eval.cpp demo_1_512_512_512.s -I $HALIDE_BUILD/include  -ldl -lpthread -o␣
↪demo_eval

# For arm-64-linux:
aarch64-linux-gnu-g++ demo_eval.cpp demo_1_512_512_512.s -I $HALIDE_BUILD/include  -
↪ldl -lpthread -o demo_eval
```

如果结果和 ref_func 一致，将得到正确性验证通过：

```
Correctness check passed!
```

4.3 Toolkit 参数说明

- `--gen`: 生成文件，在 generator 目录下，描述算子的计算过程, 编译后将 toolkit/samples 下获得生成的.h,.s,.schedule.h 等代码文件
- `--target`: 后端，我们已测试以下后端:
  - x86-64-linux
  - x86-64-linux-opencl
  - x86-64-linux-cuda
  - arm-64-linux
  - arm-64-linux-opencl

  更多后端:
  - arch:arm, hexagon, mips, powerpc, riscv, wasm, x86.
  - bits：32, 64
  - os：android, ios, linux, windows…
  - features: avx, avx2, avx512, cl_half, cuda, opencl…
- `-autotune`: 自动调优生成 schedule
  - CUDA: 使用 sioutas2020 autoschedule
  - OpenCL: 使用 li2018 autoschedule
  - CPU: 使用 adams2019 autoschedule，可以通过调整 num_tune_loops 和 batch_size 参数值来控制优化效果和调优时间。
    * num_tune_loops: 用于 retrain cost model 的迭代次数
    * batch_size: 每个迭代中，用于提取特征和执行时间的样本个数
- `-compute_time`:
  - compute_samples: 测试时间的样本数量，取最小时间
  - num_iterators: 测试重复次数，取平均时间该测试时间的伪代码如下:

```
avg_times=[]
for i in compute_samples:

    time_start
    for j in num_iterators:
        //func
    time_end
    avg_time=(time_end-time_start)/num_iterators
```

(下页继续)

(续上页)

```
    avg_times.append(avg_time)
print("autokernel time:  min(avg_times)")
```

- `data transform`: 在自动调优的过程，可使用该参数来自动选择合适的数据排布
- `shape_config.py`：用于配置算子的 shape。args_dict 中的 key 是用户 xxx_gen.cpp 中设置的 demo 名称，在矩阵乘法的例子中，demo 的函数名称在 `generator/batch_matmul.cpp` 文件中在此处定义

```
HALIDE_REGISTER_GENERATOR(BatchMatmul, matmul)
```

CHAPTER 5

AutoKernel Plugin 快速入门

使用 docker 镜像配置开发环境

```
# 拉取镜像(可能需要一段时间，请耐心等待)
docker pull openailab/autokernel
# 启动容器，进入开发环境
docker run -it openailab/autokernel /bin/bash
```

docker 里面提供了安装好的 Halide 和 Tengine

```
/workspace/Halide        # Halide
/workspace/Tengine  # Tengine
```

克隆 AutoKernel 项目

```
git clone https://github.com/OAID/AutoKernel.git
```

我们首先看看 autokernel_plugin/src/的文件目录:

```
autokernel_plugin/src/
|-- CMakeLists.txt
|-- direct_conv
|   |-- build.sh
|   |-- direct_conv.cpp
|   |-- direct_conv.h
|   |-- direct_conv_gen.cc
```

(下页继续)

(续上页)

```
|-- im2col_conv
|   |-- build.sh
|   |-- im2col_conv.cpp
|   |-- im2col_conv.h
|   `-- im2col_conv_gen.cc
`-- plugin_init.cpp
```

可以看到 src 目录下有两个文件夹，每个文件夹的目录下有：

- xxx_gen.cc, 用 Halide 语言的算子描述 (algorithm) 和调度策略（schedule)
- build.sh 用于编译 xxx_gen
- xxx.h 和 xxx.cpp 是用 Tengine 算子接口封装的算子实现

一键生成算子汇编代码

```
cd AutoKernel/autokernel_plugin
find . -name "*.sh" | xargs chmod +x
./scripts/generate.sh  #自动生成算子汇编文件
```

运行完这一步，可以看到原来的目录下多了两个自动生成的文件：

```
|-- im2col_conv
|   |-- halide_im2col_conv.h
|   |-- halide_im2col_conv.s
|-- direct_conv
|   |-- halide_direct_conv.h
|   `-- halide_direct_conv.s
```

接下来使用自动生成的文件，把 Autokernel 注册进 tengine，一键编译 libAutoKernel.so

```
mkdir build
cd build
cmake ..
make -j4
```

生成的库在/workspace/AutoKernel/autokernel_plugin/build/src/libautokernel.so 运行测试，在测试代码中调用 load_tengine_plugin()：

```
cd AutoKernel/autokernel_plugin
./build/tests/tm_classification -n squeezenet
```

分类网络的运行结果如下：

```
AutoKernel plugin inited
function:autokernel_plugin_init executed

...

Repeat 1 times, avg time per run is 55.932 ms
max time is 55.932 ms, min time is 55.932 ms
--------------------------------------
0.2732 - "n02123045 tabby, tabby cat"
0.2676 - "n02123159 tiger cat"
0.1810 - "n02119789 kit fox, Vulpes macrotis"
0.0818 - "n02124075 Egyptian cat"
0.0724 - "n02085620 Chihuahua"
--------------------------------------
ALL TEST DONE
```

可以看到，输出结果显示调用了 AutoKernel plugin 里的函数。

CHAPTER 6

Tengine 快速入门

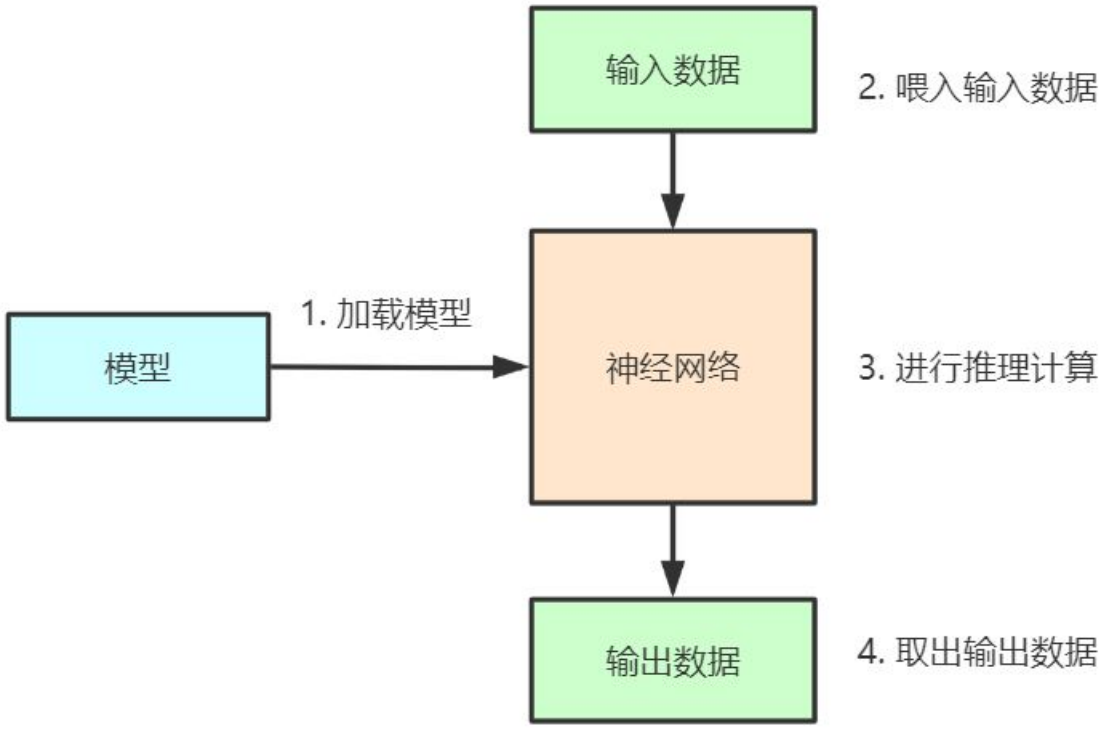
Tengine 是一个轻量级深度神经网络推理引擎。本文档将在 x86 Linux 平台，以分类模型（Squezenet 模型）为例，带你快速上手 Tengine。

6.1 深度学习神经网络计算流程

概念理解

- 神经网络: 神经网络可以理解为计算图 (graph)，一个计算图由多个算子 (operator) 节点组成，这些节点可以是卷积算子 (Convolution), 池化算子（Pooling), 全连接算子 (Fc) 等。
- 神经网络模型: 神经网络模型是由深度学习训练框架（Tensorflow, Caffe, Pytorch, Mxnet 等）训练得到的，模型包含了两个信息:
  - 神经网络的计算图结构
  - 算子的权重数据

计算流程



inference

1. 加载模型：得到神经网络结构和权重数据
2. 准备输入数据，喂入输入数据
3. 进行模型推理计算
4. 获取输出数据

6.2 Tengine Squeezenet 示例

本示例将按照神经网络推理计算流程，演示如何在 Tengine 中进行 Squeezenet 分类网络的推理计算

1. 加载模型

```
/* load model */
graph_t graph = create_graph(NULL, "tengine", model_file);
```

`model_file` 是 tengine 格式的模型文件：”squeezenet.tmfile”

2. 准备输入数据, 喂入输入数据

```
/* prepare input data */
tensor_t input_tensor = get_graph_input_tensor(graph, 0, 0);
set_tensor_shape(input_tensor, dims, 4);
set_tensor_buffer(input_tensor, input_data, img_size * sizeof(float));
```

3. 进行模型推理计算

```
/* forward */
run_graph(graph, 1);
```

4. 获取输出数据

```
/* get result */
tensor_t output_tensor = get_graph_output_tensor(graph, 0, 0);
float* output_data = ( float* )get_tensor_buffer(output_tensor);
```

- 代码：
  - 完整的代码源文件在: data/02_tengine_tutorial.cpp, 代码非常清晰简洁 ~
  - 代码使用了一些工具函数，在文件tengine_operations.h中
- 编译

```
cd tutorials/data
cp /workspace/Tengine/examples/common -r .
mkdir build
cd build
cmake ..
make
```

- 执行

```
cd tutorials/data/build

#下载模型和图片
wget https://github.com/OAID/TengineModels/raw/main/images/cat.jpg .
wget https://github.com/OAID/TengineModels/raw/main/tmfiles/squeezenet.tmfile .
./02_tengine_tutorial
```

得到结果

