Применимость тайлинга для компиляции редукций в языке C[]

А. Калинов, А. Ластовецкий, И. Ледовских, М. Посыпкин

1. Введение

Вычисления над массивами данных широко распространены. В языках типа FORTRAN, C, PASCAL и многих других такие вычисления записываются с помощью циклов. В связи с распространенностью подобного рода вычислений, ряд современных языков FORTRAN90, FORTRAN95, ZPL [ZPL], C[] [Cbr, Cbr1] помимо обычных операций над данными скалярного типа также имеют операции над массивами. Эти операции мы будем называть векторными операциями, а выражения и операторы в которых они фигурируют – векторными выражениями и операторами соответственно.

Векторные операции позволяют с одной стороны более компактно выражать алгоритмы, содержащие вычисления над массивами, а с другой стороны – предоставляют компилятору больше информации о структуре алгоритма, которую можно использовать при генерации оптимального кода.

Операции над массивами делятся на поэлементные, когда операция применяется к соответствующим элементам массивов одинаковых размерностей, и редукции, когда одна и та же операция применяется к элементам или группам элементов некоторого массива, при этом размерность результата операции меньше размерности операнда. Примером редукции может служить операция SUM в FORTRAN90, которая вычисляет сумму элементов либо подмассивов некоторого массива.

В процессе компиляции произвольное векторное выражение заменяется последовательностью **базовых** векторных операторов, каждый из которых затем транслируется в одну систему вложенных циклов.

Важным часто встречающимся классом базовых векторных операторов является **простой редуктивный оператор** (более подробно это понятие рассматривается в разделе “Синтаксис C[] и понятие редуктивного выражения”). В редуктивных операторах результат редукции используется повторно в получаемой системе вложенных циклов. В подобной ситуации можно заметно ускорить выполнение программы, если предотвратить, либо уменьшить вытеснение элементов результата из кэша. С этой целью применяется тайлинг [Lam, Wolfe]. Суть этого преобразования заключается в

увеличении степени вложенности циклов с одновременным уменьшением числа повторений внутренних циклов. Результаты экспериментов подтверждают эффективность этого оптимизирующего преобразования для реализации редуктивных выражений.

Анализ корректности применения тайлинга для произвольной системы вложенных циклов не является тривиальной задачей [Lam, Wolfe]. В [Wolfe] доказывается критерий применимости тайлинга к системам идеально вложенных циклов. В данной работе показывается, что система вложенных циклов, получаемая в результате компиляции любого простого редуктивного оператора, удовлетворяет этому критерию. Это означает, что при применении тайлинга к таким системам вложенных циклов анализ корректности не требуется, что позволяет существенно упростить алгоритм генерации кода для редуктивного оператора.

2. Синтаксис C[] и понятие редуктивного выражения

Язык C[] [Cbr, Cbr1], является строгим расширением ANSI C. Основным новым понятием, введенным в C[], является понятие **вектора**. Вектор определяется как упорядоченная последовательность значений одного произвольного типа. Принципиальным отличием вектора от массива является то, что элементы вектора вовсе не обязаны регулярно располагаться в памяти как элементы массива. Элементами вектора также могут быть вектора.

В языке Си массив в арифметических выражениях преобразуется к типу “указатель на тип элемента”. Поэтому в языке C[] предусмотрена операция, которая предотвращает преобразование массива к указателю и, тем самым, позволяет использовать его в арифметических выражениях как единое целое – **операция вырезки**.

Операция вырезки предназначена для обеспечения доступа к сегментам массивов. Она имеет следующий синтаксис:

```
e [l : r : s],
```

где *e* – выражение, обозначающее массив, а выражения *l, r, s* имеют целый тип и обозначают левую границу, правую границу и шаг вырезки соответственно. При этом, *e [l : r : s]* обозначает вектор из *(r-l)/s + 1* элементов, где *i*-м элементом этого вектора является *e[l + i * s]*. Например, если массив *a* состоит из пяти элементов, то выражение *a[2 : 4 : 2]* обозначает вектор длины *2* , элементами которого являются *a[2]* и *a[4]*. С помощью вырезки можно получать доступ к различным совокупностям элементов не только одномерных, но и многомерных массивов.

