AutoKernel

OPEN AI LAB

Jun 04, 2021

INTRODUCTION

CHAPTER

ONE

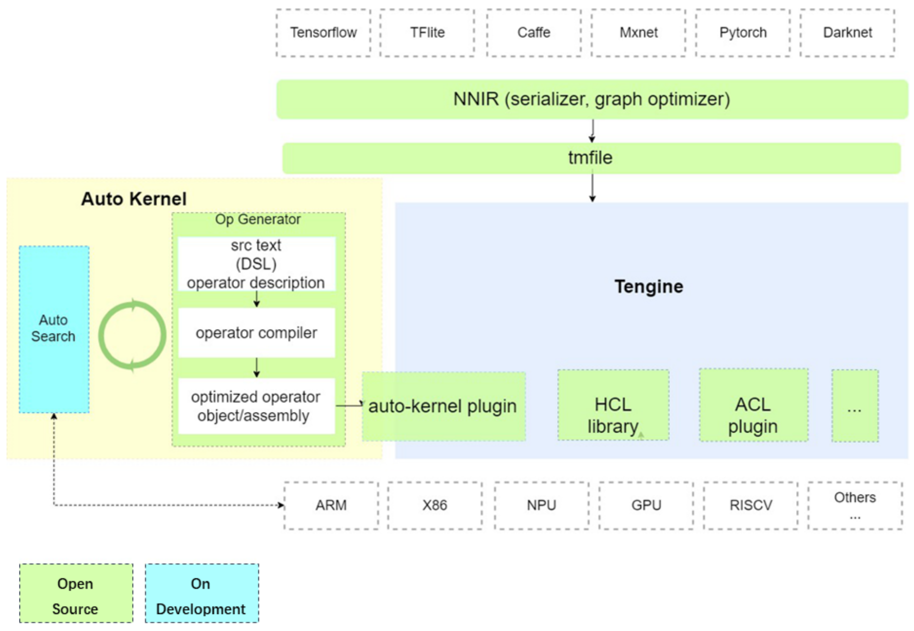
AUTOKERNEL

1.1 Introduction

Neural networks are now used in a wide variety of applications. Efficient execution of Neural networks on various devices plays a critical role for these applications. Facing the rapid evolution of deep learning algorithms, there are limited qualified programmers to write hand optimized low-level kernels on different hardware platforms. Using automatic optimization tools to generate high-performance implementations become a promising solution.

AutoKernel began as a research project at OPEN AI LAB. The project is now open source. AutoKernel is an operator optimzation tools for automatically generating high-performance low-level codes for diverse hardware backends. It aims to accelerate the development of high performance operators on various hardware including specialized accelerators.

1.2 AutoKernel Architecture



AutoKernel consists of three modules

Operator Generator

This module uses the open source project Halide. Halide is a domain specific language (DSL), embedded in C++, designed to make it easier to write high-performance image processing code on modern machines. Halide seperates the algorithm description from its schedule. The input of this module is the algorithm description of operator, and the output is compiled optimized assembly code/object file for corresponding back-ends.

AutoSearch

AutoSearch is an automatic module for searching optimized schedules for halide operators, using multiple optimization algorithms (greedy algorithm, reinforce learning, marchine learning, . . .). It supports searching optimized schedules on both CPU and GPU, and generate code files running on different platforms (x86 or arm).

AutoKernel Plugin

AutoKernel Plugin realizes one-click integration of auto-generated optimized operator codes into Deep learning inference framework Tengine, without modifying the core code base of Tengine. AutoKernel plugin "inserts" auto-generated operator implementations into the deployment framework Tengine, enabling the framework to mixed-use both "autokernels" and "hand-written kernels".

1.3 Supported backends

Following targets have been tested:

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

More target archs/features:

- arch:arm, hexagon, mips, powerpc, riscv, wasm, x86.
- bits32, 64
- osandroid, ios, linux, windows. . .
- features: avx, avx2, avx512, cl_half, cuda, opencl. . .

CHAPTER

TWO

INSTALL FROM SOURCE

1. Compile & install Halide

```
git clone https://github.com/halide/Halide
cd Halide && mkdir build && cd build
export HALIDE_DIR=/path/halide-install # you can revise the path according to␣
↪specific circumstances
cmake .. -DTARGET_WEBASSEMBLY=OFF -DCMAKE_INSTALL_PREFIX=${HALIDE_DIR}
make -j `nproc` && make install # compile & install
```

2. Compile Tengine

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

3. Compile AutoKernel

```
git clone https://github.com/OAID/AutoKernel.git
cd         AutoKernel/autokernel_plugin
find . -name "*.sh" | xargs chmod +x  #Add executable permissions to sh script
```

`Edit the installation path of Halide library in ./scripts/generate.sh`

```
# Modify the HALIDE_DIR path in the first line to the installation path in the␣
↪first step
export HALIDE_DIR=/path/halide-install
```

Edit Tengine's installation path, open the `TENGINE_ROOT` in `autokernel_plugin/CMakeLists.txt`

```
set(TENGINE_ROOT /path/Tengine) # Modify the directory where the Tengine project is␣
↪located
set(TENGINE_DIR /path/tengine-install) # Modify to the installation path in the␣
↪second step
```

Compile AutoKernel

```
./scripts/generate.sh  #Automatically generate operator assembly file
mkdir build && cd build
cmake .. && make -j `nproc`
```

Add the path where libautokernel.so is located to LD_LIBRARY_PATH, and execute it in the `AutoKernel/autokernel_plugin/build/` directory

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

Execute the test program in the AutoKernel/autokernel_plugin directory

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

Execution results see as follows

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

CHAPTER THREE

BASED ON DOCKER

AutoKernel provides docker imagewhere Halide and Tengine have been installed:

3.1 Docker Image:

- cpu

  ```
  docker pull openailab/autokernel
  ```

- cuda:

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

  [NOTE]: To use cuda image, you need to use nvidia-docker, detalis see here nvidia-docker install-guide.

- opencl:

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

3.2 Dockerfile

Detailed Dockerfile see

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

3.3 AutoKernel tutorials for installation

1. Pull the image (it may take a while, please wait patiently, depending on the network speed, it may take 10-20mins)

   ```
   docker pull openailab/autokernel
   ```

2. Create a container and enter the development environment

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

3. Halide, Tengine have been installed in docker

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

4. AutoKernel

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

Add executable permissions to sh scripts and automatically generate operator assembly files

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

Compile

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

Run test

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

3.4 Halide in Docker

Halide has been installed in Autokernel docker, and the Python API has been configured.

Halide related files are all in`/workspace/Halide/` directorythe install files of Halide are all in`/workspace/Halide/halide-build` directory

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

- Halide-related files are in `/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
```

- Compiled Halide library are in`/workspace/Halide/halide-build/src` diectory, where we can find `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
```

- Run Halide

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

Execution Results

```
Success!
```

- Run the Python interface of HalideFirst look up the system path of Python

```
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']
```

We can see that the Python system path already has Halide's compiled python package path`'/workspace/Halide/halide-build/python_bindings/src'`

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

`import halide`

3.5 Tengine in Docker

There has been Tengine installed in Autokernel dockerrelated files are all in`/workspace/Tengine/`directory.

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

- Tengine related files are all in`/workspace/Tengine/build/install/include`

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

tengine_c_api.h
tengine_cpp_api.h
```

- Compiled Tengine library are in `/workspace/Tengine/build/install/lib`directory, where we can find`libtengine-lite.so`

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

libtengine-lite.so
```

- Run Tengine

  This example run the performance benchmark of each network model of Tengine on the target computer.

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

Execution results

```
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
        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

FOUR

AUTOSEARCH

AutoSearch is an automatic module for automatically searching optimized schedules for halide operators. It supports optimized schedules on both CPU and GPU, and generate code files running on different platforms (x86 or arm). We also offer a method to optimize the input data layout (called data transform).

We provide:

- one-click compilation to compute time of manual schedule
- one-click autotune to generate schedule
- evaluation template to verify correctness

4.1 Install

Before using AutoSearch, please install Halide. You can directly using Autokernel docker with Halide installed in `/workspace/Halide/`.

```
docker pull openailab/autokernel
```

install AutoSearch with the following commands:

```
export HALIDE_HOME=<path>/<to>/Halide
cd <path>/<to>/AutoKernel/AutoSearch
mkdir build & cd build
cmake ..
make -16
```

4.2 Toolkit usefule parameters

- `--gen`: the generator file
- `--target`: backend target, we have tested the following targets:
  - x86-64-linux
  - x86-64-linux-opencl
  - x86-64-linux-cuda
  - arm-64-linux
  - arm-64-linux-opencl

More:

- arch:arm, hexagon, mips, powerpc, riscv, wasm, x86.
- bits32, 64
- osandroid, ios, linux, windows...
- features: avx, avx2, avx512, cl_half, cuda, opencl...

- `-autotune`: using autotunung to generate schedules
  - CUDA: using sioutas2020 autoschedule
  - OpenCL: using li2018 autoschedule
  - CPU: using adams2019 autoschedule, with 2 more arguments:
    * num_tune_loops: number of iterations to retrain cost model
    * batch_size: for each iteration, the number of samples to get featurization and runtime-cost
- `-compute_time`:
  - compute_samples: repeat to take minimum time
  - num_iterators: repeat to take average time

  the pseudocode is:

```
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

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

- `data transform`: using automatic data transforming to find a better input data layout

4.3 Evaluate Manual Schedule Time

All following tests are with default shape config M=N=K=512 for matmul op.

1. CPU: x86-64-linux

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

Tested on Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz, it gets:

```
autokernel time:        1.79585 ms
```

1. Nvida GPU: CUDA

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

Tested on GeForce GTX 1080 Ti, it gets:

```
autokernel time:        0.6114 ms
```

1. ARM CPU

this depends on GNU C++ cross compile toolchain `aarch64-linux-gnu-g++`

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

copy the executable file to RK3399, and run

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

Tested on RK3399, it gets:

```
autokernel time:        245.7393 ms
```

1. ARM Mali GPU: Opencl

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

Tested on RK3399 Mali-T860 , it gets:

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

4.4 AutoTune: Evaluate Generated Schedule Time

1. CPU: x86-64-linux

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

Tested on Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz, it gets:

```
autokernel time:        0.880885 ms
```

1. Nvida GPU: CUDA

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

(continues on next page)

(continued from previous page)

```
```

Tested on GeForce GTX 1080 Ti, it gets:

```
autokernel time:        0.604405 ms
```

1. ARM CPU

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

Tested on RK3399 , it gets:

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

The generated arm-cpu schedules are not satisfying, since this workflow is based on baseline.weights, not retrained on real arm cpu.

1. ARM Mali GPU

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

Tested on RK3399 Mali-T860 , it gets:

```
autokernel time:         87.249 ms
```

4.5 DataTransform

During the autotune process, you can use automatic data transforming to find a better input data layout.

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

4.6 Correctness verification template

Demo code is provided in `toolkit/template/demo_eval.cpp` to demostrate how to compile generate codes and verify correctness.