```
0.273198, 281
0.267550, 282
0.181006, 278
0.081798, 285
0.072406, 151
--------------------------------------
ALL TEST DONE
```

这是一个分类网络，1000 类，index 从 0 到 999，每个类别有一个概率分数，运行结果打印出了排名前 5 的概率分数 score 和 index.

6.3 更多 Tengine 示例

更多 Tengine 的应用示例在Tengine/examples：

- 分类任务
- 人脸关键点检测任务
- ssd 目标检测任务
- retinaface 人脸检测任务
- yolact 实例分割任务
- yolov3 目标检测任务
- yolov4-tiny 目标检测任务
- openpose 人体姿态识别任务
- crnn 汉字识别任务

CHAPTER 7

Halide

7.1 Halide 简介

Halide 是基于 C++ 的领域特定语言 domain specific language (DSL)，用于图像处理的高性能算子自动生成。

7.1.1 编译

Halide 支持两种编译模式:

- Ahead of Time (AOT)
- Just in Time (JIT)

7.1.2 目标硬件支持

Halide 目前支持的后端:

- CPU architectures: X86, ARM, MIPS, Hexagon, PowerPC, RISC-V
- Operating systems: Linux, Windows, macOS, Android, iOS, Qualcomm QuRT
- GPU Compute APIs: CUDA, OpenCL, OpenGL Compute Shaders, Apple Metal, Microsoft Direct X 12

7.1.3 Halide 资源

- Halide 官网: http://halide-lang.org/
- github: https://github.com/halide/Halide
- API 文档: http://halide-lang.org/docs
- Halide 教程 想要对 Halide 有更多了解，欢迎自行学习

7.2 Halide 算法描述

下面以简单的函数来演示 Halide 语言的基本数据结构。

7.2.1 函数 Func

和数学上的函数类似，定义了一个计算过程。复杂的计算过程可以拆成多个小函数来实现。例子：函数 f 的每个像素的值是其 x 和 y 坐标之和。

```
f(x, y) = x + y;
```

例子：多个函数的定义，函数 g 调用函数 f, 函数 h 调用函数 g。

```
f(x, y) = x + y;
g(x, y) = f(x, y) + 1;
h(x, y) = g(x, y) * 2;
```

7.2.2 变量 Var

可以理解为函数的自变量，比如要描述一个图像的像素，需要两个变量 x 和 y 来描述 w 维度和 h 维度的坐标。

```
blur_x(x , y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
```

7.2.3 Expr

表达式 Expression 和函数 Func 类似，定义一个计算过程

```
Expr e = x + y;
Output(x, y) = 3*e + x;
```

7.3 Halide 调度策略 Schedule

7.3.1 循环调度策略

该部分的调度策略主要考虑在一个 stage 内的循环调度策略，主要有以下调度原语：

7.4 Halide 初体验

在深入了解 Halide 之前，我们先来体验一下 Halide 的黑魔法。

进入 AutoKernel 的 docker, docker 里已有 Halide 的 python 环境，直接运行

```
python data/03_halide_magic.py
```

可以得到输出

```
func_origin__ cost 0.510215 second
func_parallel cost 0.122265 second
```

以上这个脚本执行了一个简单的函数计算：`func[x,y] = x + 10*y` 对比了两个函数的运行时间：

- func_origin: 默认函数
- func_parallel: 添加了 Halide 的一个调度策略：`func.parallel(y,4)`, 对 y 维度进行并行化，并行度为 4

结果可以看到，第二个函数的耗时是第一个函数的四分之一。

无需底层优化汇编知识，只需添加一行代码，就能得到比较好的优化效果

7.4.1 Halide 语言基础

要想调用 Halide 的调度策略，首先要掌握基本的 Halide 语言，用 Halide 语言来描述算子的计算。下面以简单的函数来演示 Halide 语言的基本数据结构。

- 变量 `Var`：可以理解为函数的自变量，比如要描述一个图像的像素，需要两个变量 x 和 y 来描述 w 维度和 h 维度的坐标。
- 函数 `Func`：和数学上的函数类似，定义了一个计算过程。复杂的计算过程可以拆成多个小函数来实现。

示例一

本例子的函数计算公式为：`func(x,y)= 10*y + x` 用 Halide 语言来描述这个函数：

- Python:

```
import halide as hl

x, y = hl.Var("x"), hl.Var("y")
func = hl.Func("func")
func[x,y] = x + 10*y
```

- C++

```
#include "Halide.h"
using namespace Halide;

Var x("x"), y("y");
Func func("func");

func(x, y) = x + 10 * y;
```

Func 的 realize 会计算函数在定义域的值并返回数值结果。调用了 realize，函数才被即时编译 (jit-compile), 在这之前只是定义了函数的计算过程。

查看计算结果

- Python:

```
out = func.realize(3, 4)   # width, height = 3,4

for j in range(out.height()):
    for i in range(out.width()):
        print("out[x=%i,y=%i]=%i"%(i,j,out[i,j]))
```

- C++

```
Buffer<int32_t> out = func.realize(3, 4);

for (int j = 0; j < out.height(); j++) {
    for (int i = 0; i < out.width(); i++) {
        printf("out[x=%d,y=%d]=%d",i,j,out(i,j));
        }
    }
```

这个函数的计算是：

```
                    wide = 3
                  x=0 x=1 x=2
                ------------
            y=0 |  0   1   2
hight = 4   y=1 | 10  11  12
            y=2 | 20  21  22
            y=3 | 30  31  32
```

完整的代码在data/03_halide_basic.py 可以直接运行：

```
python data/03_halide_basic.py
```

另外可以调用 `func.trace_stores()` 来跟踪函数的值

示例二

本示例演示如何喂入输入数据，取出输出数据完整的代码在data/03_halide_feed_data.py

本示例的函数：

```
B(x,y)=A(x,y)+1
```

A 是输入数据，可以定义 Halide.Buffer, 然后把 numpy 的 array 数据喂入 buffer

```
    # feed input
    input_data = np.ones((4,4),dtype=np.uint8)
    A = hl.Buffer(input_data)
```

定义函数 B

```
    i,j = hl.Var("i"), hl.Var("j")
    B = hl.Func("B")
    B[i,j] = A[i,j] + 1
```

获取输出数据, 有以下几种方式

```
    # 1
        output = B.realize(4,4)
        print("out: \n",np.asanyarray(output))
    # 2
        output = hl.Buffer(hl.UInt(8),[4,4])
        B.realize(output)
        print("out: \n",np.asanyarray(output))
    # 3
        output_data = np.empty(input_data.shape, dtype=input_data.dtype,order="F")
```

(下页继续)

(续上页)

```
        output = hl.Buffer(output_data)
        B.realize(output)
        print("out: \n",output_data)
```

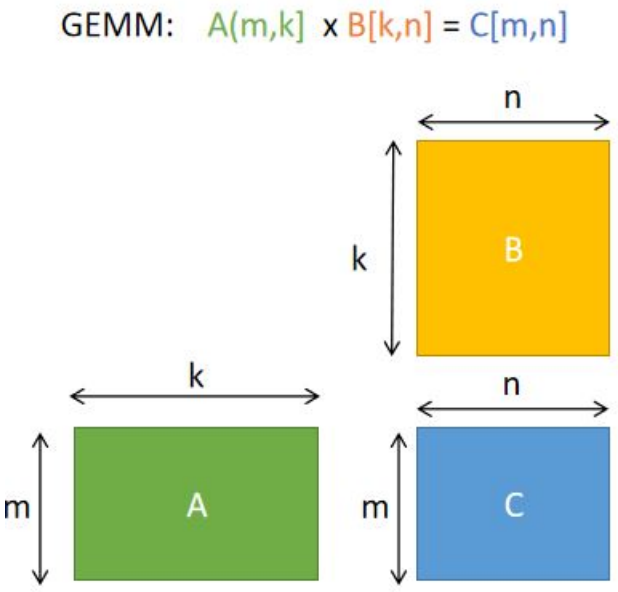
可以直接运行完整代码：

```
python data/03_halide_feed_data.py
```

CHAPTER 8

x86 平台的 GEMM 优化

本教程将带领大家逐步优化矩阵乘法 GEMM。无需手工撸代码，编写繁杂冗长的底层汇编代码，只需十几行简洁的调度代码。



gemm.jpg

运行环境搭建： AutoKernel 提供了 docker 镜像，docker 里已经配置好运行环境，进入 docker 即可直接运行 demo 代码：

```
# 拉取镜像
docker pull openailab/autokernel
```

(下页继续)

(续上页)

```
# 启动容器，进入开发环境
docker run -it openailab/autokernel /bin/bash
# 获取代码
git clone https://github.com/OAID/AutoKernel.git
cd AutoKernel/doc/tutorials/data/
```

目录下的 build.sh 是 demo 的执行脚本，运行需要指定优化步骤 step，可选的 step 是从 1 到 7，其中 step= 1 是默认不优化的，step=7 是最极致优化的。**优化效果：**

```
# 执行demo
./build.sh 1
./build.sh 7
```

下图展示了在 Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz 的电脑上的优化效果，无需手工撸代码，无需编写繁杂冗长的底层汇编代码，只需十几行简洁的调度代码, 就能性能优化 200+ 倍。

下面是详细的优化步骤：STEP 1 第一个步骤是不带任何优化的。用 Halide 语言直接描述 GEMM 的计算过程。

```
1. Var x,y;
2. RDom k(0, K);
3. Func gemm("gemm");
4. gemm(x, y) += A(k, y) * B(x, k);
```

计算 M=N=K=640 的矩阵乘法。运行脚本第一个参数指定 step=1。耗时结果如下：

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 1
step =  1
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      240.8523 ms  ␣
↪  1.1376 ms
```

STEP 2 这一步我们采用分块 tile。分块的目的是为了充分利用缓存。如果原来的循环较大，tile 分块改成小块数据去计算，可以使得每次计算的数据都比较舒适地呆在缓存里，不用经历重复的驱逐（在缓存中重复的添加和删除数据)。分块后进行 reorder 操作，交换两个嵌套循环的顺序，目的是最内层的内存访问友好。我们按照 x,y 维度划分成 16x8 的小分块去计算：

```
1.         gemm.update()
2.             .tile(x, y, xo, yo, xi, yi, 16, 8)
3.             .reorder(xi, yi, k, xo, yo);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 2
step =  2
```

(下页继续)

(续上页)

```
M N K = 640 640 640     err 0.00        [rep 50] halide | blas  81.8148 ms      1.
↪1281 ms
```

性能从 240ms 优化到 82ms，提升了近 3 倍。

STEP 3 我们在上一步的基础上增加向量化 vectorize。向量化是把几个标量计算（scale) 转换为一个向量计算（vector), 充分利用 SIMD 向量指令。大部分现代 CPU 支持 SIMD（Single Instruction Multiple Data，单指令流多数据流）。在同一个 CPU 循环中，SIMD 可在多个值上同时执行相同的运算/指令。

```
1.          gemm.update()
2.                  .tile(x, y, xo, yo, xi, yi, 16, 8)
3.                  .reorder(xi, yi, k, xo, yo)
4.                  .vectorize(xi, 8);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 3
step =  3
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      27.5433 ms  ␣
↪  1.1445 ms
```