Если шаг вырезки равен *1*, то его можно не указывать, второе двоеточие тоже при этом можно опустить. Правая и левая границы также могут быть опущены.

Опущенная левая граница считается равной *0*, а опущенная правая считается равной *N-1*, где *N* – число элементов массива, к которому применяется вырезка. Таким образом, значением выражения *a[:]* является вектор, состоящий из всех элементов массива *a*.

Выражение, образованное применением нескольких операций вырезки к обозначению массива, будем называть **тривиальным векторным выражением**.

Семантика некоторых арифметических операций языка Си и операций присваивания расширена таким образом, что их операндами и результатами могут быть векторные значения. В этом случае, соответствующая операция применяется поэлементно. Например, код для нахождения поэлементной суммы массивов *a* и *b* с помещением результата в массив *c* может иметь следующий вид:

```
double a[N], b[N], c[N];
...
c[:] = a[:] + b[:];
```

В данном случае, векторный оператор *c[:] = a[:] + b[:];* эквивалентен следующему циклу:

```
for(i = 0; i < N; i ++)
c[i] = a[i] + b[i];
```

Некоторым бинарным операциям языка Си поставлены в соответствие редуктивные операции по следующему правилу: если выражение *e* является *N*-элементным вектором, а ω – некоторая бинарная операция, то соответствующая ω редуктивная операция обозначается через $[\omega]$, а результат ее применения к *e* определяется следующим образом: $[\omega]e = (\ldots((e_0\ \omega\ e_1\ \omega\ e_2)\ldots\ \omega\ e_N)$, где $e_0, e_1, \ldots, e_N$ – компоненты вектора *e*. Простейшим примером применения редуктивной операции может служить код, вычисляющий сумму строк матрицы:

```
double A[M][N], a[N];
...
a[:] = [+] A[:][:];
```

Введем понятие **ранга** вектора. Если элементы вектора не являются векторами, то ранг вектора полагаем равным *1*. Если элементы вектора являются векторами размерности *n*, то ранг вектора полагается равным *n + 1*. Рангом векторного выражения *E*, обозначаемого через *rank(E)*, называется ранг вектора, являющегося его значением.

Также нам понадобится понятие **терминального элемента вектора**. Если элементами вектора являются объекты невекторного типа, то они называются его терминальными элементами. Если элементами вектора *v* являются вектора,

то терминальными элементами вектора *v* будут все терминальные элементы его элементов.

3. Компиляция векторных выражений

Выражение, содержащее векторные операции, называется **векторным выражением**. Векторные выражения можно условно разделить на два класса – **поэлементные** и **редуктивные**. Поэлементным называется векторное выражение, не содержащее редуктивных операций. Выражение вида $[\omega]\ \Psi$, где Ψ – поэлементное выражение, будем называть редуктивным. Например, *a[:] + b[:]* – поэлементное выражение, а *[*] (a[:] + b[:])* – редуктивное.

Редуктивным оператором называется оператор вида $\Phi = \Psi$, где Φ – тривиальное векторное выражение, а Ψ – редуктивное выражение. **Поэлементным оператором** называется оператор вида $\Phi = \Psi$, где Φ – тривиальное векторное выражение, а Ψ – поэлементное выражение. Например, оператор *a[:] = [+] A[:][:];* является редуктивным.

При вычислении любого векторного выражения, очевидно, происходит чтение из памяти и, возможно, запись в память некоторых данных. Если векторное выражение не содержит операций присваивания и операций с побочными эффектами, то при вычислении такого выражения не производится запись, а производится только чтение данных из памяти.

Множество данных, которые записываются или считываются при вычислении данного векторного выражения Φ, будем называть **памятью, используемой данным выражением** и обозначать через $\mu(\Phi)$.