In the example of matmul of size 512, generated codes are in directory `toolkit/template/samples`, we need to include generated header and call the function

```
#include "demo_1_512_512_512.h"

matmul(Halide_A,Halide_B,Halide_C);
```

compile

```
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
```

If implementation consistent with `ref_func`, it will get

```
Correctness check passed!
```

CHAPTER

FIVE

AUTOKERNEL PLUGIN

Using autokernel docker

```
# Pull the image (may take a while, please be patient)
docker pull openailab/autokernel
# Start the container and enter the development environment
docker run -it openailab/autokernel /bin/bash
```

Installed Halide and Tengine are provided in docker

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

Clone AutoKernel

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

Let's see the directory of `autokernel_plugin/src/`

```
autokernel_plugin/src/
|-- CMakeLists.txt
|-- direct_conv
|   |-- build.sh
|   |-- direct_conv.cpp
|   |-- direct_conv.h
|   |-- direct_conv_gen.cc
|-- im2col_conv
|   |-- build.sh
|   |-- im2col_conv.cpp
|   |-- im2col_conv.h
|   `-- im2col_conv_gen.cc
`-- plugin_init.cpp
```

We can see that there are two folders in the `src` directory, each of which contains:

- `xxx_gen.cc`, use Halide language operator description (algorithm) and scheduling strategy (schedule)
- `build.sh` is used to compile xxx_gen
- `xxx.h` and xxx.cpp are operator implementations encapsulated by the Tengine operator interface

One-click operator assembly code generation

```
cd AutoKernel/autokernel_plugin
bash ./scripts/generate.sh
```

When finishing this step, we can see that there are two more automatically generated files in the original directory:

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

Next, use the automatically generated file to register Autokernel into tengine, and compile libAutoKernel.so with one click

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

The generated library is in`/workspace/AutoKernel/autokernel_plugin/build/src/libautokernel.so` To run the test, call `load_tengine_plugin()` in the test code:

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

The results of the classification network are as follows:

```
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
```

As you can see, the output shows that `AutoKernel plugin` is called.

CHAPTER

SIX

TENGINE: QUICK START

Tengine is a lightweight deep neural network inference framework. This document will take the classification model (Squezenet model) as an example (based on the x86 Linux platform) to take you quickly to get started with Tengine.

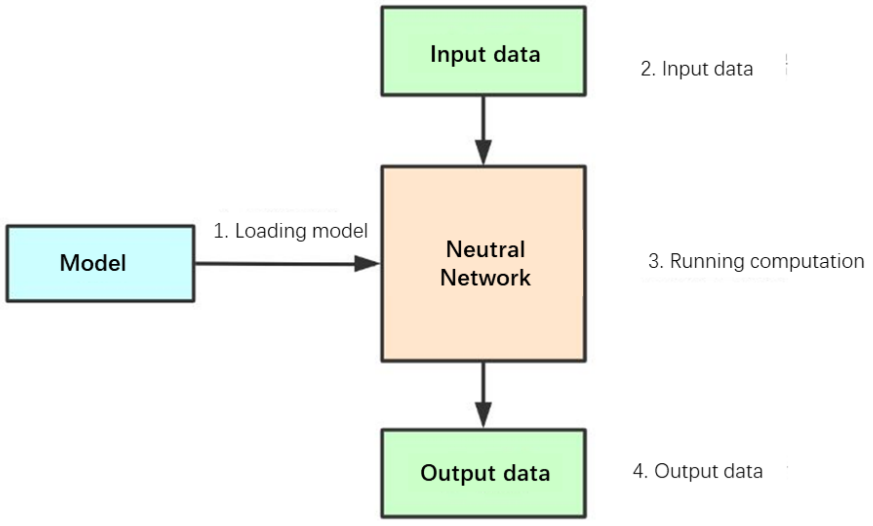
6.1 Deep learning neural network calculation process

Concept

`-Neural network`: Neural network can be understood as a graph. A graph is composed of multiple operator nodes. These nodes can be Convolution or Pooling. , Fully connected operator (Fc), etc.

`-Neural network model`: The neural network model is trained by the deep learning training framework (Tensorflow, Caffe, Pytorch, Mxnet, etc.). The model contains two pieces of information:-The computational graph structure of the neural network-The weight data of the operator

Calculation process



inference

1. Load the model: get the neural network structure and weight data
2. Prepare input data, feed input data

3. Perform model inference calculations
4. Get output data

6.2 Tengine Squeezenet Example

This example will follow the neural network inference calculation process to demonstrate how to perform the inference calculation of the Squeezenet classification network in Tengine

1. Loading model

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

   `model_file`Is a model file in tengine format"squeezenet.tmfile"

2. Prepare input data, feed input data

   ```
   /* 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. Perform model inference calculations

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

4. Get output data

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

- codes
  - The full codes see here: data/02_tengine_tutorial.cpp.
  - Tool functions see here: tengine_operations.h
- Compile

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

- Execute

```
cd tutorials/data/build

#Download model & figure
wget https://github.com/OAID/TengineModels/raw/main/images/cat.jpg .
```

(continues on next page)

(continued from previous page)

```
wget https://github.com/OAID/TengineModels/raw/main/tmfiles/squeezenet.tmfile .
./02_tengine_tutorial
```

Get results

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

This is a classification network, 1000 classes, index from 0 to 999, each category has a probability score, the running result prints out the top 5 probability scores and index.

6.3 More Tengine Examples

More Tengine application examples are in Tengine/examples:

- Classification
- Face key point detection
- ssd target detection
- retinaface face detection
- yolact instance segmentation
- yolov3 target detection
- yolov4-tiny target detection
- openpose human body gesture recognition
- crnn Chinese character recognition

CHAPTER

SEVEN

HALIDE

7.1 Introduction to Halide

Halide is an open source, domain specific language (DSL) designed to make it easier to write high-performance image and array processing code on modern machines.

7.1.1 Embedded in C++

Halide is a domain specific language (DSL) embedded in C++. This means you write C++ code that builds an in-memory representation of a Halide pipeline using Halide's C++ API.

7.1.2 Decouple algorithm from schedule

Halide separates the program into two conceptual parts:

- The algorithm – Defines what is computed at each pixel
- The schedule – Defines how the computation should be organized

Through this design, developers can focus on how to implement their algorithm in isolation from how it should be best scheduled for execution, to optimize for locality, parallelism, and ultimately performance.

7.1.3 Compile

Halide offers two modes to perform this compilation:

- Ahead of Time (AOT): Halide is first compiled to a header and object file and then statically linked by the developer into their app.
- Just in Time (JIT): performs compilation at runtime (i.e., while the app is running) but requires that the Halide compiler be hosted on the target platform.

7.1.4 Target

Halide currently targets:

- 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

7.1.5 Resources

- github: https://github.com/halide/Halide
- For API doc, see http://halide-lang.org/docs

Ref:

1. https://halide-lang.org/
2. https://developer.qualcomm.com/blog/optimizing-image-processing-algorithms-halide

7.2 Algorithm

Every Halide program has two parts, an algorithm and a schedule. The algorithm defines what is computed at each pixel. Here we present the constructs that Halide provides to write an algorithm.

7.2.1 Func

Func: a pure function, defining image values over some domain. In the following example, f is a Func that is used to compute an image where every pixel is the sum of its x and y coordinates.

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

A Func can be an input to another Func. In this case, an image processing pipeline can be constructed by cascading multiple Funcs (each of which is a pipeline stage) in producer-consumer relationships.

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

f, g and h are all Funcs, each representing a stage of this pipeline. h is the final stage of the pipeline.

7.2.2 Var

Var: a free variable in the domain of a function.

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

In this example, x and y have no meaning on their own. However, when used with the blur_x and input Funcs, they signify the two dimensions of these Funcs.

7.2.3 Expr

An Expr in Halide is composed of Funcs, Vars, and other Exprs, for example: of Exprs in Halide.

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

In these examples, x and y are Vars, e is an Expr, and Output is a Func.

7.3 Schedule

The schedule specifies how the algorithm computation is to be structured/organized.

There are mainly two kinds of schedules:

- schedules within stages
- schedules across stages

7.3.1 Schedules within stages: Domain Order

Schedules within one stage define the order of iteration domain of function, using a traditional set of loop transformation concepts, applied to the dimensions of the function. In the Halide's schedule language, these choices are applied as annotations on each function.

- Dimensions can be traversed sequentially (the default) or in parallel `f.parallel(y)`.
- Constant-size dimensions can be unrolled `f.unroll(x)` or vectorized `f.vectorize(x)`
- Dimensions can be reordered (e.g., from column- to row-major: `f.reorder(y, x)`).
- Dimensions can be split by some factor, creating two new dimensions: an outer dimension, and an inner dimension (`f.split(x,outer,inner,factor)`.
- Dimensions can be fused, turning two dimensions into one (f.fuse(x, y)).

7.3.2 Schedules across stages: Call Schedule

In addition to the order of evaluation within the domain of each function, the schedule also specifies the granularity with which to interleave the computation and storage of each function with the domain of the functions that call it. We call these choices the call schedule. In the language of Halide's schedules, we control the call schedule of each function with annotations as followings:

7.4 Halide: Quick start

Before we dive into Halide, let's experience Halide's magic.

Enter AutoKernel's docker, there is already Halide's python environment in docker, just run

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

Here are the output

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

The above script performs a simple function calculation`func[x,y] = x + 10*y` Compare the running time of the two functions-func_origin: default function-func_parallel: Added a scheduling strategy of Halide: `func.parallel(y,4)`, parallelizes the y dimension, and the parallelism is 4

As you can see, the second function takes a quarter of the time of the first function.

No need to optimize the assembly knowledge at the bottom, just add a line of code, you can get a better optimization effect.

7.4.1 Language basic of Halide

If you want to call Halide's scheduling strategy, you must first master the basic Halide language, and use Halide language to describe the calculation of operators. The following simple functions are used to demonstrate the basic data structure of the Halide language.

`-Variable Var`: It can be understood as an independent variable of a function. For example, to describe the pixels of an image, two variables x and y are needed to describe the coordinates of the w dimension and the h dimension.-`Function Func`: Similar to mathematical functions, it defines a calculation process. The complex calculation process can be divided into multiple small functions to achieve.

Example one

The function calculation formula of this example is: `func(x,y)= 10*y + x`Use Halide language to describe this function:

- Python:

```python
import halide as hl

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

- C++

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

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

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

Func's realize will calculate the value of the function in the domain and return the numerical result. After calling realize, the function is jit-compile. Before that, only the calculation process of the function is defined.

Look up the calculation result.