性能从 82ms 优化到 27ms，又加速了接近 3 倍。可以看到，围绕前面提到的两条优化宗旨：优化内存访问和提高并行性，从 step1 到 step3，性能已经提升了近 9 倍。

STEP 4 调度策略在 step3 的基础上增加并行化 parallel。对一个循环并行化是把循环的每次迭代分给多个线程或者处理器去同时处理，每个线程处理通过代码段（loop body), 但是处理不同的数据。

```
1.          gemm(x, y) += A(k, y) * B(x, k);
2.          gemm.update()
3.              .tile(x, y, xo, yo, xi, yi, 16, 8)
4.              .reorder(xi, yi, k, xo, yo)
5.              .vectorize(xi, 8)
6.              .parallel(yo);
```

执行结果

```
root@bd3faab0f079:/home/chunying/AutoKernel/doc/tutorials# ./06_build.sh 4
step =  4
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      7.2605 ms  ␣
↪  1.1605 ms
```

增加并行化后，build.sh 默认指定四线程，性能直接翻了近 4 倍，从 27ms 到 7.3ms。

STEP 5 调度策略在上一步的基础上增加 unroll 展开。如果循环体内的语句没有数据相关依赖，循环展开可以增加并发执行的机会，使得更充分利用寄存器，减少循环时每个操作内存加载和保存的次数。

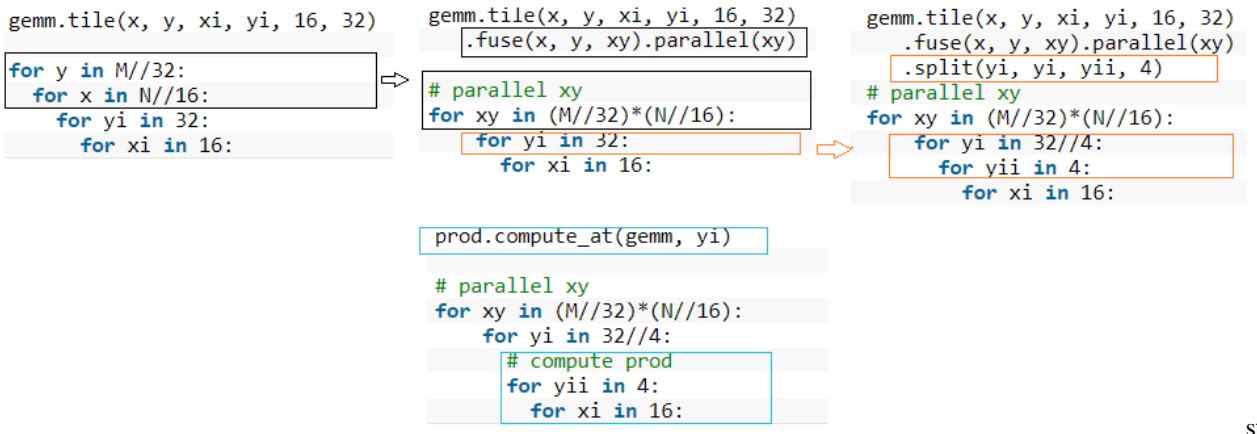
```
1.          gemm.update()
2.              .tile(x, y, xo, yo, xi, yi, 16, 8)
3.              .reorder(xi, yi, k, xo, yo)
4.              .vectorize(xi, 8)
5.              .parallel(yo)
6.              .unroll(xi)
7.              .unroll(yi,2);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 5
step =  5
M N K = 640 640 640      err 0.00         [rep 50] autokernel | blas       4.7617 ms
↪  1.1597 ms
```

unroll 展开后，性能从 7.3ms 优化到 4.8ms。

STEP 6 前面的分块成 16 x 8 的小 kernel, 这一步先划分成 16 x 32 的分块，然后把每个分块再分成 16 x 8 的子分块。我们把最外层的两层循环合并到一层，并对这一层进行并行化。这一步计算描述多了一个 prod 函数来定义子分块的计算，prod 函数的计算公式和总的 gemm 是一样的，我们通过 compute_at 指定在 yi 维度之下计算 prod，则 prod 计算的是 16x8 的小 kernel, 大致逻辑如下：



step6.png

总的代码如下：

```
1.          Func prod;
2.          prod(x, y) += A(k, y) * B(x, k);
3.          gemm(x, y) = prod(x, y);
4.
5.          gemm.tile(x, y, xi, yi, 16, 32)
6.              .fuse(x, y, xy).parallel(xy)
7.              .split(yi, yi, yii, 4)
8.              .vectorize(xi, 8)
9.              .unroll(xi)
```

(下页继续)

(续上页)

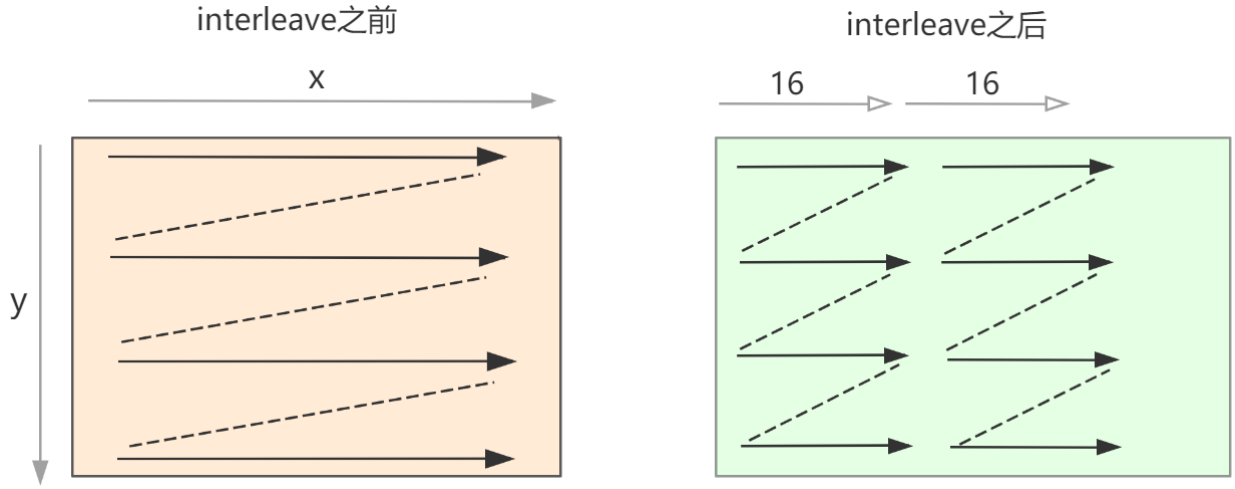
```
10.             .unroll(yii);
11.
12.         prod.compute_at(gemm, yi)
13.             .vectorize(x, 8).unroll(y);
14.
15.         prod.update()
16.             .reorder(x, y, k)
17.             .vectorize(x, 8)
18.             .unroll(x)
19.             .unroll(y)
20.             .unroll(k, 2);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 6
step =  6
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      3.1824 ms
↪  1.1373 ms
```

这一步距离 STEP1 性能已经优化了近 80 倍了，性能越来越接近 OpenBlas 了。

STEP 7 这一步添加的操作是对矩阵 B 进行数据重排，使得在计算小 kernel 16x8 时，内存读取更顺畅。因为小 kernel 的 x 维度是按照 16 划分的，因此重排数据 B 的 x 维度也是按照 16 重排。



interleave.png

总的代码如下：

```
1.         Func B_interleave("B"), Bs("Bs");
2.         Bs(x, y, xo) = B(xo * 16 + x, y);
3.         B_interleave(x, y) = Bs(x % 16, y, x / 16);
4.
5.         Func prod;
```

(下页继续)

(续上页)

```
6.          prod(x, y) += A(k, y) * B_interleave(x, k);
7.          gemm(x, y) = prod(x, y);
8.
9.          gemm.tile(x, y, xi, yi, 16, 32)
10.              .fuse(x, y, xy).parallel(xy)
11.              .split(yi, yi, yii, 4)
12.              .vectorize(xi, 8)
13.              .unroll(xi)
14.              .unroll(yii);
15.
16.          prod.compute_at(gemm, yi)
17.              .vectorize(x, 8).unroll(y);
18.
19.          prod.update()
20.              .reorder(x, y, k)
21.              .vectorize(x, 8)
22.              .unroll(x)
23.              .unroll(y)
24.              .unroll(k, 2);
25.          Bs.compute_root()
26.              .split(y, yo, yi, 16)
27.              .reorder(x, yi, xo, yo)
28.              .unroll(x)
29.              .vectorize(yi).parallel(yo, 4);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 7
step =  7
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      1.1957 ms
↪  1.1425 ms
```

至此，我们的每一步调优策略始终都围绕两条优化宗旨“优化内存访问”，“提高并行性”展开优化，到最后性能已经与 OpenBlAS 差不多了，距离 STEP1 已经加速了 200+ 倍了。

CHAPTER 9

AutoKernel 实力展示：将 GEMM 的性能提升 200 倍!