Векторное выражение присваивания $\Phi = \Psi$ будем называть **базовым**, если $\mu(\Phi) \cap \mu(\Psi) = \varnothing$. Аналогично, **базовым оператором** будем называть оператор, состоящий из одного базового выражения. Базовое выражение называется простым, если выражения Ψ и Φ не содержат операций с побочным эффектом [Cstd]. **Простым оператором** будем называть оператор, состоящий из одного простого выражения.

На первом этапе трансляции компилируемое векторное выражение заменяется эквивалентной последовательностью базовых векторных операторов, каждый из которых является либо редуктивным, либо поэлементным. Например, выражение *a[:] = [+] A[:][:] * [+] B[:][:]* заменяется следующей последовательностью операторов:

```
tmp1[:] = [+]A[:][:];
tmp2[:] = [+]B[:][:];
a[:] = tmp1[:] * tmp2[:];
```

На последующих этапах трансляции для каждого базового оператора генерируется своя система вложенных циклов. Любой базовый оператор может

быть вычислен с помощью одной системы вложенных циклов. Для поэлементного базового оператора $E = F;$ эта система циклов имеет вид:

```
for(i1 = 0; i < N1; i1 ++)            /* цикл 1*/
 for(i2 = 0; i2 < N2; i2 ++)          /* цикл 2*/
       ...
       ...
for(in = 0; in < Nn; in++)            /* цикл n */
       E[i1, i2,..., in] = F[i1, i2,..., in];
```

где $rank(E) = rank(F) = n$, а $E[i_1,i_2,...,i_n]$ и $F[i_1,i_2,...,i_n]$ – терминальные элементы векторов, являющихся значениями выражений E и F с соответствующими индексами.

Рассмотрим базовый редуктивный оператор

```
E= [ω] F;
```

где F – поэлементное выражение ранга n, E – вектор ранга $\underline{n-1}$, а $[\omega]$ – редуктивная операция. Для вычисления данного редуктивного оператора требуется вычислить значение выражение F, а затем применить к нему редуктивную операцию $[\omega]$. Это можно сделать с помощью следующей системы вложенных циклов:

```
for(i1 = 0; i1< N1; i1 ++)             /* цикл 1*/
 for(i2 = 0; i2 < N2; i2 ++)          /* цикл 2*/
...
       ...
       for(in = 0; in < Nn; in ++)    /* цикл n */
            if(i==0)
                E[i2,...,in] = F[i1,i2,...,in];
            else
                E[i2,...,in] ω= F[i1,i2,...,in];
            ...
```

Проиллюстрируем описанную схему компиляции на примере редуктивного оператора $a[:]=[+]A[:][:]$. Данный оператор можно вычислить с помощью следующей системы вложенных циклов:

```
for(i = 0;i < M;i ++)              /* цикл 1 */
for(j = 0;j < N;j ++)             /* цикл 2 */
if(i == 0)
       a[j] = A[i][j];
else
       a[j] += A[i][j];
```

Заметим, что данные по ссылке $a[i]$ повторно используются в цикле 1. Большое число итераций цикла 2 может препятствовать повторному использованию ссылки $a[j]$, так как к моменту повторного использования данные по ссылке $a[j]$

могут быть вытеснены из кэша. Поэтому целесообразно применить тайлинг этого цикла, в результате чего получится следующий код:

```
for(J = 0; J < N; J += T)            /* цикл c2 */
 for(i = 0; i <M; i ++)               /* цикл 1 */
  for(j = J; j < min(J + T, N);j ++) /* цикл 2 */
  if(i == 0)
       a[j] = A[i][j];
  else
       a[j] += A[i][j];
```

Аналогично, в случае произвольного редуктивного оператора, тайлинг циклов 2,...,n применяется для увеличения числа попаданий в кэш для ссылки $E[i_2,...,i_n]$, которая повторно используется в цикле 1. В следующем разделе будет показана корректность такого преобразования для системы циклов, полученной в результате компиляции произвольного простого редуктивного оператора.

4. Применимость тайлинга к простому редуктивному оператору

В начале этого раздела мы рассмотрим модель итерационного пространства [Wolfe, Som97], которую обычно используют для анализа применимости тайлинга и других преобразований к системе идеально вложенных циклов. Затем покажем, как можно применить данную модель для случая циклов, получающихся в результате компиляции редуктивного оператора.