- 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));
        }
    }
```

The process of function is

```
                     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
```

Full codes see here data/03_halide_basic.py we can run directly

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

In addition, you can call `func.trace_stores()` to track the value of the function

Example Two

This example demonstrates how to feed input data and remove output data.Full codes see heredata/03_halide_feed_data.py

The function of this example:

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

A is the input data, you can define Halide.Buffer, and then feed the numpy array data into the buffer

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

Function B

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

There are several ways to obtain output data

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

We can run the full codes

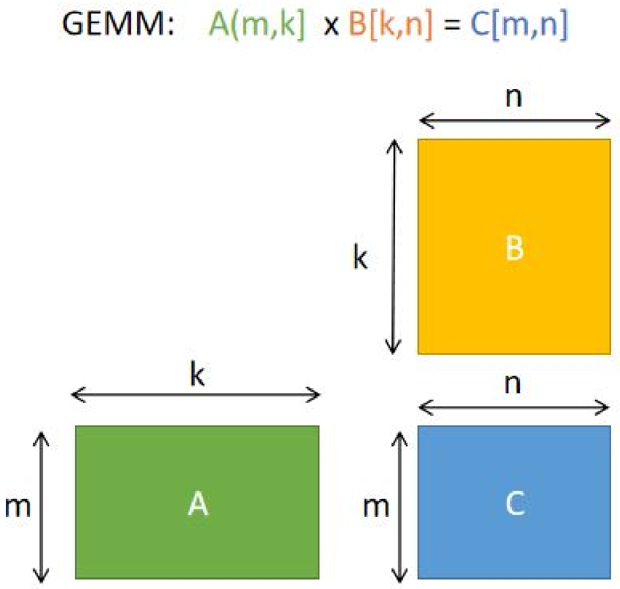
```
python data/03_halide_feed_data.py
```

CHAPTER

EIGHT

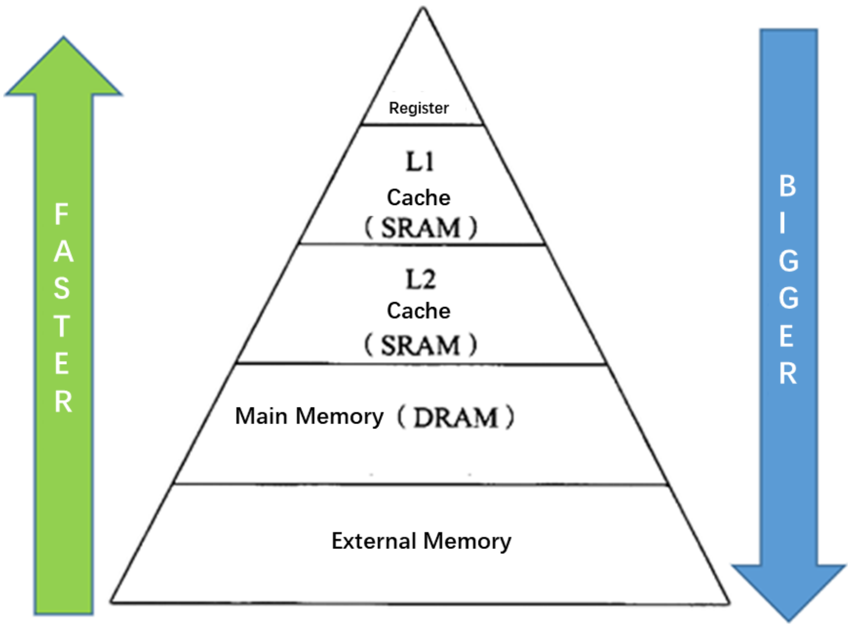
OPTIMIZATION TUTORIAL OF GEMM ON X86

Here, this tutorial will lead you to optimize the matrix multiplication GEMM step by step. There is no need to manually code, compile complicated and lengthy low-level assembly code, but only need a dozen lines of concise scheduling code.



gemm.jpg

Two optimization goals Before explaining the optimization steps in detail, let's talk about the essence of optimization. When we were talking about "optimization", what did the bottom layer of the computer do? What is the "bottleneck" of optimization? Why can performance be improved through a wave of "optimization operations"? How does Halide used by AutoKernel achieve automatic optimization?To answer these questions, we need to understand the basic architecture of the hardware and understand how the hardware works, so that when implementing algorithms in software, we should consider using some of the characteristics of the hardware as much as possible to achieve efficient and extreme optimization.



memory.png

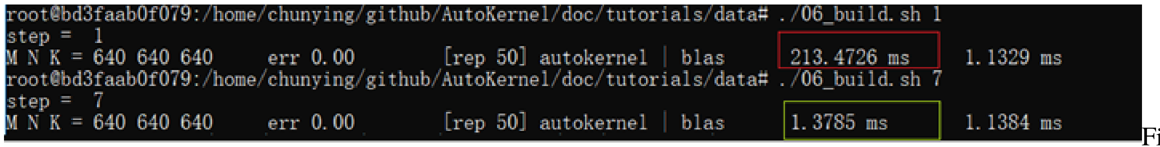
Operating environment preparation AutoKernel provides a docker image, with the running environment being configured, and user can run the demo code directly after entering the docker

```
# Pull mirror
docker pull openailab/autokernel
# Start the container and enter the development environment
docker run -it openailab/autokernel /bin/bash
# access codes
git clone https://github.com/OAID/AutoKernel.git
cd AutoKernel/doc/tutorials/data/
```

The build.sh in the directory is the execution script of the demo. To run, user need to specify the optimization step 'step'. The optional step is from 1 to 7, where step=1 is not optimized by default, and step=7 is the most optimized.**Effect of optimization**

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

The following figure shows the optimization effect on a computer with Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz. There is no need to manually write code, no need to write complicated and lengthy low-level assembly code, only a dozen lines of concise scheduling code , The performance can be optimized by 200+ times.



```
root@bd3faab0f079:/home/chunying/github/AutoKernel/doc/tutorials/data# ./06_build.sh 1
step =  1
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      213.4726 ms     1.1329 ms
root@bd3faab0f079:/home/chunying/github/AutoKernel/doc/tutorials/data# ./06_build.sh 7
step =  7
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas      1.3785 ms       1.1384 ms
```

Figure1.png

Detailed optimizatio stepsSTEP 1 The first step is without any optimization. Use Halide language to directly describe the calculation process of GEMM.

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

Calculate the matrix multiplication of M=N=K=640. The first parameter of the running script specifies step=1. The time-consuming results as follows

```
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 In this step, we use tiled tiles. The purpose of Tiling is to make full use of the cache. If the original loop is large, the tiles are changed to small pieces of data to calculate, so that the data calculated each time can stay in the cache more comfortably, without repeated eviction (repeat adding and deleting data in the cache) . The reorder operation is performed after the block is divided, the order of the two nested loops is exchanged, and the purpose is to make the innermost memory access friendly. We divide the x and y dimensions into 16x8 small blocks to calculate

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

execution result

```
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
```

Performance has been optimized from 240ms to 82ms, an increase of nearly 3 times.

STEP 3 We add 'vectorize' based on the previous step. Vectorization is to convert several scalar calculations (scale) into a vector calculation (vector), making full use of SIMD vector instructions. Most modern CPUs support SIMD (Single Instruction Multiple Data). In the same CPU cycle, SIMD can execute the same operation/instruction on multiple values at the same time.

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

execution result

```
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
```

The performance was optimized from 82ms to 27ms, which was accelerated by nearly 3 times. It can be found that around the two optimization purposes mentioned above: optimizing memory access and improving parallelism, from step1 to step3, the performance has been improved by nearly 9 times.

STEP 4 The scheduling strategy adds parallelization on the basis of step3. Parallelizing a loop is to divide each iteration of the loop into multiple threads or processors for simultaneous processing. Each thread processes through the code segment (loop body), but processes different data.

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

execution result

```
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
```

After adding parallelization, build.sh specifies four threads by default, and the performance is directly increased by nearly 4 times, from 27ms to 7.3ms.

STEP 5 The scheduling strategy adds unroll expansion on the basis of the previous step. If the statements in the loop body have no data-related dependencies, loop unrolling can increase the chance of concurrent execution, make full use of registers, and reduce the number of times each operation memory is loaded and saved during the loop.

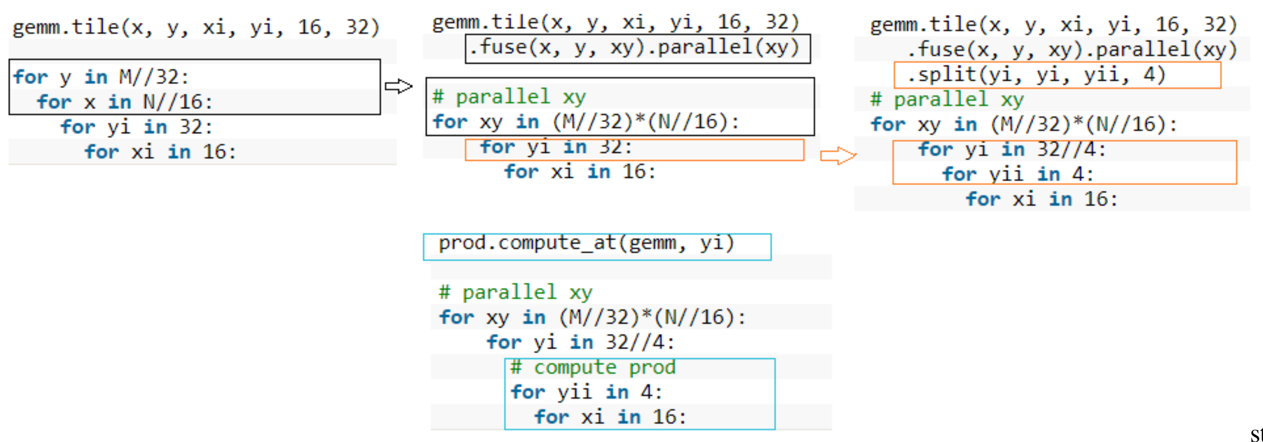
```
        gemm.update()
            .tile(x, y, xo, yo, xi, yi, 16, 8)
            .reorder(xi, yi, k, xo, yo)
            .vectorize(xi, 8)
            .parallel(yo)
            .unroll(xi)
            .unroll(yi,2);
```

execution result

```
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
```

After unroll is expanded, the performance is optimized from 7.3ms to 4.8ms.

STEP 6 The previous block is divided into 16 x 8 small kernels. This step is first divided into 16 x 32 blocks, and then each block is divided into 16 x 8 sub-blocks. We merge the two outermost loops into one layer and parallelize this layer. The calculation description in this step adds a prod function to define the calculation of sub-blocks. The calculation formula of the prod function is the same as the total gemm. We use compute_at to specify the calculation of prod under the one dimension, and the calculation of prod is 16x8. kernel, the general logic is as follows:



```
gemm.tile(x, y, xi, yi, 16, 32)

for y in M//32:
  for x in N//16:
    for yi in 32:
      for xi in 16:
```

```
gemm.tile(x, y, xi, yi, 16, 32)
  .fuse(x, y, xy).parallel(xy)

# parallel xy
for xy in (M//32)*(N//16):
  for yi in 32:
    for xi in 16:
```

```
gemm.tile(x, y, xi, yi, 16, 32)
   .fuse(x, y, xy).parallel(xy)
   .split(yi, yi, yii, 4)
# parallel xy
for xy in (M//32)*(N//16):
  for yi in 32//4:
    for yii in 4:
      for xi in 16:
```

```
prod.compute_at(gemm, yi)

# parallel xy
for xy in (M//32)*(N//16):
    for yi in 32//4:
      # compute prod
      for yii in 4:
        for xi in 16:
```