随着 AI 技术的快速发展，深度学习在各个领域得到了广泛应用。深度学习模型能否成功在终端落地应用，满足产品需求，一个关键的指标就是神经网络模型的推理性能。于是，一大波算法工程师为了算法的部署转岗算子优化工程师。然而，优化代码并不是一件简单的事，它要求工程师既要精通计算机体系架构，又要熟悉算法的计算流程，于是，稍微有经验的深度学习推理优化工程师都成了各家公司争抢的“香饽饽”。人才少，需求多，算子优化自动化是未来的大趋势。

近日，致力于降低优化门槛，提升优化开发效率的算子自动优化工具 AutoKernel 正式开源了。

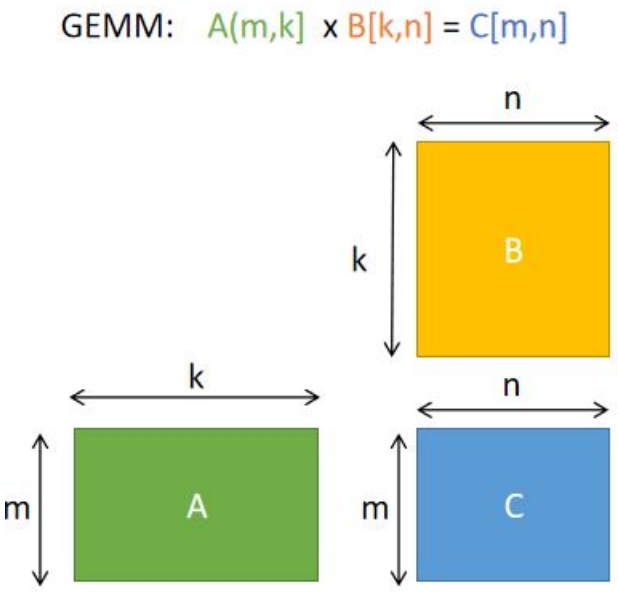
9.1 AutoKernel 特色

- 低门槛：无需底层优化汇编的知识门槛
- 简单易用: 提供 docker 环境，无需安装环境，plugin 一键集成到推理框架 Tengine
- 高效率: 无需手写优化汇编，一键生成优化代码，一键部署

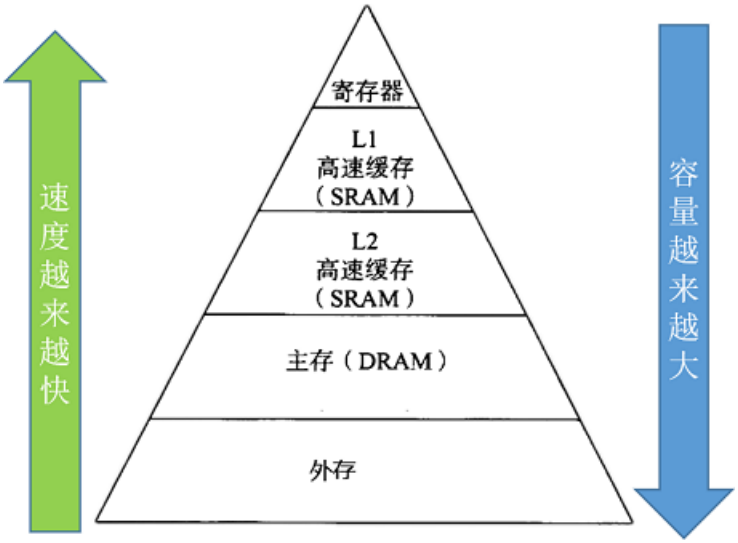
AutoKernel 使用业界广泛使用的自动代码生成项目 Halide，通过输入计算描述和调度策略，自动生成底层代码。AutoKernel 支持以 plugin 的形式，将生成的自动优化算子一键部署到推理框架 Tengine 中。

9.2 GEMM 优化教程

下面，本教程将带领大家一步步优化矩阵乘法 GEMM。无需手工撸代码，编写繁杂冗长的底层汇编代码，只需十几行简洁的调度代码。



两个优化宗旨：在详细讲解优化步骤前，我们先谈谈优化的本质。我们在谈”优化“的时候，计算机底层做了什么？优化的”瓶颈“是什么？为什么通过一波”优化操作“，性能就能提升呢？AutoKernel 使用的 Halide 是如何实现自动优化的呢？要解答这些疑问，我们需要了解一下硬件的基础的体系结构，了解硬件如何工作，才能在软件上实现算法的时候，尽可能去考虑利用硬件的一些特性，来做到高效的、极致的优化。



上图是典型的存储理器层次结构：主存容量大，访问速度慢，寄存器和缓存读取速度快，但容量有限。在寄存器的层级上，CPU 可以在一个时钟周期内访问它们，如果 CPU 去访问外部的 DDR 的话，延迟是非常大的，大概是 200 个时钟周期左右。如果 CPU 去访问 cache 的话，一般需要 6 到 12 个 cycle 就够了。所以，一个很重要的一个优化宗旨是：**优化内存访问**，充分利用寄存器和高速缓存去存数据。第二个优化宗旨是 **提高并行性**：充分利用 SIMD 进行指令向量化和多核心并行。大部分现代 CPU 支持 SIMD（Single Instruction Multiple Data，单指令流多数据流）。在同一个 CPU 循环中，SIMD 可在多个值上同时执行相同的运算/指令。如果我们在 4 个数据点上进行向量化，一次计算四个数据，理论上就可以实现 4 倍的加速。

运行环境搭建：AutoKernel 提供了 docker 镜像，docker 里已经配置好运行环境，进入 docker 即可直接运行 demo 代码：

```
# 拉取镜像
docker pull openailab/autokernel
# 启动容器，进入开发环境
docker run -it openailab/autokernel /bin/bash
# 获取代码
git clone https://github.com/OAID/AutoKernel.git
cd AutoKernel/doc/tutorials/data/
```

目录下的 build.sh 是 demo 的执行脚本，运行需要指定优化步骤 step，可选的 step 是从 1 到 7，其中 step= 1 是默认不优化的，step=7 是最极致优化的。**优化效果：**

```
# 执行 demo
./build.sh 1
./build.sh 7
```

下图展示了在 Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz 的电脑上的优化效果，无需手工撸代码，无需编写繁杂冗长的底层汇编代码，只需十几行简洁的调度代码, 就能性能优化 200+ 倍。

下面是详细的优化步骤：STEP 1 第一个步骤是不带任何优化的。用 Halide 语言直接描述 GEMM 的计算过程。

```
1. Var x,y;
2. RDom k(0, K);
3. Func gemm("gemm");
4. gemm(x, y) += A(k, y) * B(x, k);
```

计算 M=N=K=640 的矩阵乘法。运行脚本第一个参数指定 step=1。耗时结果如下：

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 1
step = 1
M N K = 640 640 640 err 0.00 [rep 50] autokernel | blas 240.8523 ms ␣
↪ 1.1376 ms
```

STEP 2 这一步我们采用分块 tile。分块的目的是为了充分利用缓存。如果原来的循环较大，tile 分块改成小块数据去计算，可以使得每次计算的数据都比较舒适地呆在缓存里，不用经历重复的驱逐（在缓存中重复的添加和删除数据)。分块后进行 reorder 操作，交换两个嵌套循环的顺序，目的是最内层的内存访问友好。我们按照 x,y 维度划分成 16x8 的小分块去计算：

```
1.         gemm.update()
2.             .tile(x, y, xo, yo, xi, yi, 16, 8)
3.             .reorder(xi, yi, k, xo, yo);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 2
step = 2
M N K = 640 640 640 err 0.00 [rep 50] halide | blas 81.8148 ms 1.
↪1281 ms
```

性能从 240ms 优化到 82ms，提升了近 3 倍。

STEP 3 我们在上一步的基础上增加向量化 vectorize。向量化是把几个标量计算（scale) 转换为一个向量计算（vector), 充分利用 SIMD 向量指令。大部分现代 CPU 支持 SIMD（Single Instruction Multiple Data，单指令流多数据流)。在同一个 CPU 循环中，SIMD 可在多个值上同时执行相同的运算/指令。

```
1.         gemm.update()
2.                 .tile(x, y, xo, yo, xi, yi, 16, 8)
```

(下页继续)

(续上页)

```
3.                .reorder(xi, yi, k, xo, yo)
4.                .vectorize(xi, 8);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 3
step =  3
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      27.5433 ms
↪  1.1445 ms
```

性能从 82ms 优化到 27ms，又加速了接近 3 倍。可以看到，围绕前面提到的两条优化宗旨：优化内存访问和提高并行性，从 step1 到 step3，性能已经提升了近 9 倍。

STEP 4 调度策略在 step3 的基础上增加并行化 parallel。对一个循环并行化是把循环的每次迭代分给多个线程或者处理器去同时处理，每个线程处理通过代码段（loop body), 但是处理不同的数据。

```
1.        gemm(x, y) += A(k, y) * B(x, k);
2.        gemm.update()
3.            .tile(x, y, xo, yo, xi, yi, 16, 8)
4.            .reorder(xi, yi, k, xo, yo)
5.            .vectorize(xi, 8)
6.            .parallel(yo);
```

执行结果

```
root@bd3faab0f079:/home/chunying/AutoKernel/doc/tutorials# ./06_build.sh 4
step =  4
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      7.2605 ms
↪  1.1605 ms
```

增加并行化后，build.sh 默认指定四线程，性能直接翻了近 4 倍，从 27ms 到 7.3ms。

STEP 5 调度策略在上一步的基础上增加 unroll 展开。如果循环体内的语句没有数据相关依赖，循环展开可以增加并发执行的机会，使得更充分利用寄存器，减少循环时每个操作内存加载和保存的次数。

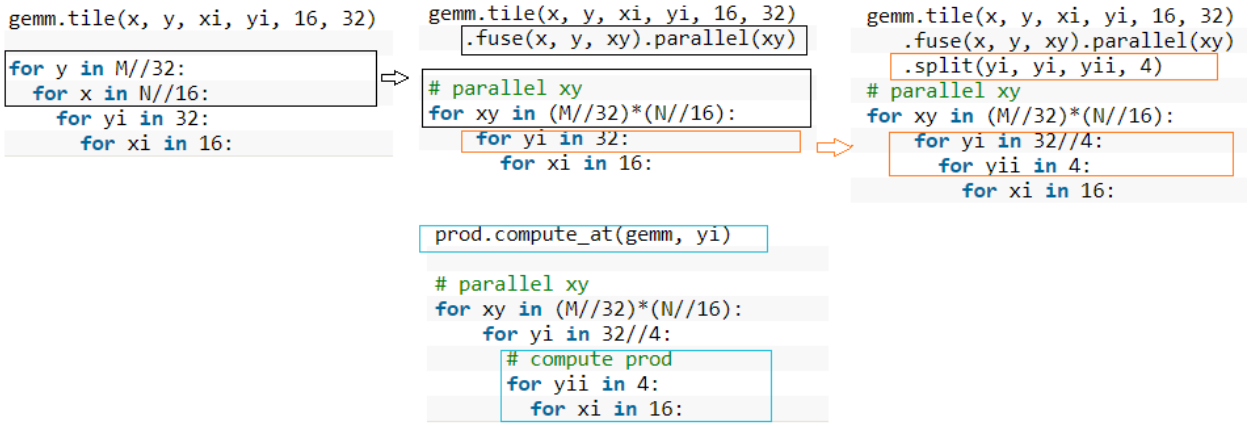
```
1.        gemm.update()
2.            .tile(x, y, xo, yo, xi, yi, 16, 8)
3.            .reorder(xi, yi, k, xo, yo)
4.            .vectorize(xi, 8)
5.            .parallel(yo)
6.            .unroll(xi)
7.            .unroll(yi,2);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 5
step =  5
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      4.7617 ms
↪  1.1597 ms
```

unroll 展开后，性能从 7.3ms 优化到 4.8ms。

STEP 6 前面的分块成 16 x 8 的小 kernel, 这一步先划分成 16 x 32 的分块，然后把每个分块再分成 16 x 8 的子分块。我们把最外层的两层循环合并到一层，并对这一层进行并行化。这一步计算描述多了一个 prod 函数来定义子分块的计算，prod 函数的计算公式和总的 gemm 是一样的，我们通过 compute_at 指定在 yi 维度之下计算 prod，则 prod 计算的是 16x8 的小 kernel, 大致逻辑如下：



总的代码如下：