Пусть имеется система из n идеально вложенных циклов типа *for*:

```
for(i1 = 0; i1 < N1; i1 ++)
 for(i2 = 0; i2 < N2; i2 ++)
       ...
  for(in = 0; in < Nn ; in ++) {
                          /* тело системы
                             вложенных циклов */
               }
```

Итерационным пространством данной системы вложенных циклов называется пространство Z^n. Тогда данной системе циклов ставится в соответствие множество всех наборов $\boldsymbol{a}$ из Z^n, таких что $0 \le a_1 < N_1$, $0 \le a_2 < N_2$, …, $0 \le a_n < N_n$. Каждой итерации данной системы циклов ставится в соответствие вектор из этого множества, компоненты которого совпадают со значениями соответствующих индексов. Такой вектор называется итерационным вектором данной итерации. Для удобства мы будем говорить “итерация $\boldsymbol{a}$” вместо “итерация с итерационным вектором $\boldsymbol{a}$”.

На итерационном пространстве Z^n можно ввести **лексикографический порядок**: >. Для пространства Z^1 он совпадает с обычным порядком на

множестве целых чисел. Предположим, что лексикографический порядок определен для пространств Z^i, $1 < i < n$. Тогда будем говорить, что вектор ***b*** из Z^n **лексикографически больше,** чем вектор ***a*** из Z^n, и писать $\boldsymbol{b} > \boldsymbol{a}$, если $b_1 > a_1$ либо одновременно $b_1 = a_1$ и $(b_2,\ldots,b_n) > (a_2,\ldots,a_n)$. Из определения лексикографического порядка следует, что итерация ***b*** выполняется после итерации ***a*** тогда и только тогда, когда вектор $\boldsymbol{b} > \boldsymbol{a}$ в лексикографическом смысле.

Если итерация ***a*** обращается к некоторому адресу памяти, итерация ***b*** обращается к этому же адресу памяти, и при этом хотя бы одно из этих обращений является записью данных, то будем говорить, что итерации ***a*** и ***b*** имеют **зависимость по данным**. Согласно условию Бернстайна [Bernstein], преобразование кода программы будет корректным, если порядок выполнения итераций, имеющих зависимость по данным, сохраняется в результате преобразования.

Геометрическим выражением зависимости по данным является понятие **дистанционного вектора** [Lam, Wolfe]. Если итерации ***a*** и ***b*** имеют зависимость по данным и $\boldsymbol{b} > \boldsymbol{a}$, то говорят, что имеется **дистанционный вектор $\boldsymbol{d} = \boldsymbol{b} - \boldsymbol{a}$ данной системы вложенных циклов**.

Применимость тайлинга для произвольной системы вложенных циклов исследована в работах [Lam, Wolfe]. Имеют место следующие две теоремы [Wolfe]:

Утверждение 1 (Wolfe). Циклы с *i*-го по *j*-й могут быть переставлены в любом порядке, если для любого дистанционного вектора ***d*** данной системы вложенных циклов $(d_1, \ldots, d_{i-1}) > \boldsymbol{0}$ или $d_k \geq 0$ для всех i, $i \leq k \leq j$.

Циклы, для которых выполнены условия утверждения 1, называются **полностью переставляемыми**.

Утверждение 2 (Wolfe). Тайлинг циклов с *i*-го по *j*-й является корректным преобразованием тогда и только тогда, когда эти циклы полностью переставляемы.

С помощью этих двух утверждений мы докажем следующую теорему:

Теорема 1. Ко всем циклам системы вложенных циклов, полученной в результате компиляции любого простого редуктивного оператора, может быть применен тайлиг.

Доказательство. Пусть имеется некоторый простой редуктивный оператор

```
E = [ω] F;
```

где F – поэлементное выражение ранга n, E – вектор ранга $n-1$, а $[\omega]$ – редуктивная операция.

