编译原理(A)大作业

算符优先分析表和卷积优化

Log Creative

2021 年 6 月 27 日

目录

第一部分　算符优先分析表

运行环境

操作系统　Windows
语言　Rust 2018[1]

```
opg [filename]
```

程序输出

样例 1 输入

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | i
```

样例 1 输出

```
      *     (     )     +     i     $
*     >     <     >     >     <     >
(     <     <     =     <     <
)     >           >     >           >
+     <     <     >     >     <     >
i     >           >     >           >
$     <     <           <     <     =
```

样例 2 输入

```
E -> E + E | E * E | ( E ) | id
```

样例 2 输出

```
The grammar is ambiguous.
```

1 题目分析

首先需要声明在算符优先语法中算符优先级的定义[2]。

定义 1 对于两个终结符 T_1 和 T_2，有下面的算符优先级定义（其中 U_1 是非终结符）

1. $T_1 = T_2$ 如果存在产生式 $U \to xT_1T_2y$ 或 $U \to xT_1U_1T_2y$。
2. $T_1 < T_2$ 如果存在产生式 $U \to xT_1U_1y$ 而且存在一个推导 $U_1 \Rightarrow z$ 使得 T_2 是 z 的最左终结符。
3. $T_1 > T_2$ 如果存在产生式 $U \to xU_1T_2y$ 而且存在一个推导 $U_1 \Rightarrow z$ 使得 T_1 是 z 的最右终结符。

本部分即针对一个上下文无关文法，输出算符优先分析表。如果文法是有二义性的，将会报错。如果两个终结符之间没有上述三个关系的其中一个，将会留空，意为没有优先关系。

根据定义 1，可以对符号构造下面两个集合以判断情况 2 和 3：

定义 2 假设 V_T 是该文法终结符号对应的集合，V_N 是该文法非终结符号对应的集合。对符号 U_1 定义下面两个集合：

$$FIRSTVT(U_1) = \{T|(U_1 \Rightarrow Ty \vee U_1 \Rightarrow U_2Ty) \wedge T \in V_T \wedge U_2 \in V_N\} \quad (1)$$

$$LASTVT(U_1) = \{T|(U_1 \Rightarrow xT \vee U_1 \Rightarrow xTU_2) \wedge T \in V_T \wedge U_2 \in V_N\} \quad (2)$$

除了上面的定义方法，对于这样的集合，还有这样的性质：

$$(U_1 \Rightarrow U_2y) \Rightarrow FIRSTVT(U_2) \subseteq FIRSTVT(U_1) \quad (3)$$

$$(U_1 \Rightarrow xU_2) \Rightarrow LASTVT(U_2) \subseteq LASTVT(U_1) \quad (4)$$

这样定义 1 就有了如下的等价定义：

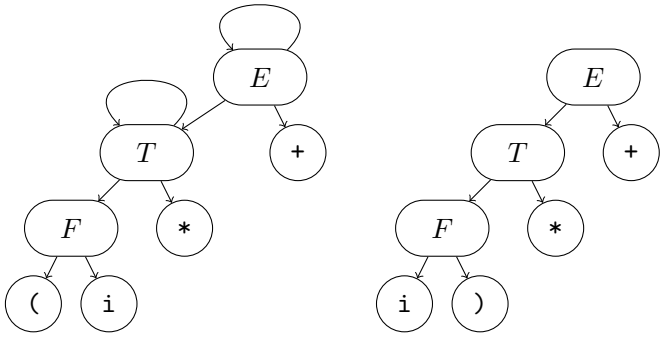
定义 3 对于两个终结符 T_1 和 T_2，有下面的算符优先级定义（其中 U_1 是非终结符）

1. 如果找到了这样的产生式右部 T_1T_2 或 $T_1U_1T_2$，那么 $T_1 = T_2$。
2. 如果找到了这样的产生式右部 T_1U_1 且 $T_2 \in FIRSTVT(U_1)$，那么 $T_1 < T_2$。
3. 如果找到了这样的产生式右部 U_1T_2 且 $T_1 \in LASTVT(U_1)$，那么 $T_1 > T_2$。

2 算法描述

2.1 集合依赖关系

首要任务就是对于每一个非终结符求出 $FIRSTVT$ 和 $LASTVT$。式 (1) 和 (2) 所对应的终结符为图中的盲端，式 (3) 和 (4) 所对应的非终结符导出的依赖节点为中间节点。这样就可以构造出集合的依赖关系有向图。



(a) $FIRSTVT$依赖关系图　　(b) $LASTVT$依赖关系图

图 1: 依赖关系图

2.2 三进 DFS

一进 DFS：归并分类　现在的依赖关系图依然是有向有环图。根据包含关系的特性，如果出现

$$S_1 \subseteq S_2 \subseteq \cdots \subseteq S_n \subseteq S_1$$

所对应的环路

$$S_1 \to S_2 \to \cdots \to S_n \to S_1$$

则这些集合都是相等的：

$$S_1 = S_2 = \cdots = S_n$$

使用 DFS 检测环路，并采用类似并查集的方法归并同类的元素，记录每个非终结符所对应的集合号码，以及每一类的非终结符集合。

表 1: 类别号码

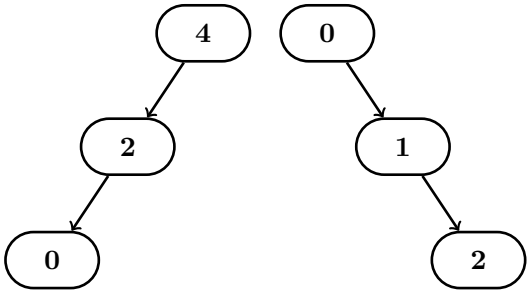
(a) *FIRSTVT*

F	E	T
0	**2**	**4**

(b) *LASTVT*

F	E	T
1	**2**	**0**

二进 DFS：连接消环 通过再一次的 DFS 构造类别之间的 DAG（有向无环图），因为此时同类型内部的环边已经被消除。



(a) *FIRSTVT* (b) *LASTVT*

图 2: 类别连接

三进 DFS：映射输出 对于每一个非终结符，查类别号码表 1 得到其对应类别，然后在类别连接图 2 中从该类别节点进行 DFS，合并路上对应的终结符节点，就可以得到该非终结符对应的集合。

表 2: 集合

(a) *FIRSTVT*

E	`i + ( *`
T	`i ( *`
F	`i (`

(b) *LASTVT*

E	`i + ) *`
T	`i ) *`
F	`i )`

2.3 识别关系

最后一步，就是根据定义 3 来遍历产生式检测终结符号之间的优先关系了。在识别之前，需要先在产生式的集合中加入对于开始符号的输入首

尾标记：

$$E \to \$E\$$$

因为该符号比较特殊，其优先级是额外定义的[3]：

$$\$ < T, T > \$, \$ = \$, \quad \forall T \in V_T$$

并不属于原有的语法，所以要在这个地方再加入。

最终就可以得到本部分开头的程序输出。

3 函数接口

以下是运行样例 1 时由 uftrace[4] 生成的函数调用列表。