```
1.          Func prod;
2.          prod(x, y) += A(k, y) * B(x, k);
3.          gemm(x, y) = prod(x, y);
4.
5.          gemm.tile(x, y, xi, yi, 16, 32)
6.              .fuse(x, y, xy).parallel(xy)
7.              .split(yi, yi, yii, 4)
8.              .vectorize(xi, 8)
9.              .unroll(xi)
10.             .unroll(yii);
11.
12.         prod.compute_at(gemm, yi)
13.             .vectorize(x, 8).unroll(y);
14.
15.         prod.update()
16.             .reorder(x, y, k)
17.             .vectorize(x, 8)
18.             .unroll(x)
19.             .unroll(y)
```

(下页继续)

(续上页)

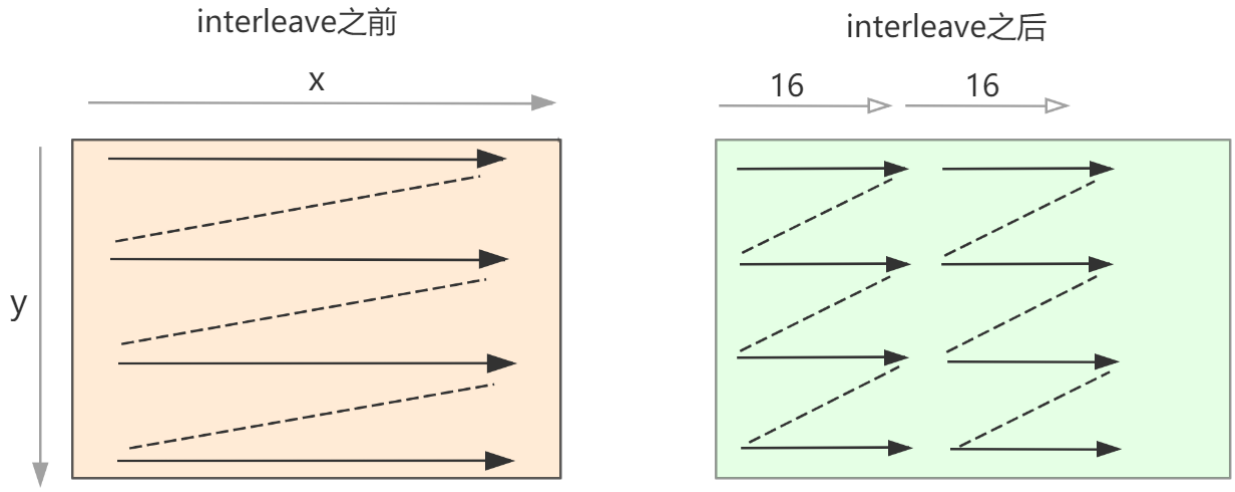
```
20.             .unroll(k, 2);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 6
step =  6
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      3.1824 ms
↪  1.1373 ms
```

这一步距离 STEP1 性能已经优化了近 80 倍了，性能越来越接近 OpenBlas 了。

STEP 7 这一步添加的操作是对矩阵 B 进行数据重排，使得在计算小 kernel 16x8 时，内存读取更顺畅。因为小 kernel 的 x 维度是按照 16 划分的，因此重排数据 B 的 x 维度也是按照 16 重排。



总的代码如下：

```
1.          Func B_interleave("B"), Bs("Bs");
2.          Bs(x, y, xo) = B(xo * 16 + x, y);
3.          B_interleave(x, y) = Bs(x % 16, y, x / 16);
4.
5.          Func prod;
6.          prod(x, y) += A(k, y) * B_interleave(x, k);
7.          gemm(x, y) = prod(x, y);
8.
9.          gemm.tile(x, y, xi, yi, 16, 32)
10.              .fuse(x, y, xy).parallel(xy)
11.              .split(yi, yi, yii, 4)
12.              .vectorize(xi, 8)
13.              .unroll(xi)
14.              .unroll(yii);
15.
```

(下页继续)

(续上页)

```
16.         prod.compute_at(gemm, yi)
17.             .vectorize(x, 8).unroll(y);
18.
19.         prod.update()
20.             .reorder(x, y, k)
21.             .vectorize(x, 8)
22.             .unroll(x)
23.             .unroll(y)
24.             .unroll(k, 2);
25.         Bs.compute_root()
26.             .split(y, yo, yi, 16)
27.             .reorder(x, yi, xo, yo)
28.             .unroll(x)
29.             .vectorize(yi).parallel(yo, 4);
```

执行结果

