Conservative Parallel Detection Algorithm for OpenACC

吴昊泽 · 张煜皓 · 陈牧歌

December 27, 2016

1 选题概述

OpenACC 指令在加速科学代码方面是一种简单而又可移植的方式。利用 OpenACC，只要在自己的 Fortran 或 C 语言代码中插入编译器提示，编译器即可将代码中计算量繁重的部分自动交由 GPU 处理，以实现更高的性能。

OpenACC 指令最大的长处就是简单，只需要插入一句编译器的提示，就可以将代码中的某个语句块进行并行优化。正因如此，我们小组选择的课题就是：针对 C 代码中的大量循环，通过对程序的静态分析，以检测哪些循环能够并行。

2 问题抽象

2.1 工具功能

由于 OpenACC 的指令较多，我们小组无法对其进行系统的分析，故我们选取了最有代表，也最关键的

#pragma acc parallel loop

#pragma acc parallel loop reduction(operation:var)

这两条指令进行分析。其指令的作用就是将下一条语句进行并行处理，后者还指出了可以 reduction 的变量。

我们工具达到的效果就是分析给定的一段代码后，自动在代码中一定可以并行的地方加入指令；如果无法判断一处是否可以并行，也会输出相应的注释，去帮助 OpenACC 的使用者更佳方便地优化自己的程序。

2.2 工具情况

我们的工具能够检测大部分情况下的并行情况，但也有部分情况受制于小组水平与静态分析本身的缺陷，无法检测是否并行。在接下来的分析中，可以看到我们的工具是有安全性的保证的。而准确性的不足，也会在后面的部分详细给出。

2.3 简化 C 语言

我们的代码分析都是在简化之后的 C 语言上进行的。之所以需要简化 C 语言，是因为由于程序静态分析的特殊需要，我们小组写了个针对 C 语言的解释器，还因为这可以简化我们的工具的制作。但值得注意的是，针对其中的每一条简化，我们都有相应的原因作为支撑，以表明这是一个合理的简化。

Table 1: 简化 C 语言

简化	原因
不支持任何形式的&运算	为了提高对于数组分析的精度，不得不对指针做出一点限制
所有的变量定义要放在一个代码块的开头	早期的 C 语言标准就是如此，并且这可以简化解释器的编写
忽略#define语句	这可以简化解释器的编写
不允许调用函数	暂时做的是过程内的分析，但可特殊处理 min,max 等简单函数
不允许一个语句中修改多个变量	算法上支持，目的仅为简化解释器的编写

3 算法与安全性证明

3.1 算法简述

首先，将整个程序 parser 成一个 CFG 图，之后在这个 CFG 上做程序流分析：逆向做一遍程序流分析，分析在该语句之后，有哪些变量是活跃变量；正向做一遍程序流分析，得到每个 for 循环与表达式的摘要，具体摘要的定义将在接下来给出。对于每个循环，我们通过对这两次分析的结果分析其能否被并行。

- 活跃变量分析：

  1) 逆向分析
  2) 半格元素：一个集合，这个操作之前，哪些变量是活跃的。
  3) 交汇操作：集合的并集
  4) 变换函数

  var 被修改， $v_1, v_2, \ldots, v_m$ **被使用** $KILL = \{var\}, GEN = \{v_1, v_2, \ldots, v_m\}$

- 语句摘要分析：

  0) 摘要：5 个集合

Array_Modify 表示在这个语句中哪些数组被修改，除了记录数组之外，还需要记录其被修改的位置 $(i_1, i_2 \ldots, i_d)$，其中 d 表示数组的维数，i_x 为一个表达式。

Array_Use 表示在这个语句中哪些数组被使用，同Array_Modify的记录一样，需要记录数组与其被使用的位置。

Var_Modify 表示在这个语句中有哪些普通变量被修改，如果语句是 if,while,for，那么这里的变量指的是在语句之外的变量，而不考虑内部的局部变量。

Var_Use 表示在这个语句中有哪些普通变量被使用，同Var_Modify的记录一样，对 if,while,for 语句特殊处理。

Var_Reduction 表示在这个语句中有哪些普通变量被修改了，并且记录对它们的操作满足结合律，即可以 reduction，集合元素就是var, op。

1) 正向分析
2) 半格元素：摘要的 5 个集合
3) 交汇操作：摘要集合的并集（流非敏感）
4) 变换函数

if,while,for 递归处理，将内部的语句的摘要进行合并，并剔除那些在语句块中的局部变量，作为对这些语句的摘要。

其他普通语句 将相应的变量（数组变量和普通变量）放入对应的集合之中。并且分析每个被改变的变量的操作是否满足结合律，如果是，也放入Var_Reduction集合之中。

- for 循环并行分析：
对于可能可以并行的 for 循环其内部而言，只关注其变量能否并行，变量分为数组变量和普通变量，for 内部的局部变量不纳入考虑。

数组 对于每个数组 a 而言，我们去该 for 语句的摘要中查找其可能被访问或者修改的位置。

a) 如果 a 只被修改，则改数组不影响并行性。
b) 如果 a 只被访问，则改数组不影响并行性。
c) 如果 a 既被修改，也被访问，那么需要知道修改的位置和访问的位置是否有重合的部分。如果有，则不能并行，反之则可以。在这里可以调用 SMT Solver 求解（详情见 3.3 准确性的不足）。但是在工具中我们只考虑最简单的可以并行的情况：即该数组的所有下标均是仅由单个循环变量构成的线性表达式，并且访问和修改处的每个下标构成的表达式完全相等的情况（具体而言：如 a[i][j+1]=a[i][j+1]+1，访问的位置为 (i,j+1)，修改的位置也为 (i,j+1)，其中该访问和修改处的每个下标构成的表达式完全相等）。（安全性证明见 3.4 安全性证明）