step6.png

Codes are listed below

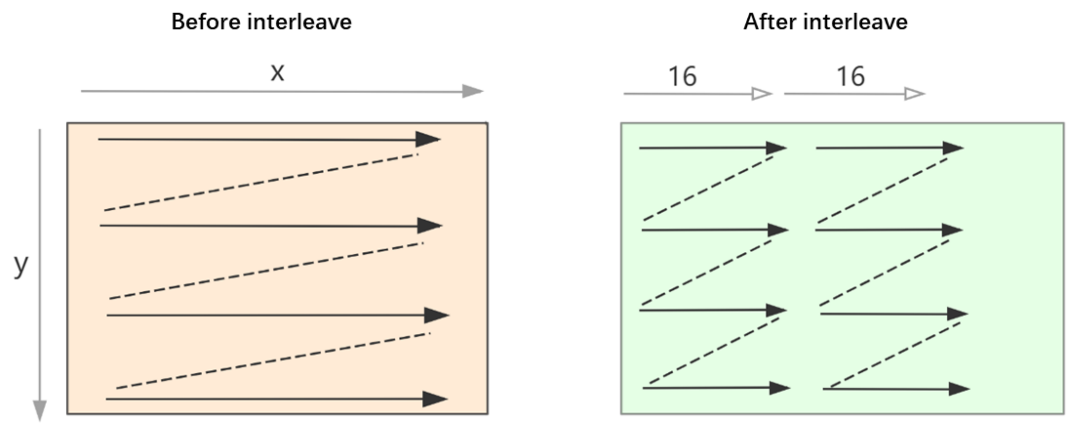
```
        Func prod;
        prod(x, y) += A(k, y) * B(x, k);
        gemm(x, y) = prod(x, y);

        gemm.tile(x, y, xi, yi, 16, 32)
            .fuse(x, y, xy).parallel(xy)
            .split(yi, yi, yii, 4)
            .vectorize(xi, 8)
            .unroll(xi)
             .unroll(yii);

         prod.compute_at(gemm, yi)
             .vectorize(x, 8).unroll(y);

         prod.update()
             .reorder(x, y, k)
             .vectorize(x, 8)
             .unroll(x)
             .unroll(y)
             .unroll(k, 2);
```

execution result

```
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
```

The performance of this step has been optimized by nearly 80 times from STEP1, and the performance is getting closer and closer to OpenBlas.

STEP 7 The operation added in this step is to rearrange the data of matrix B to make the memory read smoother when calculating the small kernel 16x8. Because the x dimension of the small kernel is divided according to 16, the x dimension of rearranged data B is also rearranged according to 16.



interleave.png

Codes are listed below

```
       Func B_interleave("B"), Bs("Bs");
       Bs(x, y, xo) = B(xo * 16 + x, y);
       B_interleave(x, y) = Bs(x % 16, y, x / 16);

       Func prod;
       prod(x, y) += A(k, y) * B_interleave(x, k);
       gemm(x, y) = prod(x, y);

       gemm.tile(x, y, xi, yi, 16, 32)
            .fuse(x, y, xy).parallel(xy)
            .split(yi, yi, yii, 4)
            .vectorize(xi, 8)
            .unroll(xi)
            .unroll(yii);

        prod.compute_at(gemm, yi)
            .vectorize(x, 8).unroll(y);

        prod.update()
            .reorder(x, y, k)
            .vectorize(x, 8)
            .unroll(x)
            .unroll(y)
            .unroll(k, 2);
        Bs.compute_root()
            .split(y, yo, yi, 16)
            .reorder(x, yi, xo, yo)
            .unroll(x)
            .vectorize(yi).parallel(yo, 4);
```

execution result

```
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
```

So far, every step of our tuning strategy has always been optimized around the two optimization purposes "optimizing memory access" and "improving parallelism". In the end, the performance is almost the same as OpenBlAS, and the distance from STEP1 has been accelerated by 200+ times.

CHAPTER

NINE

THE POWER OF AUTOKERNEL: OPTIMIZE THE PERFORMANCE OF GEMM BY 200 TIMES!

With the rapid development of AI technology, deep learning has been widely used in various fields. Whether the deep learning model can be successfully applied to the terminal and meet the product needs, a key indicator is the reasoning performance of the neural network model. As a result, a large wave of algorithm engineers transferred to operator optimization engineers for the deployment of algorithms. However, optimizing code is not a simple task. It requires engineers not only to be proficient in computer architecture, but also to be familiar with the calculation process of the algorithm. Therefore, a little experienced deep learning reasoning optimization engineer has become the wanted targets of various companies. With few in engineers and many in demand, operator optimization and automation is the general trend in the future.

automatic.png

Recently, AutoKernel, an automatic operator optimization tool dedicated to lowering the optimization threshold and improving the efficiency of optimization development, is officially open source.

Operator
Conv
Pool
Fc
Conv
AutoKernel
Optimized source codes
</>

autokernel.png

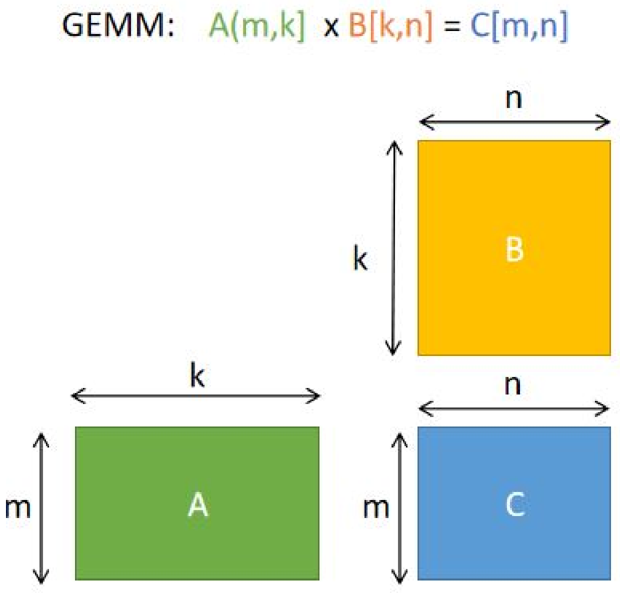
9.1 Features of AutoKernel

- Low threshold: no knowledge threshold for bottom-level optimization compilation.
- Simple and easy to use: Providing a docker environment, no need to install the environment, the plugin is integrated into the inference framework Tengine with one click.
- High efficiency: no need to write optimized assembly, one-click to generate optimized code and one click to deploy.

AutoKernel uses Halide, an automatic code generation project widely used in the industry, to automatically generate the underlying code by inputting calculation descriptions and scheduling strategies. AutoKernel supports one-click deployment of the generated automatic optimization operator to the inference framework Tengine in the form of a plugin.

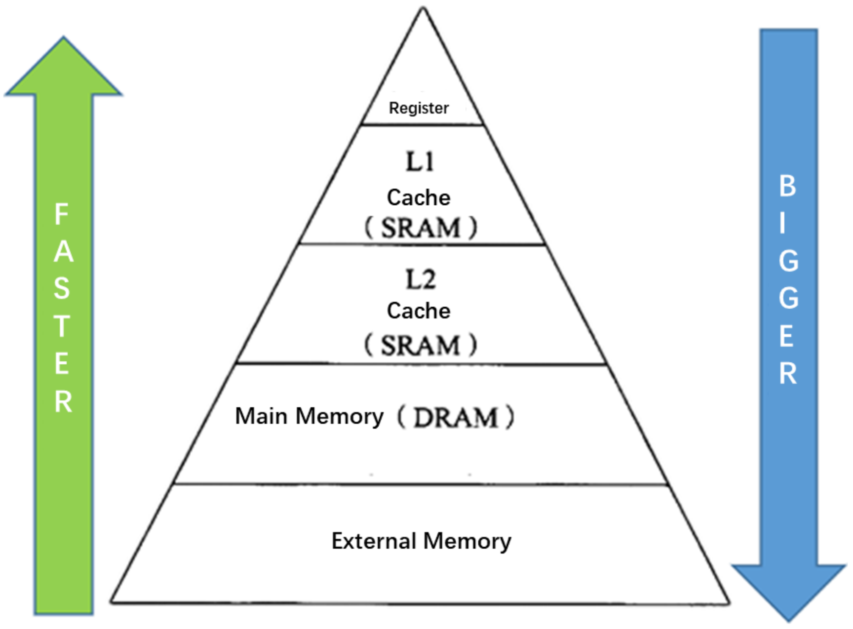
9.2 Optimization tutorial of GEMM

Here, this tutorial will lead you to optimize the matrix multiplication GEMM step by step. There is no need to manually code, compile complicated and lengthy low-level assembly code, but only need a dozen lines of concise scheduling code.



gemm.jpg

Two optimization goals Before explaining the optimization steps in detail, let's talk about the essence of optimization. When we were talking about "optimization", what did the bottom layer of the computer do? What is the "bottleneck" of optimization? Why can performance be improved through a wave of "optimization operations"? How does Halide used by AutoKernel achieve automatic optimization?To answer these questions, we need to understand the basic architecture of the hardware and understand how the hardware works, so that when implementing algorithms in software, we should consider using some of the characteristics of the hardware as much as possible to achieve efficient and extreme optimization.



memory.png

The figure listed above is a typical memory processor hierarchy: the main memory has a large capacity, with a slow access speed, and the register and cache read speed is fast, but the capacity is limited. At the register level, the CPU can access them within one clock cycle. If the CPU accesses the external DDR, the delay is very large, about 200 clock cycles. If the CPU accesses the cache, it usually takes 6 to 12 cycles. Therefore, a very important optimization goal is: **Optimize memory access**, make full use of registers and caches to store data.

The second optimization goal is to **improve parallelism**: make full use of SIMD for instruction vectorization and multi-core parallelism. Most modern CPUs support SIMD (Single Instruction Multiple Data). In the same CPU cycle, SIMD can execute the same operation/instruction on multiple values at the same time. If we vectorize on 4 data points and calculate 4 data at a time, we can theoretically achieve 4 times speedup.

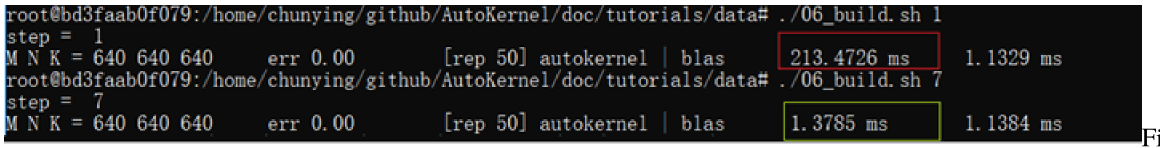
Operating environment preparation AutoKernel provides a docker image, with the running environment being configured, and user can run the demo code directly after entering the docker

```
# Pull mirror
docker pull openailab/autokernel
# Start the container and enter the development environment
docker run -it openailab/autokernel /bin/bash
# access codes
git clone https://github.com/OAID/AutoKernel.git
cd AutoKernel/doc/tutorials/data/
```

The build.sh in the directory is the execution script of the demo. To run, user need to specify the optimization step 'step'. The optional step is from 1 to 7, where step=1 is not optimized by default, and step=7 is the most optimized.**Effect of optimization**

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

The following figure shows the optimization effect on a computer with Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz. There is no need to manually write code, no need to write complicated and lengthy low-level assembly code, only a dozen lines of concise scheduling code , The performance can be optimized by 200+ times.



```
root@bd3faab0f079:/home/chunying/github/AutoKernel/doc/tutorials/data# ./06_build.sh 1
step =  1
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas     213.4726 ms     1.1329 ms
root@bd3faab0f079:/home/chunying/github/AutoKernel/doc/tutorials/data# ./06_build.sh 7
step =  7
M N K = 640 640 640     err 0.00        [rep 50] autokernel | blas     1.3785 ms       1.1384 ms
```

Figure1.png

Detailed optimizatio stepsSTEP 1 The first step is without any optimization. Use Halide language to directly describe the calculation process of GEMM.