```
root@bd3faab0f079:/AutoKernel/doc/tutorials/data# ./06_build.sh 7
step =  7
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      1.1957 ms
↪  1.1425 ms
```

至此，我们的每一步调优策略始终都围绕两条优化宗旨“优化内存访问”，“提高并行性”展开优化，到最后性能已经与 OpenBlAS 差不多了，距离 STEP1 已经加速了 200+ 倍了。

CHAPTER 10

AutoKernel: 带你回顾神经网络编译器

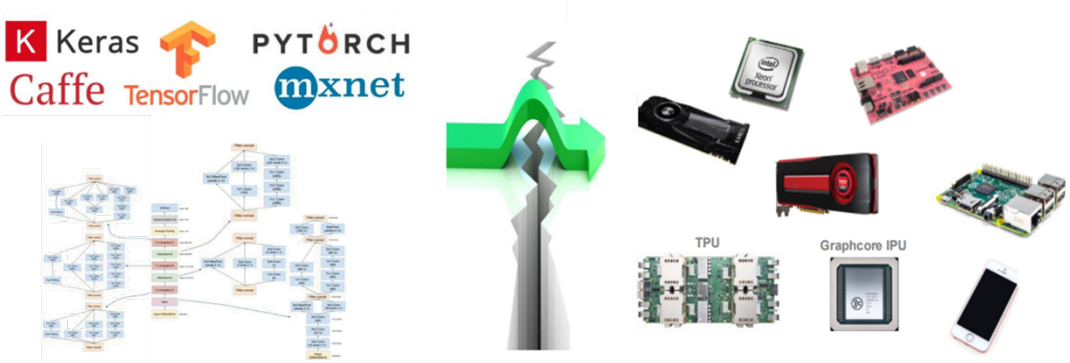
10.1 一、神经网络编译器概览

近年来，以机器学习、深度学习为核心的 AI 技术得到迅猛发展，深度神经网络在各行各业得到广泛应用：

1. CV（计算机视觉）：目标检测，场景识别，图像分割等。
2. 智慧语音：语音识别，声纹识别等。
3. NLP（自然语言处理）：自动搜索引擎，对话服务机器人，文本分类，智能翻译等。
4. 科学研究：应用于物理、生物、医学等多研究领域。高能粒子对撞分类，宇宙天体图数据分析，星系形状建模，生物结构的蛋白质折叠预测，精准医疗与疾病预测。

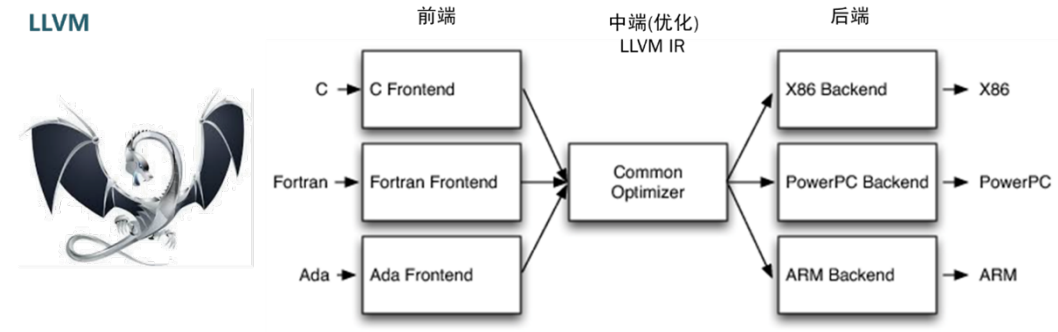
这些应用催生更多的新模型出现：CNN, RNN, LSTM, GAN, GNN，也催生着如 Tensorflow, Pytorch, Mxnet, Caffe 等深度学习框架出现。目前训练框架开始收敛，逐步形成了 PyTorch 引领学术界，TensorFlow 主导工业界的一个双雄局面。

但是，深度学习算法要实现落地应用，必须被部署到硬件上，例如 Google 的 TPU、华为麒麟 NPU，以及其他在 FPGA 上的架构创新。

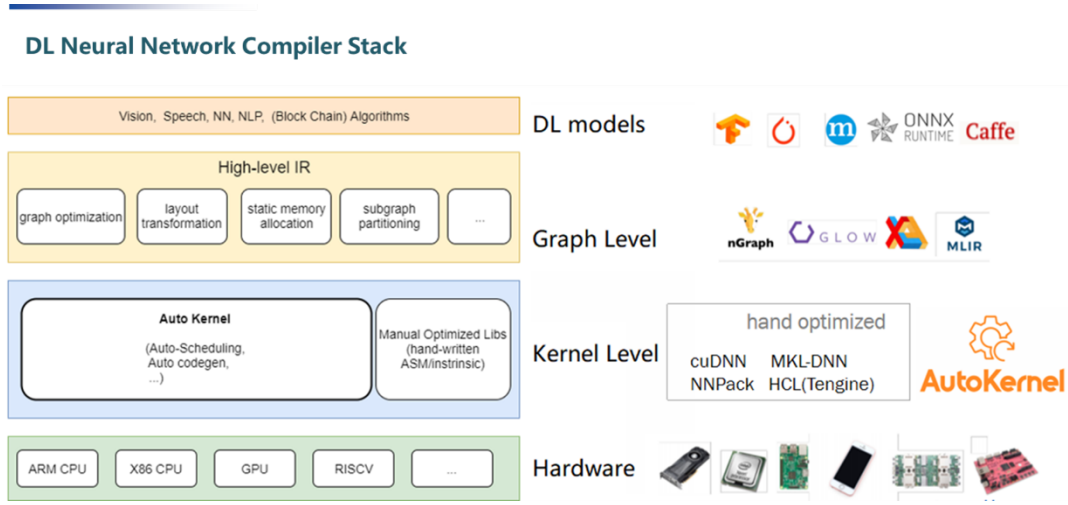


这些各训练框架训练出来的模型要如何部署到不同的终端硬件呢，这就需要深度学习神经网络编译器来解决。

在神经网络编译器之前，我们使用的是传统编译器。**传统编译器：**以 LLVM（low level virtual machine）为例，其输入是高级编程语言源码，输出是机器码，由一系列模块化的编译器组件和工具链组成。LLVM 通过模块分为前端，中端（优化）和后端三部分。每当出现新的编程语言，只需要开发相应的前端，将编程语言转换成 LLVM 的中间表示；类似地，出现新的硬件架构，只需要开发相应的后端，对接上 LLVM 的中间表示。模块化的划分，避免了因编程语言和 CPU 架构的翻新而引发的编译器适配性问题，大大简化了编译器的开发工作。



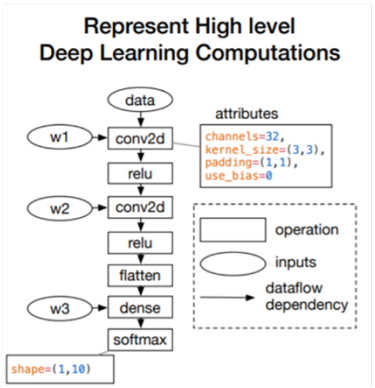
神经网络编译器：其输入是深度学习训练框架训练出来的模型定义文件，输出是能够在不同硬件高效执行的代码。

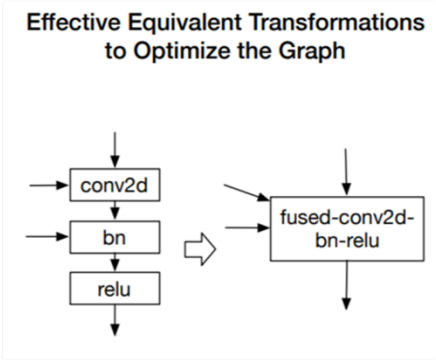


从上至下由四个层级组成：

1. 最上层对接各个深度学习训练框架训练出来的算法模型（Tensorflow, Caffe, Pytorch, Mxnet 等）。
2. 图层级（High-level IR）：神经网络的结构可以表示成计算图，图层级的操作则是对计算图进行一些和具体硬件和框架无关的操作，比如算子融合，内存分配优化，数据类型和数据维度的推导等。

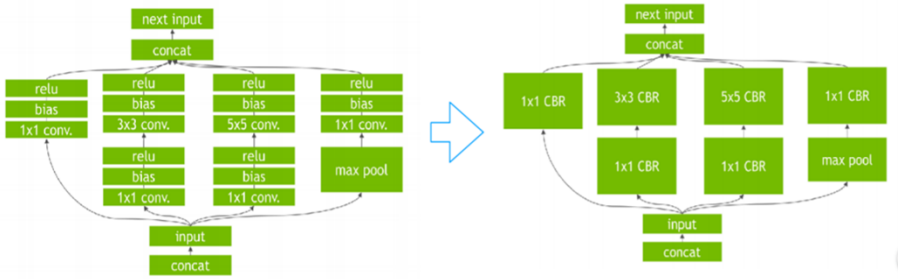
Compute Graph as IR





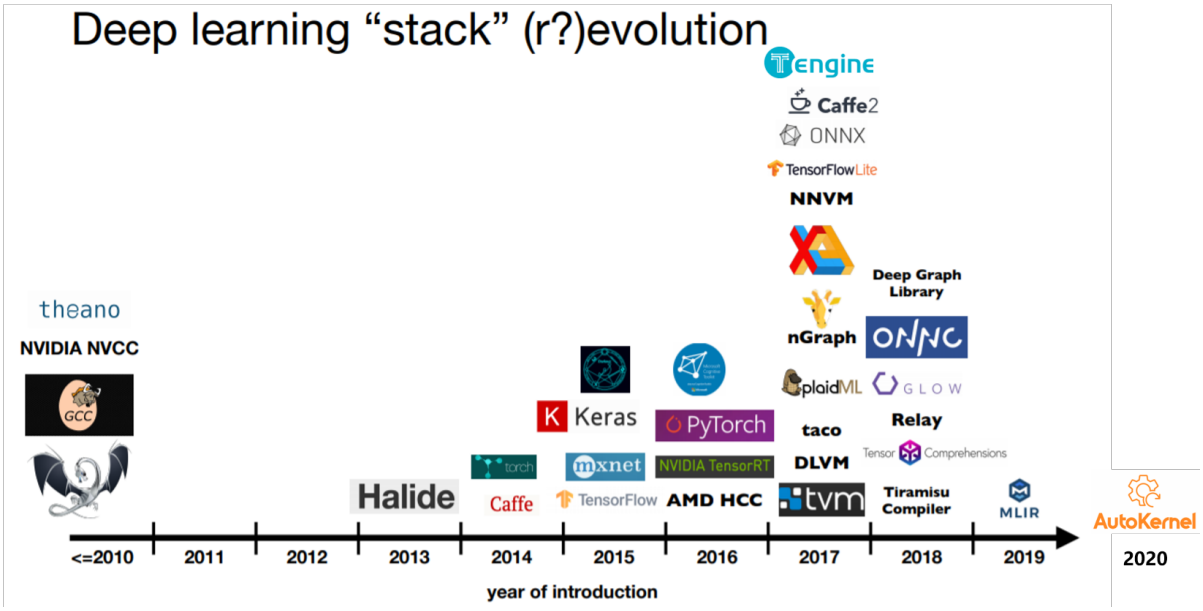
我们可通过算子融合的方式，避免中间数据频繁的在寄存器和内存直接来回读写，从而提升整体推理性能。

Nvidia TensorRT Fuse Conv-Relu



Nividia 通过把 *conv, bn, relu* 这三个算子融合成一个算子 *fuse- CBR.* 实现了三倍的推理性能提升。

1. 算子层级（operator level/kernel level）算子层级主要是张量计算。为了实现这些计算在硬件上高效实现，发挥芯片的性能，通常硬件芯片配有专门优化的算子计算库，如 Intel 的 MKL, Nvidia 的 CuDNN, TensorRT。这个层级需要支持每个硬件后端的每个算子实现。
2. 各硬件后端：GPU, ARM CPU, X86 CPU, NPU 等。



自深度学习编译器的概念提出以来，各类编译器层出不穷。

10.2 二、TVM 的前世今生

在编译器快速发展的浪潮中，较为突出的便是 TVM（Tensor Virtual Machine)。

TVM 最早提出是 2017 年，是深度学习系统的编译器堆栈。

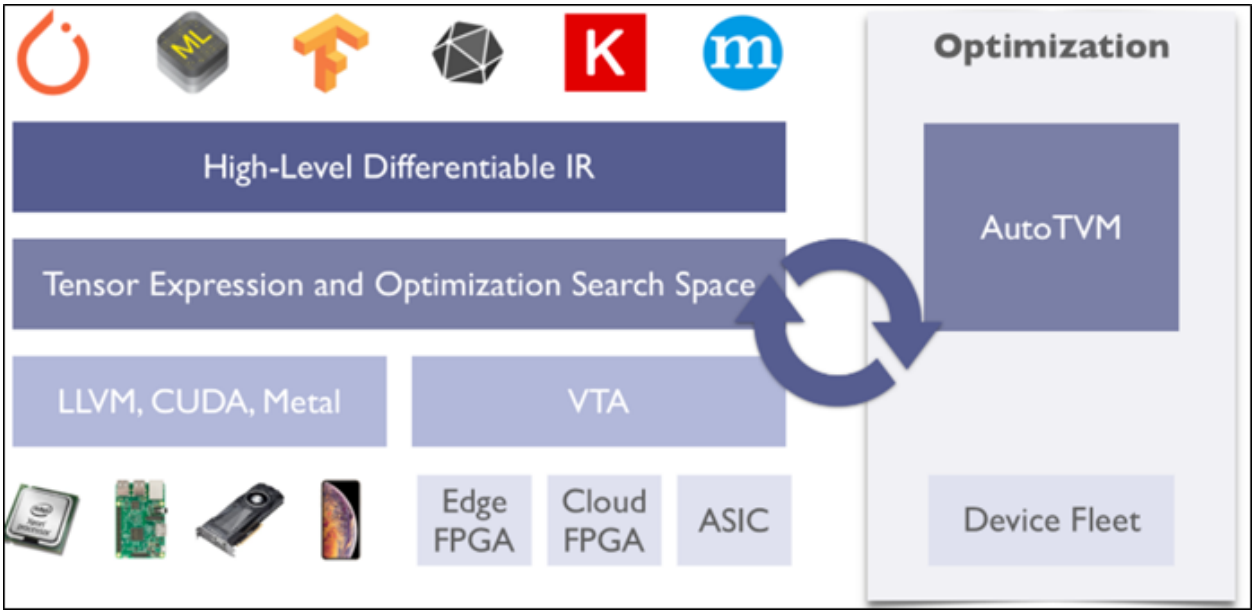
第一代 TVM 的设计借鉴了借鉴传统编译器框架 LLVM 的设计思路，设计抽象出中间表示层，不同的模型只需要开发相应的前端接口，不同的后端只需要开发相应的后端接口。TVM 全称为 Tensor Virtual Machine，属于算子层级，主要用于张量计算，提供独立于硬件的底层计算中间表示，采用各种方式（循环分块，缓存优化等）对相应的计算进行优化。第一代的图层级表示叫 NNVM（Neural Network Virtual Machine)。NNVM 的设计目标是：将来自不通深度学习框架的计算图转换为统一的计算图中间表示（IR)，对之进行优化。

第一代的静态图存在一定的缺陷：

1. 不能较好支持控制流，如分支跳转，循环等。
2. 不能支持计算图输入形状，取决于输入 tensor 大小的模型，比如 word2vec 等。

虽然 Tensorflow 有如 tf.cond.Tf.while_loop 的控制接口来在某种程度上解决第一个问题，tf.fold 来解决第二个问题，但是这种方式对刚刚接触深度学习框架的小白来说不是特别友好。

后面出现的动态图摒弃了传统的计算图先定义，后执行的方式，采用了计算图在运行时定义的模式，这种计算图就称为动态图。第二代 TVM 的图计算层变为 Relay VM，Relay 和第一代的图计算表示 NNVM 的最主要区别是 Relay IR 除了支持 dataflow（静态图)，能够更好地解决 control flow（动态图)。它不仅是一种计算图的中间表示，也支持自动微分。



总结一下，目前 TVM 的架构是：

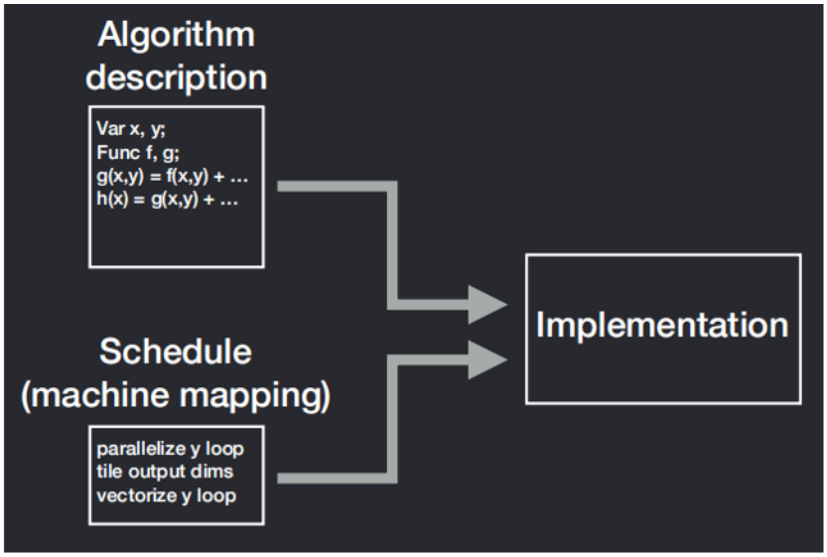
1. 最高层级支持主流的深度学习前端框架，包括 TensorFlow,MXNet,Pytorch 等。
2. Relay IR 支持可微分，该层级进行图融合，数据重排等图优化操作。
3. 基于 tensor 张量化计算图，并根据后端进行硬件原语级优化，autoTVM 根据优化目标探索搜索空间，找到最优解。
4. 后端支持 ARM、CUDA/Metal/OpenCL、加速器 VTA（Versatile Tensor Accelerator）。

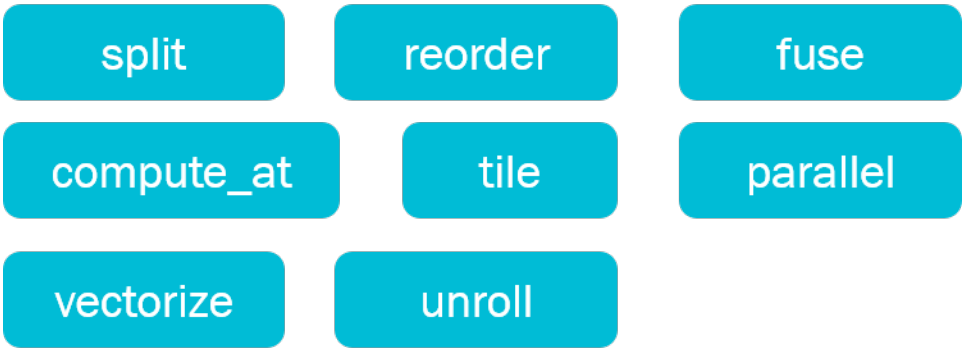
10.3 三、Halide

Halide 于 2012 年提出，主要用于自动优化。其嵌入到 C++ 中，是 MIT 研究人员专门为图像处理设计的一种程序语言。Halide 语言易于编写，语法简单，数据结构清晰，能过自动对代码进行优化，使得程序获得比较好的执行效率。

它设计的核心思想是把算法和调度分离。这样做的好处是，在给定算法的情况下只需要去调整它的 Schedule 调度选择，不用重写算法实现不同的 Schedule。当调整 Schedule、探索设计空间时也不会担心因为重写算法而导致计算的正确性会发生变化。

Algorithm 部分主要是算法描述和计算的数学表达式。Schedule 部分则是告诉机器什么时候分配内存，如何计算（分块计算还是顺序计算）——目前已经提供了一些调度策略。


Algorithm description
Var x, y;
Func f, g;
g(x,y) = f(x,y) + ...
h(x) = g(x,y) + ...
Schedule (machine mapping)
parallelize y loop
tile output dims
vectorize y loop
Implementation

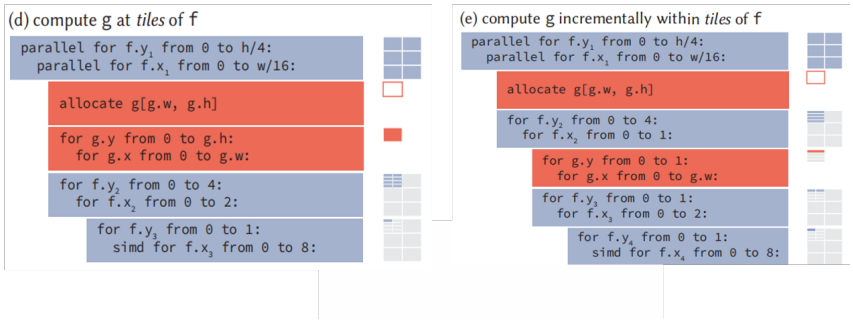

split
reorder
fuse
compute_at
tile
parallel
vectorize
unroll

Call Schedule

