Приведение гнезд циклов к тесно-вложенному виду

Н.А. Катаев

Институт прикладной математики им. М.В. Келдыша

1. Введение

Преобразование программы является неотъемлемой составляющей современного оптимизирующего компилятора. При этом многие компиляторы используют внутреннее представление отличное от исходной программы (GCC GIMPLE, LLVM IR), которое последовательно меняется, чтобы улучшить программу в соответствии с заранее заданным критерием (время работы, объем занимаемой памяти). Об исходной программе компиляторы обычно забывают за исключением отдельных точек соприкосновения между исходным кодом и низкоуровневым представлением, описываемых с помощью отладочной информации. Этой информации, в целом, достаточно, чтобы определить свойства исходной программы. Но использовать ее для восстановления кода на высокоуровневом языке программирования, близкого к коду исходной программы, невозможно. Однако, необходимость выполнять преобразования на уровне исходного кода в системе САПФОР [1] с целью устранения проблем мешающих распараллеливанию или с целью оптимизации полученной программы в модели DVM [2,3] остается.

В связи с этим преобразования в системе САПФОР были разделены на две группы:

- Преобразования, необходимые для исследования свойств программы.
- Преобразования, необходимые для ее распараллеливания.

О преобразованиях первой группы могут выполняться на уровне LLVM IR, что существенно упрощает их реализацию и позволяет использовать реализованные в LLVM [4] проходы. Нет необходимости преобразовывать исходную программу в соответствии с этими преобразованиями.

Ко второй группе относятся преобразования, которые тем или иным образом меняют свойства исходной программы, делая ее пригодной к распараллеливанию. Для реализации данных преобразований используются возможности Clang [5]. Но этих возможностей оказывается недостаточно для проверки допустимости преобразований и в этом случае анализ допустимости выполняется на двух уровнях абстракции Clang AST и LLVM IR согласованно.

Система DVM позволяет получить значительный выигрыш от распараллеливания тесно вложенных гнезд циклов (tightly nested loops, perfectly nested loops). В таких гнездах циклов телом внешнего цикла является внутренний цикл, никакие операторы не должны располагаться между двумя такими циклами. Многомерные гнезда циклов могут быть отображены на многомерную решетку процессоров в соответствии с распределением данных. Такая возможность хорошо согласуется со структурой решаемых физических задач, в которых используются сеточные методы, и позволяет получить максимальный параллелизм при расчетах по каждому измерению сетки. Использование тесно вложенных циклов оказывается полезным и при использовании технологии OpenMP. Во многих случаях использование указания collapse позволяет повысить уровень параллелизма и лучше загрузить вычислительные процессоры, что положительно сказывается на эффективности распараллеливания.

Приведение гнезд циклов к тесно-вложенному может быть реализовано в системе САПФОР. Преобразования выполняются над for-циклами, представленными в канонической форме [6, раздел 2.6], так как только такие циклы могут быть распараллелены как с помощью DVM, так и с помощью OpenMP. Приведение циклов к каноническому виду выходит за рамки данного преобразования, но может быть реализовано отдельным проходом в системе САПФОР. Данное преобразование состоит из четырех этапов:

- Определение не тесно вложенных циклов программы.
- Определение того, что цикл представлен в каноническом виде.
- Проверка допустимости преобразования.
- Выполнение преобразования.

Первый и последний этапы могут быть выполнены исключительно средствами Clang. Второй и третий этап представляют наибольший интерес с точки зрения реализации, так как должны

совместно использовать LLVM IR и Clang AST для анализа гнезд циклов. Остановимся более подробно на проверке допустимости преобразования.

2. Проверка допустимости преобразования

Для проверки допустимости преобразования гнезда циклов к тесно-вложенному виду понадобятся результаты предварительно выполненных анализов достигающих определений и живых переменных [6, глава 9]. Данные анализы реализованы в САПФОР отдельными проходами и выполняются над низкоуровневым представлением LLVM IR. Анализ достигающих определений реализован в виде анализа потока данных [7, глава 9], оператор сбора и передаточная функции имеют вид:

- $MRD_{in}(S) = \bigcap_{t \in pred(S)} MRD_{out}(t)$
- $MRD_{out}(S) = MRD_{in}(S) \cup DEF(S)$

В данном случае S – базовый блок [7, глава 8], $pred(S)$ – множество базовых блоков, из которых есть переходы на S, $DEF(S)$ – множество участков памяти, в которые записывается значение в базовом блоке. В процессе анализа рассматриваются не переменные как таковые, а участки памяти, фигурирующие в дереве псевдонимов. Множества $MRD_{in}(S)$ и $MRD_{out}(S)$ описывают участки памяти, которые содержат некоторое значение на входе и выходе из базового блока S соответственно. В алгоритме также вычисляется множество $USE(S)$ – множество участков памяти, которые используются в базовом блоке S и получают значение до входа в этот базовый блок.