普通变量 对于每个变量 var 而言，仅需考虑其在 for 之外的影响。

a) 如果 var 只被修改，要看在下文中，这个变量是否是活跃的。因为并行的副作用就是导致这个变量在 for 之行结束后的值不确定，如果活跃，则认为这个 for 不能并行。
b) 如果 var 只被访问，则改变量不影响并行性。

c) 如果 var 既被修改，也被访问，需要从摘要中得到这个变量是否是 reduction 的，如果是，则这个变量不影响并行性；如果不是，则不能并行。

特殊情况 如果 for 循环中出现了 break,continue,return 等语句，则直接认为不能并行。

3.2 安全性证明

逆向活跃变量分析是完全基于课上所提及的方法，故正确性证明在此略去。这里只对语句摘要分析和 for 循环并行分析做安全性的证明。

1) 语句摘要分析：可以看出这里的摘要分析是流非敏感的，这是对 for 循环能并行的一个充分条件。因为 for 循环能够并行，等价于说对于循环变量 i，无论 i 的执行顺序如何，最后对于外部数组和普通变量的计算结果不发生改变。这里的流非敏感分析将条件加强为，对于循环变量 i，无论 i 的执行顺序如何，无论循环内部的执行顺序如何，最后对于外部数组和普通变量的计算结果不发生改变。正因为是充分条件，所以这个抽象是安全的。

2) 循环并行分析：每个改变和使用的数组的下标必须是一个循环变量的线性表示，这一条件是一个充分条件。首先，单个循环变量的线性表示，保证了在这个循环之中，数组被修改到的位置仅被访问一次，不会出现需要原子操作 (atomic) 的情况。其次，且对于每个数组 a，对于所有被访问的下标 $pair(i_1, j_1, k_1 \dots)$ 和对于所有被修改的下标 $pair(i_2, j_2, k_2 \dots)$ 要求完全相同，这个条件也是一个充分条件，这保证了所有被修改的位置和所有被使用的位置不会有交集。

3.3 准确性的不足

1) 摘要分析的一个不准确性在于做的是流非敏感的分析，主要受制于 for 循环分析的复杂性。但反过来看，如果一个语句不在任何 for 循环之中，那么无论在 CFG 上的位置如何（在不考虑活跃变量分析时），均不会对后续的分析起到任何作用。所以准确性的不足只体现在 for 循环内部的语句中，因此我们组认为这种准确性不足相比于算法的复杂是更可以接受的。

2) 循环并行分析之中，在处理数组下标的问题上，我们的项目工具加了很强的限制，即每个改变和使用的数组的下标必须是一个循环变量的线性表示，且对于每个数组被访问位置的下标要求完全相同。通过使用 SMT Solver 可以提高准确性。为此可以形式化地写出这个问题的两个方面：

   a) 设一个被修改的数组在此循环中的某一维下标（为了简化问题，我们把下标拆开来考虑）的表达式集合是 $Expr_1, Expr_2, \dots, Expr_n$，并且循环变量是 i，那么其每个下标仅访问一次的充分必要条件为

   $$\forall j, k \in [1, n],\ i_1, i_2 \in \text{for 循环中所有取值},\ i_1 \neq i_2\ st. Expr_j(i_1) \neq Expr_k(i_2)$$

   这可以通过向 SMT Solver 添加 assert 语句查询逆命题能否满足，如果 SMT Solver 回答可以满足，即存在一个 i_1, i_2 的取值使得任意两个表达式有交集，则不能并行；反之则可以。

b) 设一个数组 a 被修改的位置在此循环中的某一维下标（为了简化问题，我们把下标拆开来考虑）的表达式集合是 $Mdf_1, Mdf_2, \ldots, Mdf_n$；被使用的位置在此循环中的某一维下标的表达式集合为 $Use_1, Use_2, \ldots, Use_m$。那么其修改和访问的位置不会相交的充分必要条件为

$$\forall j \in [1,n],\ k \in [1,m],\ i_1, i_2 \in \text{for 循环中所有取值},\ i_1 \neq i_2\ st. Mdf_j(i_1) \neq Use_k(i_2)$$

也通过向 SMT Solver 添加 assert 语句查询逆命题能否满足，如果 SMT Solver 回答可以满足，即存在一个 i_1, i_2 的取值使得两个表达式有交集，则不能并行；反之则可以。

4 程序运行和安全性检查

在 demo 文件夹下 make 即可看到 demo。

在 demo/samples 中，我们写了若干份 c 代码用于测试。

使用 py2 或 py3 文件夹中的 python 代码：运行 analysis.py cpu.c gpu.c，即可将 cpu 代码转为用于 OpenACC 的 gpu 代码。

安全性检查：编译以后，我们通过分别运行 cpu 和 gpu 代码，查看输出结果是否一致。

整个项目都是用 python3.6 完成的。

由于实验室的机器上只有 Python2.x，所以我们使用 Python 的 3to2 工具把我们的代码转换成 Python2 的代码以后放在 py2 文件夹供测试用。我们实际写的代码在 py3 这个文件夹中。

具体的使用可以参考 README.md。

5 小组分工

吴昊泽同学负责 OpenACC 平台的搭建，编写 C 代码的 parser 和 CFG 图的构建，算法讨论。

张煜皓同学负责 llvm 的接口尝试，实现基于 CFG 图的数据流分析算法，算法讨论。

陈牧歌同学负责对最后工具的测试，算法讨论。

report.pdf 由三人共同完成。

6 Used tools

pgcc 16.10-0 64-bit target on x86-64 Linux -tp haswell

Tesla K80 x 2

GeForce GTX x 1