```
g(x, y) =  ...;
f(x, y) = g(x, y-1) + g(x, y+1);
```

- g is called by f
- f.tile(x,y, x_1, y_1, x_2, y_2, 16, 4)
  .tile(x_2, y_2,x_3, y_3, 8, 1)

- g.compute_root()
- g.compute_at(f, x_1)
- g.compute_at(f, x_2)
  .store_at(f, x_1)



不同调度策略考虑重复冗余计算和局部性（*locality*）的权衡。

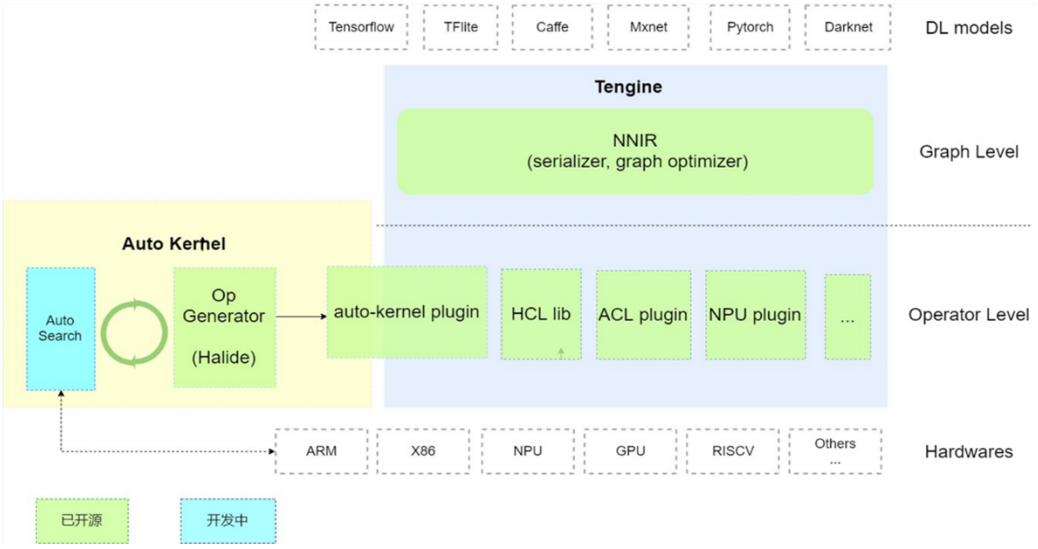
10.4 四、AutoKernel

深度学习模型能否成功在终端落地应用，满足产品需求，一个关键的指标就是神经网络模型的推理性能。目前的高性能算子计算库主要是由高性能计算优化工程师进行手工开发。然而新的算法/硬件的不断涌现，导致了算子层级的优化开发工作量巨大。同时优化代码的工作并不是一件简单的事，它要求工程师既要精通计算机体系架构，又要熟悉算子的计算流程。人才少，需求多，技术门槛高，因此我们认为算子优化自动化是未来的大趋势。而提出 AutoKernel 的初衷便是希望能把这个过程自动化，从小处入手，在算子层级的优化，实现优化代码的自动生成。

AutoKernel 的输入是算子的计算描述（如 Conv、Poll、Fc），输出是经过优化的加速源码。这一工具的开发旨在降低优化工作的门槛，不需要有底层汇编的知识门槛，不用手写优化汇编。可通过直接调用开发的工具包便可生成汇编代码。同时还提供了包含 CPU、GPU 的 docker 环境，无需部署开发环境，只需使用 docker 便可。还可通过提供的插件——plugin，可以把自动生成的算子一键集成到推理框架中——Tengine。

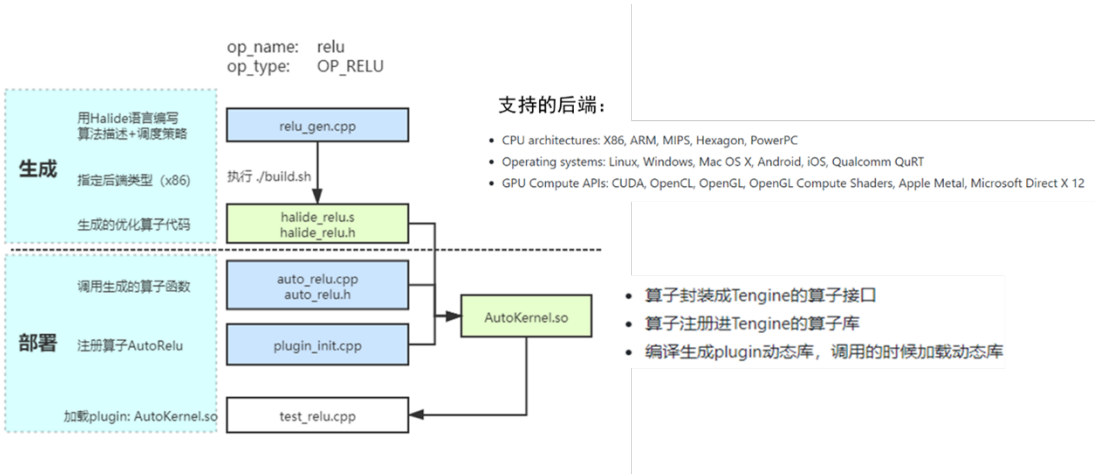
对应地，算子层级的 AutoKernel 则主要分为三个模块，

1. Op Generator：算子生成器，采用了开源的 Hallide。
2. AutoSearch：目前还在开发中，目标是通过机器学习、强化学习常用算法自动搜索出优化策略。
3. AutoKernel Plugin：把生成的自动算子以插件的形式插入到 Tengine 中，和人工定制互为补充。



Tengine 对象层对接了不同的神经网络模型，图层级的 NNIR 包含了模型解析、图层优化，算子层级则包括高性能计算库（HCL lib）。

AutoKernel Plugin 主要分为生成和部署两部分，生成部分用 Hallid 填写算法描述和调度策略，执行时指定后端类型（基本覆盖目前的主流后端）。部署部分则封装为 Tengine 的库，直接调用。



相信随着更多开发者的加入，AutoKernel 社区会有更大的突破与成长，在未来的深度学习编译器领域中，留下浓重的一笔！

CHAPTER 11

Data Transform 开发文档

11.1 背景

Currently, most auto-schedule solutions (Halide, tvm, ansor, flextensor, ···) do auto-tuning to search best schedules based on functions(multi-dimension array) defined on algorithm/compute descriptions.

But the data layout of defined functions(multi-dimension array) are also important for the final performance.

11.2 Key idea to understand why data transform?

- loop transformations are used to improve `temporal locality`
- data layout optimizations are used to improve `spatial locality`.

we will exploiting `spatial locality` for given function data layout with defined computation.

11.3 Auto Data transform

对一个 Pipeline，我们定义了 input 数据排布的 cost，然后自动对 Pipeline 中的每个 input 遍历 data transform，计算其 cost，选择 cost 最小的一种排布策略。

具体的文件说明如下：

11.4 DataTransform.h

- 源码文件: AutoKernel/AutoSearch/src/common/DataTransform.h

11.4.1 DataTransformer 类