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

Calculate the matrix multiplication of M=N=K=640. The first parameter of the running script specifies step=1. The time-consuming results are as follows

```
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 In this step, we use tiled tiles. The purpose of Tiling is to make full use of the cache. If the original loop is large, the tiles are changed to small pieces of data to calculate, so that the data calculated each time can stay in the cache more comfortably, without repeated eviction (repeat adding and deleting data in the cache) . The reorder operation is performed after the block is divided, the order of the two nested loops is exchanged, and the purpose is to make the innermost memory access friendly. We divide the x and y dimensions into 16x8 small blocks to calculate

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

execution result

```
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
```

Performance has been optimized from 240ms to 82ms, an increase of nearly 3 times.

STEP 3 We add 'vectorize' based on the previous step. Vectorization is to convert several scalar calculations (scale) into a vector calculation (vector), making full use of SIMD vector instructions. Most modern CPUs support SIMD (Single Instruction Multiple Data). In the same CPU cycle, SIMD can execute the same operation/instruction on multiple values at the same time.

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

execution result

```
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
```

The performance was optimized from 82ms to 27ms, which was accelerated by nearly 3 times. It can be found that around the two optimization purposes mentioned above: optimizing memory access and improving parallelism, from step1 to step3, the performance has been improved by nearly 9 times.

STEP 4 The scheduling strategy adds parallelization on the basis of step3. Parallelizing a loop is to divide each iteration of the loop into multiple threads or processors for simultaneous processing. Each thread processes through the code segment (loop body), but processes different data.

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

execution result

```
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
```

After adding parallelization, build.sh specifies four threads by default, and the performance is directly increased by nearly 4 times, from 27ms to 7.3ms.

STEP 5 The scheduling strategy adds unroll expansion on the basis of the previous step. If the statements in the loop body have no data-related dependencies, loop unrolling can increase the chance of concurrent execution, make full use of registers, and reduce the number of times each operation memory is loaded and saved during the loop.

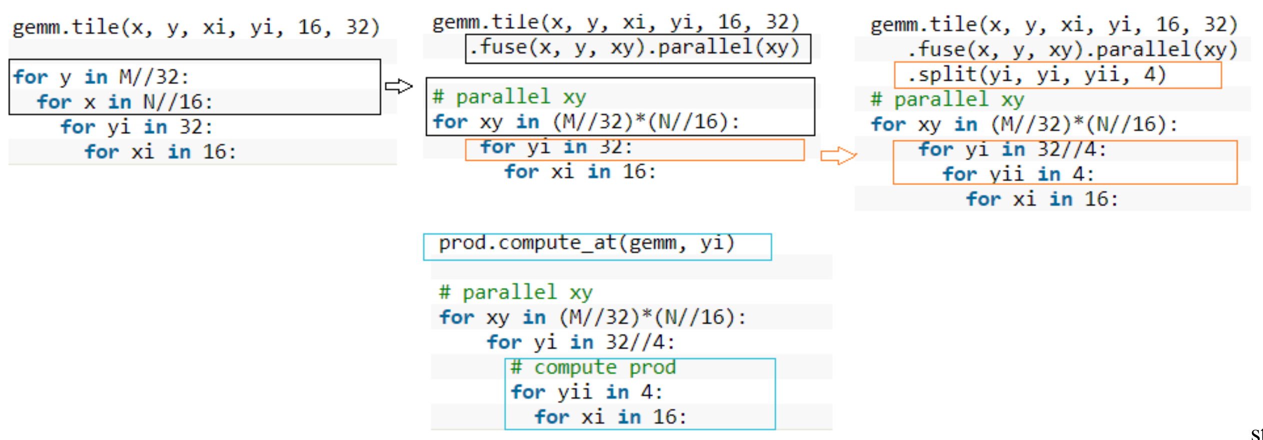
```
        gemm.update()
            .tile(x, y, xo, yo, xi, yi, 16, 8)
            .reorder(xi, yi, k, xo, yo)
            .vectorize(xi, 8)
            .parallel(yo)
            .unroll(xi)
            .unroll(yi,2);
```

execution result

```
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
```

After unroll is expanded, the performance is optimized from 7.3ms to 4.8ms.

STEP 6 The previous block is divided into 16 x 8 small kernels. This step is first divided into 16 x 32 blocks, and then each block is divided into 16 x 8 sub-blocks. We merge the two outermost loops into one layer and parallelize this layer. The calculation description in this step adds a prod function to define the calculation of sub-blocks. The calculation formula of the prod function is the same as the total gemm. We use compute_at to specify the calculation of prod under the one dimension, and the calculation of prod is 16x8. kernel, the general logic is as follows:



```
gemm.tile(x, y, xi, yi, 16, 32)

for y in M//32:
  for x in N//16:
    for yi in 32:
      for xi in 16:
```

```
gemm.tile(x, y, xi, yi, 16, 32)
  .fuse(x, y, xy).parallel(xy)

# parallel xy
for xy in (M//32)*(N//16):
    for yi in 32:
      for xi in 16:
```

```
gemm.tile(x, y, xi, yi, 16, 32)
    .fuse(x, y, xy).parallel(xy)
    .split(yi, yi, yii, 4)
# parallel xy
for xy in (M//32)*(N//16):
    for yi in 32//4:
      for yii in 4:
        for xi in 16:
```

```
 prod.compute_at(gemm, yi)

 # parallel xy
 for xy in (M//32)*(N//16):
     for yi in 32//4:
       # compute prod
       for yii in 4:
         for xi in 16:
```

step6.png

Codes are listed below

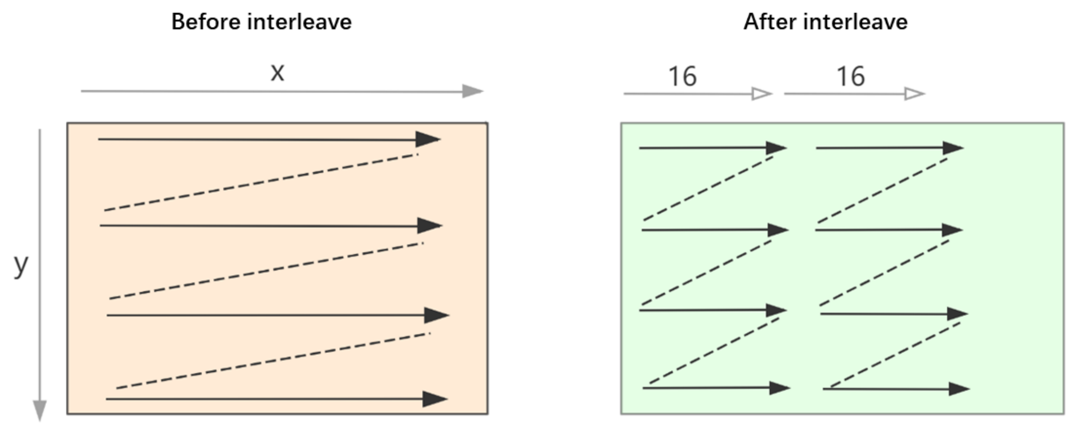
```
Func prod;
prod(x, y) += A(k, y) * B(x, k);
gemm(x, y) = prod(x, y);

gemm.tile(x, y, xi, yi, 16, 32)
    .fuse(x, y, xy).parallel(xy)
    .split(yi, yi, yii, 4)
    .vectorize(xi, 8)
    .unroll(xi)
     .unroll(yii);

 prod.compute_at(gemm, yi)
     .vectorize(x, 8).unroll(y);

 prod.update()
     .reorder(x, y, k)
     .vectorize(x, 8)
     .unroll(x)
     .unroll(y)
     .unroll(k, 2);
```

execution result

```
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
```

The performance of this step has been optimized by nearly 80 times from STEP1, and the performance is getting closer and closer to OpenBlas.

STEP 7 The operation added in this step is to rearrange the data of matrix B to make the memory read smoother when calculating the small kernel 16x8. Because the x dimension of the small kernel is divided according to 16, the x dimension of rearranged data B is also rearranged according to 16.



interleave.png

Codes are listed below

```
        Func B_interleave("B"), Bs("Bs");
        Bs(x, y, xo) = B(xo * 16 + x, y);
        B_interleave(x, y) = Bs(x % 16, y, x / 16);

        Func prod;
        prod(x, y) += A(k, y) * B_interleave(x, k);
        gemm(x, y) = prod(x, y);

        gemm.tile(x, y, xi, yi, 16, 32)
             .fuse(x, y, xy).parallel(xy)
             .split(yi, yi, yii, 4)
             .vectorize(xi, 8)
             .unroll(xi)
             .unroll(yii);

         prod.compute_at(gemm, yi)
             .vectorize(x, 8).unroll(y);

         prod.update()
             .reorder(x, y, k)
             .vectorize(x, 8)
             .unroll(x)
             .unroll(y)
             .unroll(k, 2);
         Bs.compute_root()
             .split(y, yo, yi, 16)
             .reorder(x, yi, xo, yo)
             .unroll(x)
             .vectorize(yi).parallel(yo, 4);
```

execution result