Анализ потока данных выполняется в прямом направлении, начиная с самых внутренних циклов. При переходе к анализу объемлющего цикла, внутренний цикл рассматривается как одна вершина в графе потока данных, а проведенный для него анализ обобщается. Для этого используются следующие формулы:

- $USE_b(L) = \bigcup_{t \in body(L)} \big(USE(t) - MRD_{in}(t)\big)$
- $DEF_b(L) = \bigcap_{t \in exits(body(L))} MRD_{out}(t)$
- $MAYDEF_b(L) = \bigcup_{t \in body(L)} (MAYDEF(t) \cup DEF(t))) - DEF_b(L)$

В данном случае L – естественный цикл, $body(L)$ – множество базовых блоков, образующих тело цикла, $exits(body(L))$ – множество базовых блоков, переходы из которых являются выходами из цикла. Множество $USE_b(L)$ будет содержать участки памяти, которые используясь в цикле, получают значение либо на предыдущей итерации, либо до цикла. Префикс MAY – признак достоверности модификации участков памяти. Участки памяти, входящие в соответствующие множества могут получить значение, а могут и не получить.

Анализ живых переменных также реализован в виде анализа потока данных. Анализ выполняется в обратном направлении, оператор сбора и передаточная функция выглядят следующим образом:

- $LIVE_{out}(S) = \bigcup_{t \in succ(S)} LIVE_{in}(t)$
- $LIVE_{in}(S) = USE(S) \cup (LIVE_{out}(S) - DEF(S))$

В отличие от анализа достигающих определений гнездо циклов анализируется сверху вниз. Сначала рассматривается внешний цикл, при этом каждый внутренний цикл рассматривается как одна вершина в теле внешнего цикла. Графом потока самого верхнего уровня является граф управления всей программы. Отдельное внимание стоит уделить заданию граничных условий:

- $LIVE_{in}(S) = LIVE_{in}(L) \cup LIVE_{out}(L)$
- $S \in (exits\big(body(L)\big) \cup latch(body(L))$

В данном случае $exits(body(L))$ – множество базовых блоков, переходы из которых являются выходами из цикла, $latch(body(L))$ – множество базовых блоков, переходы из которых ведут к заголовку цикла, $LIVE_{in}(L)$ и $LIVE_{out}(L)$ – множества живых участков памяти на входе в и выходе из вершины, соответствующей циклу в теле внешнего цикла.

Основной этап заключается в проверке базовых блоков, расположенных между заголовками внешнего и внутреннего циклов, и базовых блоков, расположенных между базовыми блоками, меняющими счетчики внутреннего и внешнего циклов. Обозначим множество данных базовых блоков как $motion(L)$, где L – внешний цикла. Будем считать, что инициализация счетчика и вычисление нижней границы внутреннего цикла вынесено в отдельный базовый блок,

предшествующий заголовку внутреннего цикла. Обозначим данный блок как *Preheader* (на Рис. 1 таким блоком является entry). Будем считать, что данный блок не входит в $motion(L)$. В разделе проверка каноничности цикла говорится о том, как можно определить инструкции, отвечающие за инициализацию счетчика цикла. На основе этой информации LLVM IR будет преобразован, чтобы выделить *Preheader*. Учитывая, что преобразованию подвергаются только for-циклы в канонической форме, выделение *Preheader* всегда возможно.

```
1. for (int I = 0; I < N; ++I);
```

```
1.  entry:
2.    store i32 0, i32* %I
3.    br label %for.cond
4.  for.cond:                                     ; preds = %for.inc, %entry
5.    %0 = load i32, i32* %I
6.    %1 = load i32, i32* %N
7.    %cmp = icmp slt i32 %0, %1
8.    br i1 %cmp, label %for.body, label %for.end
9.  for.body:                                     ; preds = %for.cond
10.   br label %for.inc
11. for.inc:                                      ; preds = %for.body
12.   %2 = load i32, i32* %I
13.   %inc = add nsw i32 %2, 1
14.   store i32 %inc, i32* %I
15.   br label %for.cond
16. for.end:                                      ; preds = %for.cond
```

Рис. 1. Структура for-цикла в представлении LLVM IR.

Цель проверки допустимости преобразования определить, что инструкции, расположенные в блоках из $motion(L)$, могут быть внесены в тело внутреннего цикла с сохранением семантики программы. Проверка опирается на исследование обращений к участкам памяти. Зная соответствие между переменными программы в Clang AST и участками памяти в LLVM IR, и выполнив данную проверку над LLVM IR, можно гарантировать, что она также будет выполнена на уровне Clang AST.