```
template <typename T>
class DataTransformer : public IRVisitor
```

DataTransformer：用来在表达式中递归得搜索到目标函数，并对其按照模板 T 执行 data transform。它继承自 IRVisitor，可以使用 accept 和 visit 来遍历表达式的每一部分，从而找到我们希望执行 data transform 的目标函数。

```
std::string target_;
Func replace_func_;
T op_;
```

三个私有成员变量，target_ 表示要进行数据排布的函数名，replace_func_ 是替换后的新的函数，op_ 是要执行的数据排布操作。

在 DataTransformer 中，我们仅考虑了四则运算，即加，减，乘，除和除余，这里以加法运算做说明：

```
void visit(const Add *add) override {
    IRVisitor::visit(add);
    Add *node = const_cast<Add *>(reinterpret_cast<const Add *>(add)) ;
    if (const Halide::Internal::Call *v= node->a.as<Halide::Internal::Call>()) {
        if (v->name==target_)
        {
            auto expr = v->args;
            op_(expr);
            node->a = replace_func_(expr);
        }
    }
    if (const Halide::Internal::Call *v= node->b.as<Halide::Internal::Call>()) {
        if (v->name==target_)
        {
            auto expr = v->args;
            op_(expr);
            node->b = replace_func_(expr);
        }
    }
}
```

当加法运算的两端中的一个是函数 (Call) 时，检查其函数名是否为目标名，如果是目标名，则获取其函数的

运算参数，并对所有运算参数执行 op_ 的 data transform 操作，然后将这个 Call 函数重新用 replace_func_ 定义。

11.4.2 data_transform 函数

```
template<typename T>
void data_transform_impl(Function f,Func target)
```

功能：对以 Function f 作为结尾的 Pipeline，找到其中的 target 函数，并对之执行模板 T 操作，T 是三种 data transform 的一种。

11.4.3 data_transform_impl 函数

```
void data_transform(std::vector<Function> &outputs,Func target,DataTransformMethod␣
↪method)
```

功能：data_transform_impl 的封装函数，根据提供的 method 来调用不同的 data_transform_impl 函数。DataTransformMethod 是枚举类，定义在 DataOP.h 中。

11.4.4 deep_copy 函数

```
std::vector<Function> deep_copy(std::vector<Function> &inp)
```

功能：对输入的 inp 进行深拷贝并返回拷贝后的结果。

11.4.5 auto_data_transform 函数

```
void auto_data_transform(std::vector<Function> &outputs)
```

功能：对以 outputs 为结尾的 pipeline，找到他们的输入，并执行最佳的 data transform 操作。auto_data_transform 是所有优化器调用数据排布的接口函数。

实现细节：对 outputs 进行深拷贝得初始的 best_output，然后对 best_output 搜索每一个 input，并对每个 input 尝试三种 data transform，在调用 compute_layout_cost_impl 函数计算数据排布的 cost，选择最小的一个作为新的 best_output，重复此过程直到无法再得到一个更低的 cost 为止。于是我们就获得了最终的 best_output。

11.4.6 compute_order_cost 函数

```
double compute_order_cost(const Definition  &def,std::string function_name,const␣
↪std::string &target, std::map<std::string,std::pair<int,int> >& bounds)
```

功能：计算一个定义（Definition）关于 target（函数名称）的 reorder_cost 值，reorder_cost 是数据排布 cost 的一种，另一种为 use_distance_cost, 具体参见 cost_model.md。bounds 是计算需要用到的变量的范围映射表。

11.4.7 compute_use_distance 函数

```
double compute_use_distance(const Definition &def,const std::string &target,std::map
↪<std::string,std::pair<int,int> >& bounds)
```

功能：计算一个定义（Definition）关于 target 的 use_distance_cost,bounds 是计算需要用到的变量的范围映射表。

11.4.8 compute_layout_cost_impl 函数

```
double compute_layout_cost_impl(std::vector<Function> &outputs)
```

功能：计算以 outputs 结尾的 pipeline 的输入数据排布的 cost，这是计算 cost 的接口函数，通过调用此函数可以得到最终的数据排布的 cost。

11.5 DataOP.h

DataOP 头文件定义了进行数据重排布算子的类，它们都继承自 DataOP 类，并且需要重载 DataOP 中的函数。

11.5.1 DataTransformMethod

```
enum class DataTransformMethod{
    REORDER,
    INTERLEAVE,
    SPLITY
};
```

这是一个枚举类，定义了 data transform 的三种方法

11.5.2 DataOP 类

```
class DataOP
{
public:
    // operate the vector<Expr>
    virtual void operator()(std::vector<Expr> &args){}
    virtual void operator()(Func &lfunc,Func &rfunc){}
    virtual std::string name(){return "$NULL";}
};
```

DataOP 需要重载三个函数，两个括号运算符重载和一个命名函数, 命名规则是 $+ 算子名。

```
virtual void operator()(std::vector<Expr> &args){}
```

这个重载是用于在定义中将目标函数的参数进行重新排布的，以 matmul.cpp 为例：

```
func(x,y,b)=input_a(k,y,b)*input_b(x,k,b)
```

执行 reorder 操作得到：

```
Br(x,y,b) = input_b(y,x,b)
func(x,y,b)=input_a(k,y,b)*Br(k,x,b)
```

operator()(std::vector &args) 用于在 func 表达式中将 input_b(x,k,b) 替换为 Br(k,x,b)

```
virtual void operator()(Func &lfunc,Func &rfunc){}
```

operator()(Func &lfunc,Func &rfunc) 用于构造 Br(x,y,b)=input_b(y,x,b) 表达式。

```
virtual std::string name(){return "$NULL";}
```

该函数用于提供一个函数名。

11.5.3 ReorderOP

ReorderOP 实现的是将一个 input 第一维和第二维数据排布交换的操作。其实现方法是定义一个新的函数，令 reorder_func(x,y,⋯) = input(y,x,⋯)。

然后在每个涉及 input 的表达式中，都将 input(x,y,..) 替换为 reorder_func(y,x,⋯)

11.5.4 InterleaveOP

InterleaveOP 实现的是将一个 input 第一维每 8 个数据划分。其实现方法是定义一个新的函数，令 interleave_func(x,y,xo,⋯) = input(xo*8+x,y,⋯)。

然后在每个涉及 input 的表达式中，都将 input(x,y,..) 替换为 interleave_func(x%8,y,x/8,⋯)

11.5.5 SplitYOP

SplitYOP 实现的是将一个 input 第二维每 8 个数据划分。其实现方法是定义一个新的函数，令 splity_func(x,y,yo,⋯) = input(x,yo*8+y,⋯)。

然后在每个涉及 input 的表达式中，都将 input(x,y,..) 替换为 splity_func(x,y%8,y/8,⋯)

11.6 AutoDataTransform 的 cost model

根据实验，几种数据重新排布的方法可以提高优化后的执行速度，为了表征这些排布的效果，设计了 cost 函数来计算他们。我们定义了两类 cost 分别为 reorder cost 和 use_distance cost

11.6.1 reorder cost

对于一个这样的表达式：

```
func(x,y,z,b) = input(y,x,z)*b
```

在 x 和 y 维度上 input 和 func 的数据读取顺序发生了不一致，这种不一致在 y 很大时会导致 cache miss 率增大，其 cache miss 的程度可以用来衡量这种数据排布的 cost。

由于当 y 较大时，对于此运算，发生 cache miss 的次数为循环次数（即 cache 几乎没有命中过），故次数为 xyz*b。

对于上述表达式，如果我们增加一个定义：

```
Reorder(x,y,z) = input(y,x,z)
func(x,y,z,b) = Reorder(x,y,z)*b
```

可以看到，结果一样，func(x,y,z,b) 这个表达式没有发生 cache miss，只有上面的 Reorder(x,y,z) 发生了 cache miss，其 miss 次数为 xyz，可以看到，miss 次数减少了 b 倍。所以我们定义 reorder cost 为发生数据读取顺序不一致的表达式的数据总量。

需要注意的是，这个 cost 并不是线性的，因为它需要 x,y 维度的大小来计算，当 x*y 在 cache 大小内时，此时 cache 中并不会完全 miss，而对于 cpu 来说，有 L1，L2,L3 三层 cache，每层大小不同，因此这个 cost 函数是阶梯上升的。具体参数设置可以从 DataTransform.h 中的 compute_order_cost 函数中获取。

11.6.2 use_distacne cost

use_distance 比 reorder 更加难理解一些，它是基于如下考虑的，当下游的一个表达式需要的数据大于一个 cache 大小时，我们可以对数据进行划分，使之按照一个 cache 大小的划分成块，这样的处理会使数据形成流水线，上游生产的数据顺序与下游需求的数据顺序一致，那么当下游读取数据时就可以不停的从 cache 中读取，而不需要经过内存-cache 访存了。

具体看如下实例：

```
Rdom k(0,K);
func(x,y,b) += input_A(k,y,b)*input_b(x,k,b);
```

在这个例子中，input_b 的数据需求始终是第二维优先的，我们可以通过 reroder 使其变成第一维，也可以使用划分的操作。我们对 x 进行划分, 形成新的表达式：

```
Rdom k(0,K);
Bi(x,y,xo,b) = input_b(xo*16+x,y,b)
func(x,y,b) += input_A(k,y,b)*Bi(x%16,k,x/16,b);
```

新增加的 Bi 表达式的数据产生顺序是 x,y,xo,b，其中 x 的范围为 0~16，因此 x*y 的大小缩小了，这部分数据可以在一个 cache 中放下，而在 func 表达式中，我们需求的数据顺序对应到 Bi 中是 y,x,xo,b。

在没有划分前，input_b 的数据准备顺序和数据读取顺序之间相差了 M 倍（input_b 的第一维的大小），而划分后，input_b 的数据准备顺序和数据读取顺序之间仅相差了 16，这大大提高了数据的读取效率。

use_distance cost 的定义：表达式相邻两次计算时读取的输入数据在内存上的距离。

注意到如果原始的尺寸中 x*y 就很小，那么 use_distance cost 也没有意义，因为所有的数据都放在 cache 中了。所以我们的 use_distance cost 是一个分段函数。