```
std::rt::lang_start() {
  std::rt::lang_start::_{{closure}}() {
    std::sys_common::backtrace::__rust_begin_short_backtrace() {
      core::ops::function::FnOnce::call_once() {
        main::main() {                                  // 程序入口
          main::opg_generate() {                        // 生成函数
            main::gen_productions();                    // 生成产生式
            main::gen_non_terminals();                  // 生成非终结符
            main::gen_firstvt() {                       // 构造 FIRSTVT 集
              main::dfs::compose_elements() {           // 聚合元素
                main::dfs::Dfs::dfs() {                 // DFS 函数
                  main::dfs::Dfs::dfs_merge() {         // 一进 DFS
                    main::dfs::Dfs::merge();            // 一进 DFS 子函数
                  } /* main::dfs::Dfs::dfs_merge */
                  main::dfs::Dfs::dfs_conn();           // 二进 DFS
                  main::dfs::Dfs::dfs_map();            // 三进 DFS
                } /* main::dfs::Dfs::dfs */
              } /* main::dfs::compose_elements */
            } /* main::gen_firstvt */
            main::gen_lastvt() {                        // 构造 LASTVT 集
              main::dfs::compose_elements() {
                main::dfs::Dfs::dfs();
              } /* main::dfs::compose_elements */
            } /* main::gen_lastvt */
            main::get_terminals();                      // 生成终结符
            main::find_eq();                            // 寻找相等关系
            main::find_less();                          // 寻找小于关系
            main::find_greater();                       // 寻找大于关系
            _<main..table..OpTable as core..fmt..Display>::fmt() { // 输出
              main::table::OpTable::to_string();
            } /* _<main..table..OpTable as core..fmt..Display>::fmt */
            main::table::OpTable::to_string();
          } /* main::opg_generate */
        } /* main::main */
      } /* core::ops::function::FnOnce::call_once */
```

