Homework 12: Halide Scheduling

Introduction to Big Data Systems course

Due: December 25, 2022 23:59 China time. Late submission results in lower (or even no) scores.

For questions or concerns, contact TA (Huanqi Cao, Mingzhe Zhang) by WeChat. Or send an email to caohq18@mails.tsinghua.edu.cn or zmz21@mails.tsinghua.edu.cn if you could not use WeChat.

Overview

In this assignment, you will optimize a dilated convolution algorithm on a multi-core CPU. You need to **write Halide schedules** for optimizing parallelism and data locality.

Environment

You need to run your code **on the server** for this assignment.

We have placed source code on the server at `/data/hw12_src`.

About Halide

Halide is a programming language designed to make it easier to write high-performance image processing code on modern machines. It can also be used to generate and optimize tensor operators. Halide decouples the algorithm definition from its execution strategy. Many loop-nest optimizations can be easily specified in Halide, enabling radically simpler programs to achieve higher performance than hand-tuned expert implementations.

We have provided a binary release of Halide (14.0.0) on the server located in `/halide` directory.

To get started writing code, please look through the tutorials.

Task 1

Background: Dilated Convolution

Regular convolution (Figure 1 (a)):

$$conv(n, h, w, oc) = \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} input(n, h + kh, w + kw, ic) * filter(ic, kh, kw, oc)$$

For simplicity, we ignore bias, and assume that data format is `NHWC` and `stride=1`.

Dilated convolution, as shown in Figure 1 (b), is a variant of basic convolution, which expands the kernel by inserting "holes" between its consecutive elements.

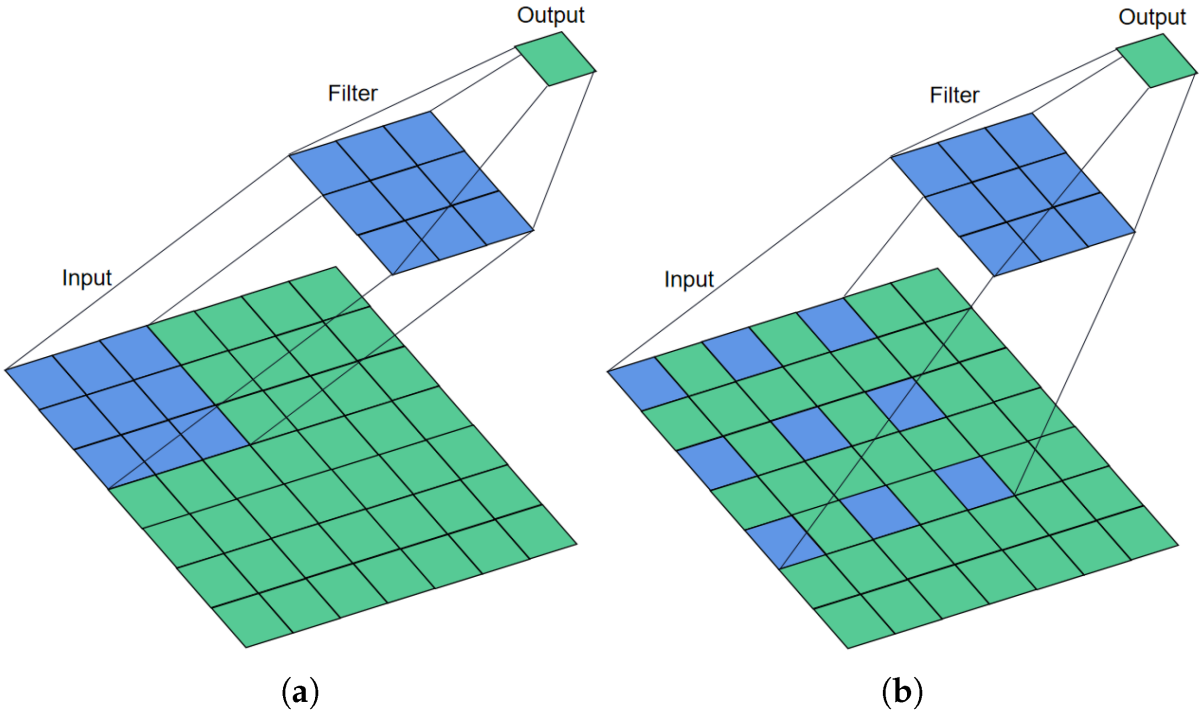

Figure 1.

Dilated convolution algorithm is defined by the following formula:

$$dilated\_conv(n, h, w, oc) = \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} input(n, h + kh \cdot (dh + 1), w + kw \cdot (dw + 1), ic) \cdot filter(ic, kh, kw, oc)$$

where `dh` and `dw` tells how many elements are skipped in the input.