Обозначим как $exits(motion(L))$ базовые блоки, переходы из которых ведут к базовым блокам, не входящим в множество $motion(L)$. Используя результаты анализа достигающих определений можно вычислить следующие множества (по аналогии с соответствующими множествами для тела цикла):

- $USE_{motion}(L) = \bigcup_{t \in motion(Outer)} \big(USE(t) - MRD_{in}(t)\big)$
- $DEF_{motion}(L) = \bigcap_{t \in exits(motion(L))} MRD_{out}(t)$
- $MAYDEF_{motion}(L) = \bigcup_{t \in motion(L)} (MAYDEF(t) \cup DEF(t))) - DEF_{motion}(L)$

Тогда для допустимости преобразования двух циклов (*Outer* – внешний и *Inner* – внутренний циклы) необходимо, чтобы

(1) $USE_{motion}(Outer) \cap (DEF_b(Inner) \cup MAYDEF_b(Inner)) = \emptyset$

(2) $USE_b(Inner) \cap \big(DEF_b(Inner) \cup MAYDEF_b(Inner)\big) \cap (DEF_{motion}(Outer) \cup MAYDEF_{motion}(Outer)) = \emptyset$

(3) $USE_{motion}(Outer) \cap (DEF_{motion}(Outer) \cup MAYDEF_{motion}(Outer)) = \emptyset$

(4) $USE(Preheader) \cap (DEF_{motion}(Outer) \cup MAYDEF_{motion}(Outer)) = \emptyset$

(5) базовые блоки из $motion(Outer)$ не должны содержать выходы из внешнего цикла и переходы из них не должны идти в обход внутреннего цикла.

Первое условие означает, что если некоторая инструкция использует участок памяти, получающий значение вне блоков из $motion(Outer)$, то данный участок памяти не должен меняться на итерации внутреннего цикла. В противном случае после внесение данной инструкции в тело внутреннего цикла она будет использовать значения, посчитанные на предыдущей итерации внутреннего цикла, что не соответствует исходной версии программы.

Второе условие означает, что если некоторая инструкция в теле внутреннего цикла использует участок памяти, получающий значение на предыдущей итерации внутреннего цикла, то этот участок памяти не должен меняться внутри блоков из $motion(Outer)$. В противном случае после выполнения преобразования данная инструкция уже не будет использовать значение с предыдущей итерации внутреннего цикла.

Третье условие означает, что если некоторая инструкция (в блоке из $motion(Outer)$) использует участок памяти, получающий значение вне блоков из $motion(Outer)$, то выполняющиеся после нее инструкции в блоках из $motion(Outer)$ не должны менять это значение. В противном случае после внесения инструкций в тело внутреннего цикла используемый участок памяти будет заново вычисляться на каждой итерации цикла. Гарантировать только за счет средств статического анализа, что после каждого такого вычисления он будет получать одно и то же значение в общем случае не возможно.

Четвертое условие говорит о том, что для вычисления начального значения не должны использоваться участки памяти, получающие значения в базовых блоках из $motion(Outer)$. В противном случае после внесения данных блоков в тело внутреннего цикла, начальное значение может быть посчитано неверно.

Еще одно условие является дополнительным и проверяется только в том случае, если нет уверенности, что как минимум одна итераций внутреннего цикла выполняется до конца (то есть выполняется блок, отвечающий за изменение счетчика цикла). В этом случае необходимо убедиться, что участки памяти, изменяемые в блоках из $motion(L)$ не используются на чтение после выхода из внешнего цикла:

$$(6)\ LIVE_{out}(Outer) \cap (DEF_{motion}(Outer) \cup MAYDEF_{motion}(Outer)) = \emptyset$$

Пересечение множеств во всех случая понимается как отсутствие двух участков памяти, принадлежащих разным множествам и пересекающихся между собой.

Литература

1. Бахтин В.А., Жукова О.Ф., Катаев Н.А., Колганов А.С., Крюков В.А., Поддерюгина Н.В., Притула М.Н., Савицкая О.А., Смирнов А.А. Автоматизация распараллеливания программных комплексов // Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции (19-24 сентября 2016 г., г. Новороссийск), М.: ИПМ им. М.В.Келдыша, 2016, С. 76-85.
2. Коновалов Н.А., Крюков В.А., Михайлов С.Н., Погребцов А.А. Fortran DVM - язык разработки мобильных параллельных программ», Ж. "Программирование", № 1, 1995, С. 49-54.
3. Бахтин В.А., Клинов М.С., Крюков В.А., Поддерюгина Н.В., Притула М.Н., Сазанов Ю.Л. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами. – Вестник Южно-Уральского государственного университета, серия "Математическое моделирование и программирование", №18 (277), выпуск 12 – Челябинск: Издательский центр ЮУрГУ, 2012. С. 82-92
4. The LLVM Compiler Infrastructure. URL: http://llvm.org/
5. "clang" C Language Family Frontend for LLVM. URL: https://clang.llvm.org/
6. OpenMP Application Programming Interface, Version 4.5, November 2015 URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
7. Ахо А.В., Лам М.С., Сети Р., УльманДж.Д. Компиляторы: принципы, технологии и инструментарий, 2-е издание: Пер. с англ.- М.ООО "И.Д. Вильямс", 2008. – 1184 с.