```
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
```

So far, every step of our tuning strategy has always been optimized around the two optimization purposes “optimizing memory access” and “improving parallelism”. In the end, the performance is almost the same as OpenBlAS, and the distance from STEP1 has been accelerated by 200+ times.

CHAPTER

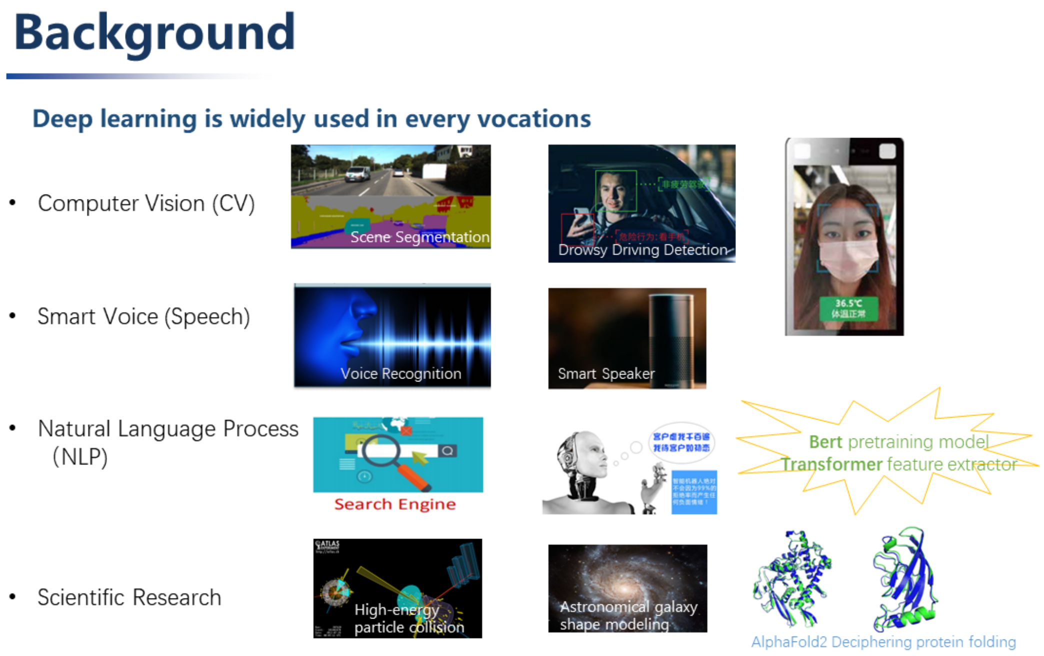
TEN

AUTOKERNEL: TO KNOW MORE ABOUT AI COMPILER

10.1 Overview of AI Compiler

In recent years, AI technology oriented with machine learning and deep learning has been rapidly developed, and deep neural networks have been widely used in various domains:

1. CV (computer vision): object detection, scene recognition, image segmentation, etc.
2. Smart voice: voice recognition, voiceprint recognition, etc.
3. NLP (Natural Language Processing): automatic search engine, dialogue service robot, text classification, intelligent translation, etc.
4. Scientific research: applied in many research fields such as physics, biology, medicine, etc. High-energy particle collision classification, cosmic celestial map data analysis, galaxy shape modeling, protein folding prediction of biological structure, precision medical treatment and disease prediction.



Background

Deep learning is widely used in every vocations

- Computer Vision (CV)
- Smart Voice (Speech)
- Natural Language Process (NLP)
- Scientific Research

Scene Segmentation
Drowsy Driving Detection
36.5℃
Voice Recognition
Smart Speaker
Search Engine
Bert pretraining model
Transformer feature extractor
High-energy particle collision
Astronomical galaxy shape modeling
AlphaFold2 Deciphering protein folding

Figure1.png

These applications have spawned more new models: CNN, RNN, LSTM, GAN, GNN, and also spawned the emergence of deep learning frameworks such as Tensorflow, Pytorch, Mxnet, and Caffe. At present, the training framework has begun to converge, gradually forming a duo situation where PyTorch leads the academic world and TensorFlow leads the industry.

However, in order for deep learning algorithms to be implemented, they must be deployed on hardware, such as Google's TPU, Huawei's Kirin NPU, and other architectural innovations on FPGA.

Figure2.png

When it comes to how to deploy the models trained by these training frameworks to different terminal hardware, this requires a deep learning neural network compiler to solve this problem.

Before the emergence of neural network compilers, we used traditional compilers.

Traditional Compilers

Take LLVM (low level virtual machine) as an example, its input is high-level programming language source code, and its output is machine code. It consists of a series of modular compiler components and tool chains.LLVM is divided into three parts: front-end, middle-end (optimized) and back-end through modules. Whenever a new programming language appears, only the corresponding front-end needs to be developed, and the programming language is converted into the intermediate representation of LLVM; similarly, when a new hardware architecture appears, only the corresponding back-end needs to be developed and connected to the intermediate representation of LLVM.The modularization avoids the problem of compiler adaptability caused by the refurbishment of the programming language and CPU architecture, and greatly simplifies the development of the compiler.

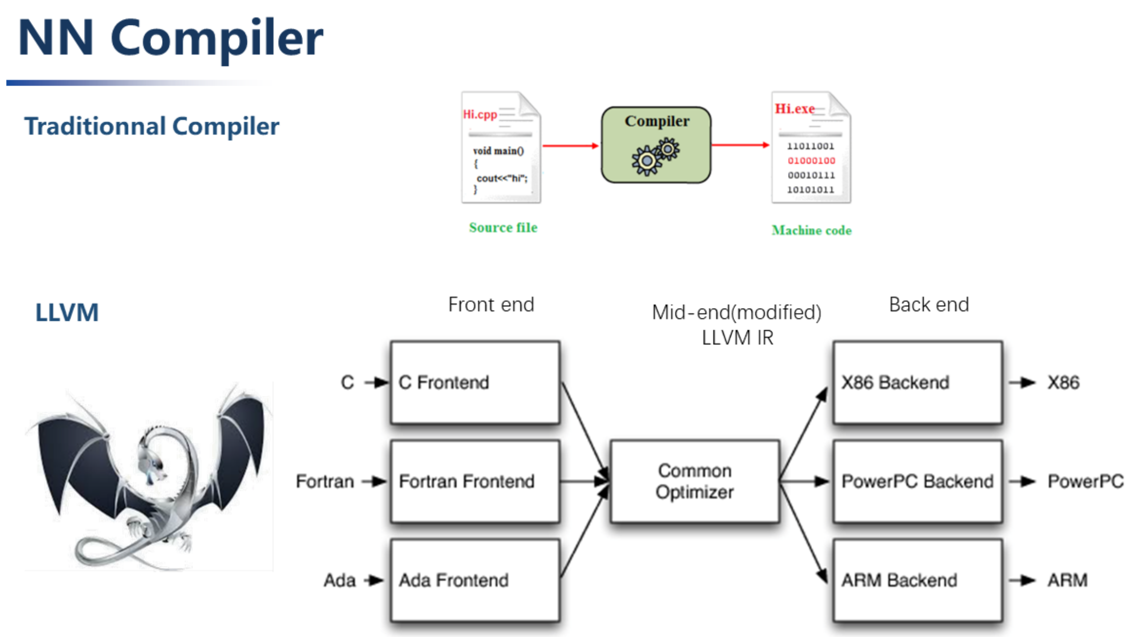


Figure3.png

Neutral Network Compiler

Its input is the model definition file trained by the deep learning training framework, and the output is the code that can be efficiently executed on different hardware.

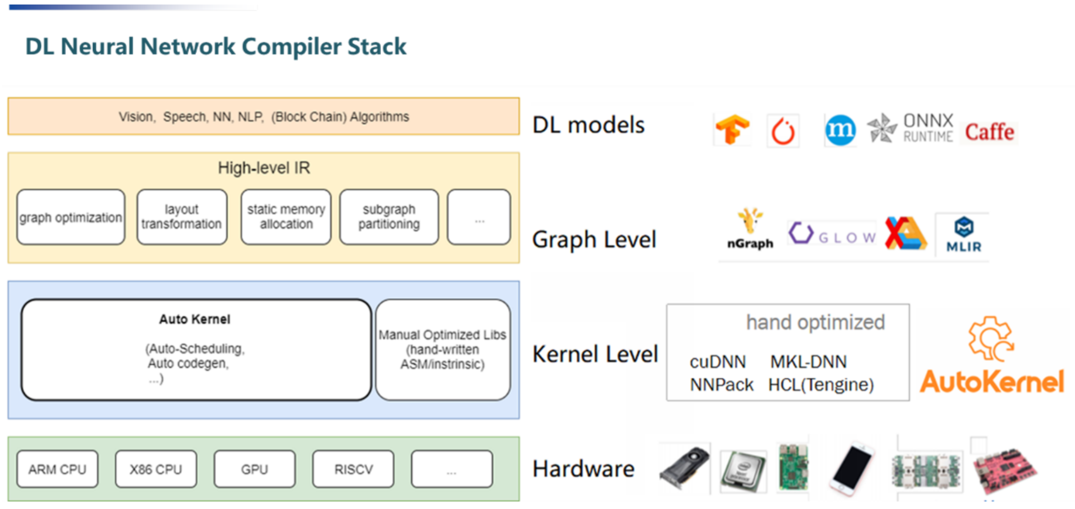


Figure6.png

It is composed of four levels from top to bottom

1. The top layer is connected to the algorithm models trained by various deep learning training frameworks (Tensorflow, Caffe, Pytorch, Mxnet, etc.).
2. Layer level (High-level IR): The structure of the neural network can be expressed as a calculation graph, and layer-level operations are to perform some operations on the calculation graph that have nothing to do with the specific hardware and framework, such as operator fusion, memory allocation optimization, Derivation of data types and data dimensions, etc.

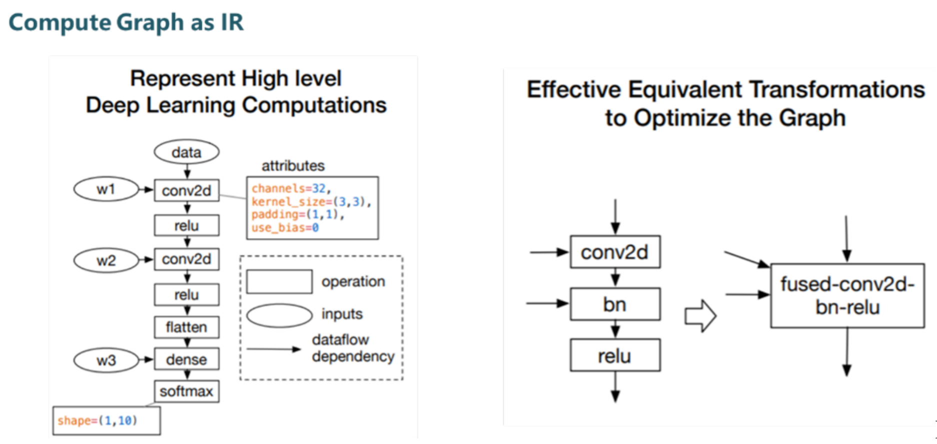


Figure4.png

We can use operator fusion to avoid frequent direct reading and writing of intermediate data between registers and memory, thereby improving overall inference performance

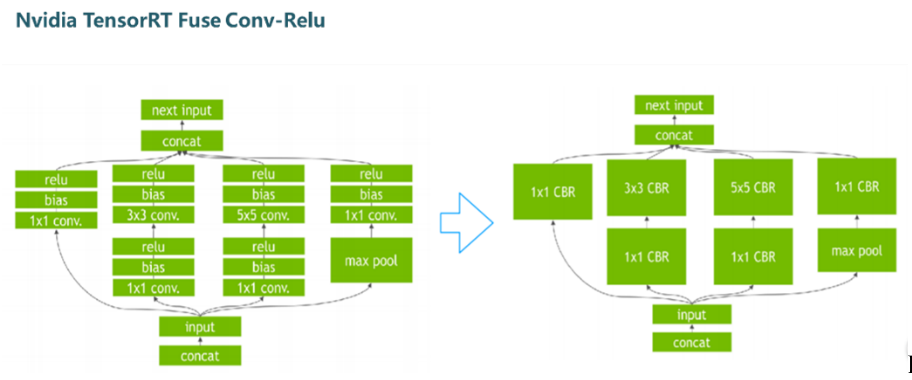


Figure5.png

Nividia has achieved a threefold increase in inference performance by fusing conv, bn, and relu into one operator fuse-CBR.

1. Operator level (operator level/kernel level): The operator level is mainly about tensor calculation. In order to realize these calculations efficiently on the hardware and give full play to the performance of the chip, the hardware chip is usually equipped with specially optimized operator calculation libraries, such as Intel's MKL, Nvidia's CuDNN, and TensorRT. This level needs to support every operator implementation of every hardware backend.