```
    } /* std::sys_common::backtrace::__rust_begin_short_backtrace */
  } /* std::rt::lang_start::_{{closure}} */
} /* std::rt::lang_start */
```

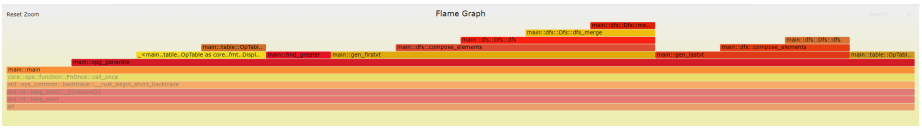


图 3: 火焰图[5]

以上结果由 该脚本 生成。关于函数的更多信息，请访问 API 文档，里面详细描述了函数的调用方法和原理。

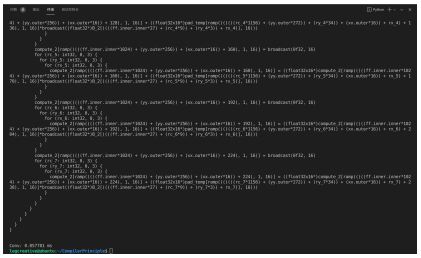
第二部分　卷积优化

运行环境

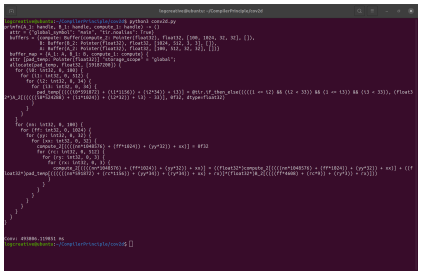
操作系统	Ubuntu 20.04 LTS 64 位
内存	15.6 GiB
处理器	Intel® Core™ i7-7660U CPU @ 2.50GHz × 4
L1 Cache	4 KiB × 2
L2 Cache	32 KiB
L3 Cache	4 MiB
TVM 版本	0.8dev0
LLVM 版本	10.0.0
Python 版本	3.8

运行截图

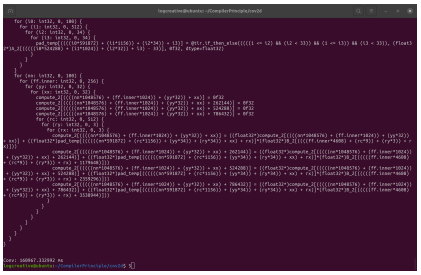
(a) 样例 1 未优化



(b) 样例 1 优化后



(c) 样例 2 未优化



(d) 样例 2 优化后

优化结果

输入大小和输出大小	优化前	优化后	提升效率
`n, ic, ih, iw = 1, 3, 32, 32` `oc, kh, kw = 32, 3, 3`	0.165312 ms	0.057781 ms	65.0%
`n, ic, ih, iw = 100, 512, 32, 32` `oc, kh, kw = 1024, 3, 3`	493806.119851 ms	160967.332992 ms	67.4%

4 源代码

全部源码请见 `conv2d/conv2d.py`。

由于采用了 0.8 版本的 TVM，引用包的方式发生了变化。

```
import tvm
import numpy as np
import tvm.topi
```

```
from tvm import autotvm
import logging
from numbers import Integral
from tvm.topi.nn.utils import get_pad_tuple
from tvm.topi.nn.conv2d import conv2d_nchw
from tvm.contrib import utils
```

将大小定义提前，因为需要通过大小判定优化方式。

```

# 声明输入输出的大小

# small batch
# target: 0.072
n, ic, ih, iw = 1, 3, 32, 32
oc, kh, kw = 32, 3, 3

# # huge batch
# # target: 197522.4
# n, ic, ih, iw = 100, 512, 32, 32
# oc, kh, kw = 1024, 3, 3
```

下面就是优化的 schedule 函数。

```
声明输入输出的大小优化启用开关
# optimize_on = False
optimize_on = True

# 这个函数是需要大家自己补充的，是需要调用各种的原语进行优化的schedule
def schedule(output):
    s = tvm.te.create_schedule(output.op)

    if(optimize_on):

        if oc > 256 :
            ## For large matrix,
            # the efficiency could be improved by
            # Virtual Multithreading
            # the computation could be distributed to
            # 4 threads.
            v_thread = 4
            # split the f dimension
```

```
42            fo, fi = s[output].split(s[output].op.axis[1],
                  nparts=v_thread)
43
44            # Virtual Multithreading
45            s[output].bind(fo, tvm.te.thread_axis("cthread"))
46        else:
47            ## For small matrix
48            # no multi-threading
49            v_thread = 1
50            # split the f dimension
51            n = s[output].op.axis[0]
52            fo, fi = s[output].split(s[output].op.axis[1],
                  nparts=v_thread)
53            # This step is not useful for all input with size
                  32
54            # Just in case that the input size is very large.
55            fio, fii = s[output].split(fi, factor=32)
56
57            # tile the block
58            yo, xo, yi, xi = s[output].tile(s[output].op.axis
                  [2], s[output].op.axis[3], x_factor=8, y_factor
                  =16)
59
60            # Loop reorder
61            s[output].reorder(n, fo, fio, fii, yo, xo, yi, xi)
62
63            # Unrolling
64            s[output].unroll(yi)
65
66            # Vectorization
67            s[output].vectorize(xi)
68
69    return s
```

5 优化原理

根据数据大小的不同，分为两种情况。

5.1 较大数据输入

较大的数据输入一般是因为只能使用一个核运行的缘故导致时间较长。这里参考 [6] 以使用虚拟多线程 (Virtual Threading) 机制以多线程对数据进行分裂 (fission)，因为卷积计算数据间没有依赖关系。

运行环境有 4 个核，所以会将线程数设置为 4。而为了让底层编译出正确的多线程代码，将不会使用下面的优化方式，否则会导致线程复杂而没有产生优化的效果。

5.2 较小数据输入

较小的数据输入如果使用多线程将会因为上下文切换产生过多的时间增长，产生负效应，这里参考 [7] 以对循环结构做基础的优化。