Algorithm Definition in Halide

```
ImageParam input(type_of<float>(), 4);
ImageParam filter(type_of<float>(), 4);
Var x("x"), y("y"), c("c"), n("n");
Func dilated_conv("dilated_conv");
RDom r(0, CI, 0, KW, 0, KH);

dilated_conv(c, x, y, n) = 0.0f;
dilated_conv(c, x, y, n) += filter(c, r.y, r.z, r.x) * input(r.x, x + r.y * (dilation + 1), y +
r.z * (dilation + 1), n);
```

This representation is quite similar to its math formula. The iteration bounds for the reduction are expressed explicitly using reduction domains (`RDom`).

Reductions are defined in two parts:

- An initial value function, which specifies a value at each point in the output domain.
- A recursive reduction function, which redefines the value at points given by an output coordinate expression in terms of prior values of the function.

Note: in Halide, the order of dimensions is **from innermost to outermost**. That's why we have `(c, x, y, n)` instead of `(n, y, x, c)`.

Inspecting and debugging

Tracing is one useful way to understand what a schedule is doing. You can also ask Halide to print out pseudocode showing what loops Halide is generating for this function:

```
dilated_conv.print_loop_nest();
```

You will see the following loop nests:

```
produce dilated_conv:
  for n:
    for y:
      for x:
        for c:
          dilated_conv(...) = ...
  for n:
    for y:
      for x:
        for c:
          for r12 in [0, 2]:
            for r12 in [0, 2]:
              for r12 in [0, 127]:
                dilated_conv(...) = ...
```

As we explained before, the first loop nest is for initialization and the second is for reduction.

```
dilated_conv.update().reorder(n, y, x, c);
dilated_conv.print_loop_nest();
```

Here we reorder variables of the second loop nest to have the **reversed order**:

```
produce dilated_conv:
  for n:
    for y:
      for x:
        for c:
          dilated_conv(...) = ...
  for c:
    for x:
      for y:
        for n:
          for r12 in [0, 2]:
            for r12 in [0, 2]:
              for r12 in [0, 127]:
                dilated_conv(...) = ...
```

If you don't understand what the Halide compiler is generating by each schedule, this technique will help you. See tutorials for more information.

What you need to do?

We provide a initial code (`dilated_conv.cpp`) which only contains the algorithm definition part. We would like you to **optimize it by applying and composing different scheduling strategies**. We also provide implementations of two most common operators (`matmul.cpp` and `conv.cpp`). You can first take a look at them.

We will **benchmark your implementation against oneDNN** (oneAPI Deep Neural Network Library). Besides, your code must **pass the correctness test**, where all inputs are randomly generated.

Hint:

You may use these scheduling primitives for optimization: `unroll` , `vectorize` , `parallel` , `split` , `tile` , `reorder` , `fuse` ...

For more detail, see Halide API documentation.

How to run the code

We provide a `Makefile` in the code directory. For dilated convolution, specify target name to `make` from command line:

```
make dilated_conv
```

Then you can run your code, checking correctness and performance against oneDNN:

```
srun -n 1 ./dilated_conv
```

If your implementation is correct, there will be a message and your running time will be reported:

```
Halide results - OK
Halide: 10021.307288ms, 1.177140 GFLOP/s
oneDNN: 39.070930ms, 301.924735 GFLOP/s

Success!
```

Task 2 (optional)

Background: Batch Normalization

The batch normalization operation is defined by the following formula:

$$dst(n,h,w,c) = \frac{src(n,h,w,c) - \mu(c)}{\sqrt{\sigma^2(c) + \epsilon}}$$

where mean and variance are computed at runtime:

$$\mu(c) = \frac{1}{NHW}\sum_{nhw} src(n,h,w,c)$$

$$\sigma^2(c) = \frac{1}{NHW}\sum_{nhw}(src(n,h,w,c) - \mu(c))^2$$

ϵ is a constant to improve numerical stability.

Operator Fusion

Operator fusion combines multiple operators into a single kernel without additional roundtrip to memory. The basic idea of such fusion is the same as the traditional loop fusion, and they lead to the following benefits: (i) eliminating unnecessary materialization of intermediate results, (ii) reducing unnecessary scans of the input; and (iii) enabling other optimization opportunities.

We provide `op_fuse.cpp` as a starting code. You need to merge batch normalization into dilated convolution and schedule the fused operator to make use of better locality and parallelism.

Scoring

Task 1 (60%)

You will get 20 point as long as your program is faster than the starting code without any optimizations.

For the rest 40 point, the faster your implementation is, the more scores you will get.

Task 2 (up to +20%)

The faster your implementation is, the more scores you will get.

Report (40%)

Your report should at least contains:

- what scheduling primitives you applied and why you used them
- performance comparison between your implementation and oneDNN with `dilation=0,15,31,63`

Hand-in

Please submit your assignment containing your report and your code `dilated_conv.cpp` and `op_fuse.cpp` (optional).

References

1. J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph., 31(4), 2012.
2. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2013.
3. Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. OSDI 2018.
4. Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2021.