2. Various hardware backends: GPU, ARM CPU, X86 CPU, NPU, etc.

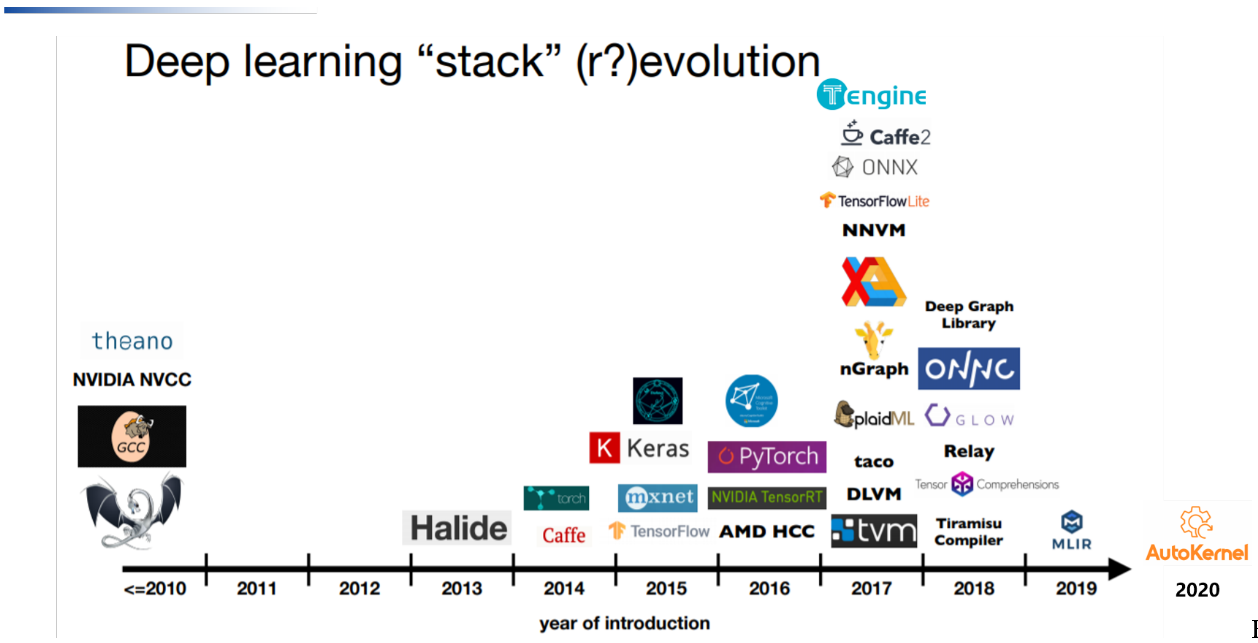


Figure7.png

Since the concept of the deep learning compiler was put forward, various types of compilers have emerged in an endless stream

10.2 TVM

In the wave of rapid development of compilers, one of the most prominent one is TVM (Tensor Virtual Machine)

TVM was first proposed in 2017 as a compiler stack for deep learning systems.

The design of the first generation of TVM draws on the design ideas of the traditional compiler framework LLVM, and the design abstracts the middle presentation layer. Different models only need to develop corresponding front-end interfaces, and different back-ends only need to develop corresponding back-end interfaces.

TVM is called Tensor Virtual Machine, which belongs to the operator level. It is mainly used for tensor calculation and provides an intermediate representation of the underlying calculation independent of the hardware. Various methods (circular block, cache optimization, etc.) are used to optimize the corresponding calculation. The first generation of layer-level representation is called NNVM (Neural Network Virtual Machine). The design goal of NNVM is to convert the calculation graph from a different deep learning framework into a unified intermediate representation (IR) of the calculation graph and optimize it.

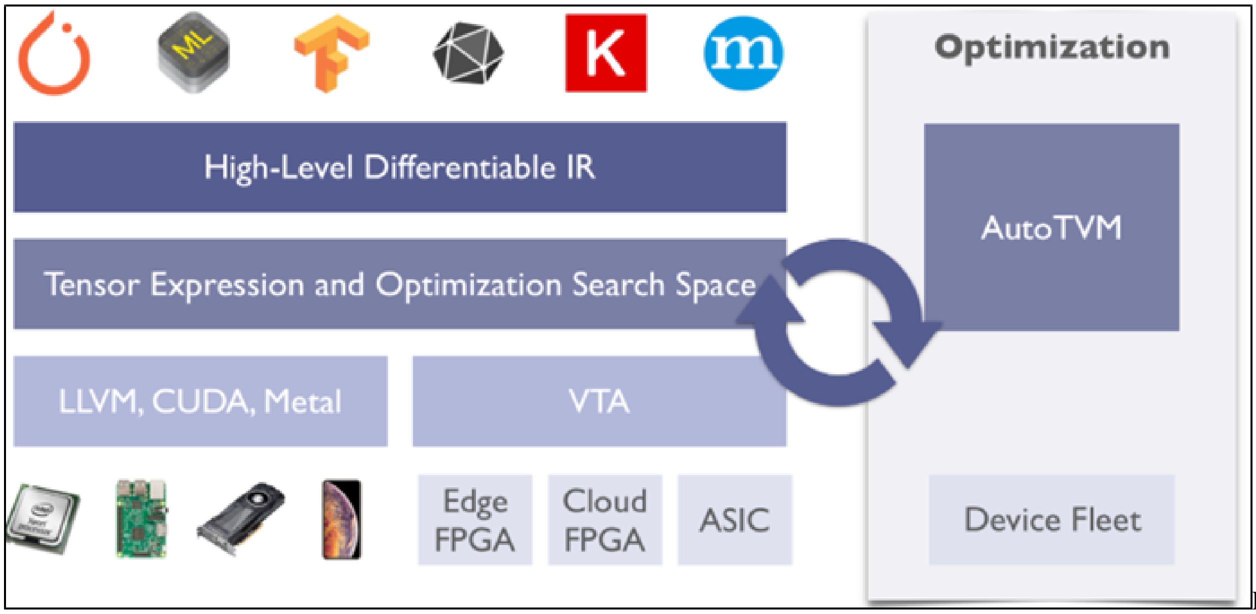
The first generation of static images has certain defects:

1. Cannot support control flow well, such as branch jump, loop, etc.
2. Can not support the input shape of the calculation graph, depending on the input tensor size model, such as word2vec.

Although Tensorflow has a control interface like tf.cond.Tf.while_loop to solve the first problem to some extent, tf.fold solves the second problem, but this method is not particularly friendly for rookies who have just come into contact with deep learning framework.

The dynamic graphs that appear later abandon the traditional way of calculating graphs that are defined first and then executed, and adopt the mode defined by the calculation graph at runtime. This kind of calculation graph is called a dynamic graph.

The graph calculation layer of the second-generation TVM becomes Relay VM. The main difference between Relay and the first-generation graph calculation indicator NNVM is that in addition to supporting dataflow (static graph), Relay IR can better solve control flow (dynamic graph). It is not only an intermediate representation of a calculation graph, but also supports automatic differentiation.


Optimization
High-Level Differentiable IR
Tensor Expression and Optimization Search Space
AutoTVM
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC
Device Fleet
Figure19.png

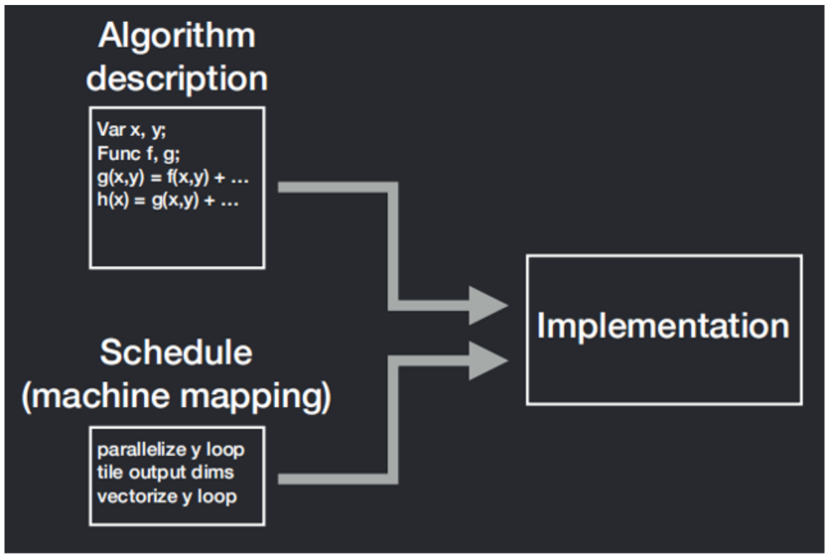
To sum up, the current structure of TVM is:

1. The highest level supports mainstream deep learning front-end frameworks, including TensorFlow, MXNet, Pytorch, etc.
2. Relay IR supports differentiability. This level performs graph fusion, data rearrangement and other graph optimization operations.
3. Based on the tensor quantization calculation graph and the hardware primitive-level optimization according to the back-end, autoTVM explores the search space according to the optimization goal and finds the optimal solution.
4. The backend supports ARM, CUDA/Metal/OpenCL, accelerator VTA (Versatile Tensor Accelerator).

10.3 Halide

Halide was proposed in 2012 and is mainly used for automatic optimization. Embedded in C++, it is a programming language specially designed for image processing by MIT researchers. The Halide language is easy to write, simple in grammar, clear in data structure, and can automatically optimize the code, so that the program can achieve better execution efficiency.The core idea of its design is to separate algorithm and scheduling. The advantage of this is that in the case of a given algorithm, you only need to adjust its Schedule scheduling options, without having to rewrite the algorithm to implement a different schedule. When adjusting the schedule and exploring the design space, there is no worry that the correctness of the calculation will change due to the rewriting of the algorithm.

The Algorithm part is mainly the mathematical expression of algorithm description and calculation.The Schedule part tells the machine when to allocate memory and how to calculate (block calculation or sequential calculation)-some scheduling strategies have been provided.



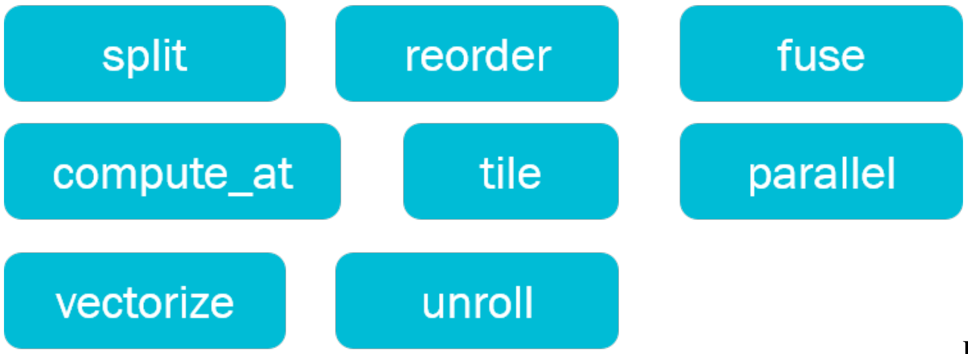
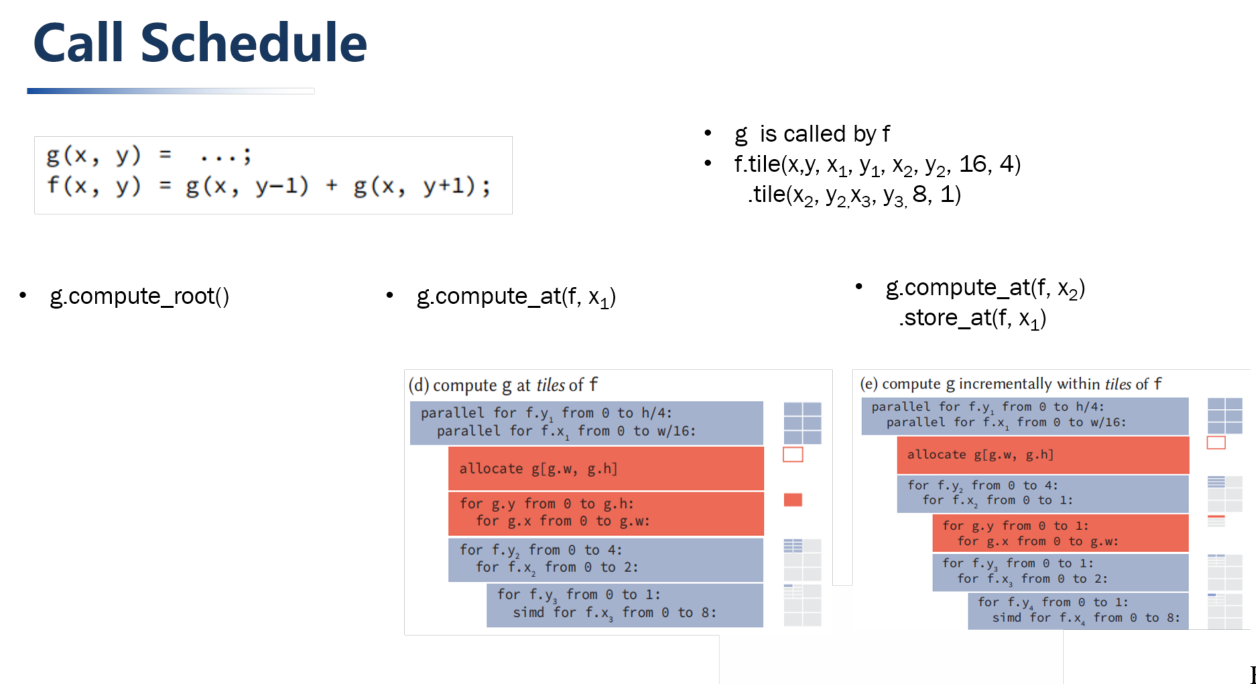


Figure17.png



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)

(d) compute g at *tiles* of f