```
for (nn: int32, 0, 2) {
   for (ff: int 32, 0, 32) {
      for (yy: int32, 0, 32) {
         for (xx: int32, 0, 32) {
            compute2[((ff*1024) + (yy*32) + xx)] = 0f32
            for (rc: int32, 0, 3) {
               for (ry: int32, 0, 3) {
                  for (rx: int32, 0, 3) {
                     compute2[...]
                  }
               }
            }
         }
      }
   }
}
```

先使用 `tile` 原语分块处理两个轴 `yy` 和 `xx`，然后使用 `reorder` 原语重新排序：

```
for (nn: int32, 0, 2) {
   for (ff.outer: int32, 0, 4) {
      for (ff.inner.inner: int32, 0, 32){
         for (yy.outer: int32, 0, 4) {
            for (xx.outer: int32, 0, 2){
               for (yy.inner: int32, 0, 8){
                  for (xx.inner: int32, 0, 16){
                     compute2[...]
```

```
                    for (rc: int32, 0, 3) {
                        for (ry: int32, 0, 3) {
                            for (rx: int32, 0, 3) {
                                compute2[...]
                            }
                        }
                    }
                }
            }
        }
    }
}
```

对最内层的 `xi` 采用 `vectorize` 原语进行向量化，正好是 64 位的宽度（16× 4 位=64 位），也是缓存的行大小，达到最优状态。

对次内层的 `yi` 采用 `unroll` 原语循环展开，这样可以保证优化性能的稳定。实验证明，如果不展开，其优化结果的方差较大。

其余的常见优化原语已经不会帮助性能的提高，仅针对 `output` 优化至此。最内核的卷积核本文不考虑其优化问题。

安装 TVM

TVM 尚处于开发阶段，会有一些 API 更改。原始文件基于 TVM 0.6.1 版本，由于操作系统较新，其 LLVM 版本较高以致于删除了字符串转换函数[8]，不能顺利编译老版源文件。故采用 0.8.0 版本。

不能使用 GCC 编译，静态编译会导致定义冲突，并且会导致 TVM 的一些特性无法使用，任何报错初始化报错都应当认为是没有使用 Clang + LLVM 编译导致的。更改 `config.cmake` 中的 `Set(LLVM ON)` 以解决这个问题。

Windows 版本曾经尝试采用 Visual Studio 2019 编译。在 LLVM 12.0.0 下，0.8dev0 版本 TOPI 编译失败，0.6.1 版本都编译失败。而且 Windows 会跳过 VTA 的编译。

分工

单人独立完成。

参考文献

[1] C. N. Steve Klabnik, *The Rust Programming Language (Covers Rust 2018)*. Random House LCC US, 2019. [Online]. Available: https://www.ebook.de/de/product/37149179/steve_klabnik_carol_nichols_the_rust_programming_language_covers_rust_2018.html

[2] R. W. Floyd, "Syntactic analysis and operator precedence," *Journal of the ACM*, vol. 10, no. 3, pp. 316–333, Jul. 1963.

[3] Operator-precedence grammar. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Operator-precedence_grammar

[4] namhyung, "uftrace." [Online]. Available: https://github.com/namhyung/uftrace

[5] brendangregg, "FlameGraph." [Online]. Available: https://github.com/brendangregg/FlameGraph

[6] T. Moreau, "2D convolution optimization." [Online]. Available: https://tvm.apache.org/docs/vta/tutorials/optimize/convolution_opt.html?highlight=convolution

[7] W. Jian and Y. Ruofei, "How to optimize GEMM on CPU." [Online]. Available: https://tvm.apache.org/docs/tutorials/optimize/opt_gemm.html#sphx-glr-tutorials-optimize-opt-gemm-py

[8] tvm, "[LLVM] Explicit llvm::StringRef to std::string conversion," Feb. 2020. [Online]. Available: https://github.com/apache/tvm/pull/4859