```
parallel for f.y1 from 0 to h/4:
  parallel for f.x1 from 0 to w/16:
    allocate g[g.w, g.h]
    for g.y from 0 to g.h:
      for g.x from 0 to g.w:
    for f.y2 from 0 to 4:
      for f.x2 from 0 to 2:
        for f.y3 from 0 to 1:
          simd for f.x3 from 0 to 8:
```

(e) compute g incrementally within *tiles* of f

```
parallel for f.y1 from 0 to h/4:
  parallel for f.x1 from 0 to w/16:
    allocate g[g.w, g.h]
    for f.y2 from 0 to 4:
      for f.x2 from 0 to 1:
        for g.y from 0 to 1:
          for g.x from 0 to g.w:
        for f.y3 from 0 to 1:
          for f.x3 from 0 to 2:
            for f.y4 from 0 to 1:
              simd for f.x4 from 0 to 8:
```

Figure18.png

Different scheduling strategies consider the trade-off between repeated redundant calculations and locality

10.4 AutoKernel

Whether the deep learning model can be successfully applied to the terminal and meet the product needs, a key indicator is the reasoning performance of the neural network model.

The current high-performance operator calculation library is mainly developed manually by high-performance computing optimization engineers. However, the continuous emergence of new algorithms/hardware has led to a huge workload of operator-level optimization and development. At the same time, optimizing the code is not a simple task. It requires engineers to be proficient in computer architecture as well as to be familiar with the calculation process of operators.There are few talents, high demand, and high technical threshold. Therefore, we believe that operator optimization and automation is the general trend in the future. The original intention of proposing AutoKernel is to automate this process, start small, optimize at the operator level, and realize the automatic generation of optimized code.

Figure11.png

The input of AutoKernel is the calculation description of the operator (such as Conv, Poll, Fc), and the output is the optimized acceleration source code.The development of this tool aims to lower the threshold of optimization work, without the knowledge threshold of the underlying assembly, and no need to write the optimization assembly by hand. The assembly code can be generated by directly calling the developed toolkit. At the same time, it also provides a docker environment containing CPU and GPU, no need to deploy development environment, just use docker. You can also integrate the automatically generated operator into the inference framework-Tengine with one click through the provided plug-in-plugin.

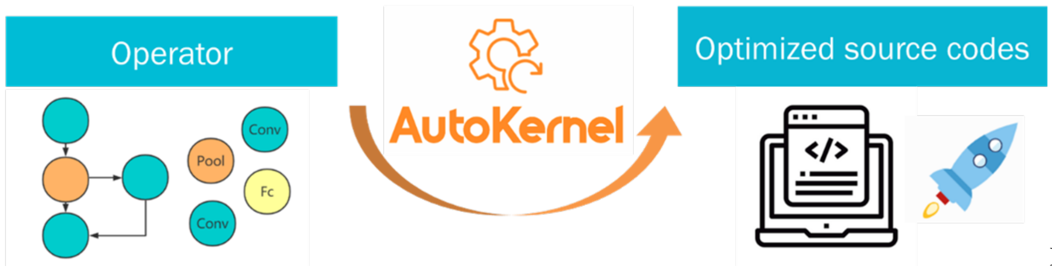


Figure12.png

Correspondingly, the AutoKernel at the operator level is mainly divided into three modules,

1. Op Generator: Operator generator, using the open source Hallide.
2. AutoSearch: Currently under development, the goal is to automatically search for optimization strategies through machine learning and reinforcement learning commonly used algorithms.
3. AutoKernel Plugin: Insert the generated automatic operator into Tengine in the form of a plug-in, which complements manual customization.

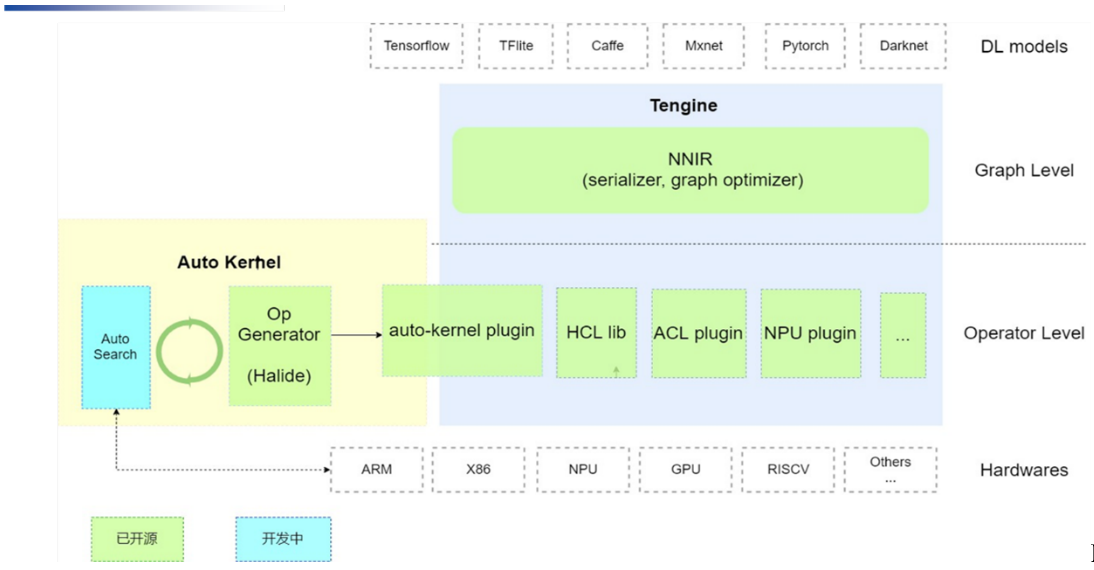


Figure13.png

The Tengine object layer is connected to different neural network models. The layer-level NNIR includes model analysis and layer optimization, and the operator level includes the high-performance computing library (HCL lib).

AutoKernel Plugin is mainly divided into two parts: generation and deployment. In the generation part, Halid is used to fill in the algorithm description and scheduling strategy, and the back-end type is specified during execution (which basically covers the current mainstream back-ends). The deployment part is packaged as a Tengine library and directly called.

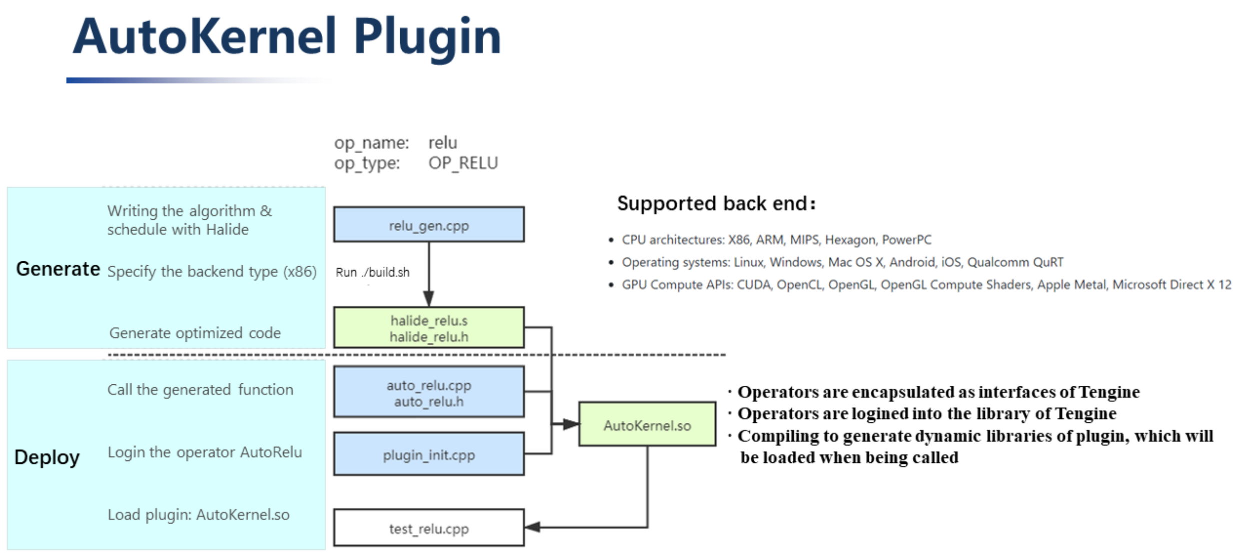

AutoKernel Plugin
op_name: relu
op_type: OP_RELU
Generate
Writing the algorithm & schedule with Halide
Specify the backend type (x86)
Generate optimized code
relu_gen.cpp
Run ./build.sh
halide_relu.s
halide_relu.h
Supported back end:
• CPU architectures: X86, ARM, MIPS, Hexagon, PowerPC
• Operating systems: Linux, Windows, Mac OS X, Android, iOS, Qualcomm QuRT
• GPU Compute APIs: CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal, Microsoft Direct X 12
Deploy
Call the generated function
Login the operator AutoRelu
Load plugin: AutoKernel.so
auto_relu.cpp
auto_relu.h
plugin_init.cpp
test_relu.cpp
AutoKernel.so
· Operators are encapsulated as interfaces of Tengine
· Operators are logined into the library of Tengine
· Compiling to generate dynamic libraries of plugin, which will be loaded when being called

Figure14.png

We believe that with the contribution of more developers, the AutoKernel community will have greater breakthroughs and growth, leaving a strong fortune in the field of deep learning compilers in the future!