Система вложенных циклов, вычисляющая данный редуктивный оператор, имеет следующий вид:


```
for(i1 = 0; i1 < N1; i1 ++)              /* цикл 1*/
 for(i2 = 0; i2 < N2;i2 ++)              /* цикл 2*/
                    ...
  for(in = 0; in < Nn; in ++)            /* цикл n*/
      if(i1 == 0)
            E[i2,...,in] = F[i1,i2,...,in];
      else
            E[i2,...,in] ω= F[i1,i2,...,in];
                    ...
```

Так как редуктивный оператор $E = [\omega]\ F;$ является простым, то вектора E и F не пресекаются в памяти и выражение $F[i_1, i_2, \ldots, i_n];$ не содержит операций записи в память. Поэтому, зависимость по данным будут иметь итерации, у которых левые части имеют один и тот же адрес в памяти. Этому условию удовлетворяют итерации, итерационные вектора которых отличаются только в первой компоненте. Таким образом, множество дистанционных векторов для рассматриваемой системы вложенных циклов состоит из векторов $(k, 0,\ldots, 0)$, где $1 \leq k < n$.

Условие утверждения 1 выполнено для данной системы вложенных циклов. Следовательно, все циклы системы полностью переставляемы, и, согласно утверждению 2, к ним может быть применен тайлинг. Теорема доказана.

6. Результаты экспериментов

На основе доказанной теоремы в компиляторе языка C[] реализован алгоритм компиляции редуктивных выражений, использующий тайлинг. Выбор оптимального размера тайла не рассматривается в данной работе.

Результаты экспериментов показывают эффективность тайлинга при генерации кода для редуктивных операторов. Для разных редуктивных операторов на различных платформах данный метод позволяет получать 5-25% ускорения.

Для иллюстрации рассмотрим задачу нахождения суммы строк матрицы $N_1 \times N_2$. Соответствующий редуктивный оператор имеет вид: *a[:] = [+] A[:][:];*. В таблице 1 приводятся значения ускорения, полученного за счет применения тайлинга для платформ Sparc Station 5 и Ultra 1.

Платформа	**Кэш**	$\boldsymbol{N_1 \times N_2}$	**Ускорение**
Sparc 5	8 Kb	*1000 × 1000* *1500 × 1500*	1.2 1.16
Ultra 1	16 Kb	*1000 × 1000* *1500 × 1500*	1.03 1.08

Таблица 1. Ускорения для оператора *a[:] = [+] A[:][:];* .

6. Заключение

Результат, доказанный в данной работе, позволяет не производить проверку корректности при применении тайлинга к системам циклов, полученных в результате компиляции простых редуктивных операторов. Это существенно упрощает алгоритм генерации кода для таких операторов. Результаты экспериментов подтверждают эффективность тайлинга для оптимизации редуктивных операторов.

Литература:

[CbrSpec] The C[] Language Specification. http://www.ispras.ru/~cbr/cbrsp.html

[Bernstein] A. J. Bernstein. Program analysis for parallel processing. IEEE Transactions on Electronic Computers, EC-15(5):757-762, Oct. 1966.

[Cstd] ANSI. Programming Language C. X3.159-1989 American National Standard Institute.

[Cbr] S. Gaissaryan, and A. Lastovetsky. ANSI C superset for vector and superscalar computers and its retargetable compiler. Journal of C Language Translation, 5(3):183-198, 1994.

[Som97] S.Ghosh, M.Martonosi, S. Malik. Cache miss equations: an analitical representation of cache misses. Proceedings of the 11th ACM Conference on Supercomputing, Vienna, Austria, July 1997.

[Cbr1} S. Katserov, A. Lastovetsky,S. Gaissaryan, D. Khaletsky, I. Ledovskih. Retargetable compiler for ANSI C superset for vector and superscalar computers. *Proceedings of Second International Conference on Software for Multiprocessors and Supercomputers. Theory, Practice, Experience,* Moscow, Russia, September 1994.

[ZPL] Calvin Lin and Lawrence Snyder. ZPL: An Array Sublanguage. *Languages and Compilers for Parallel Computing*, pp. 96-114, 1993.

[Lam] M. Wolfe, M. Lam. A data locality optimizing algorithm., in *ACM SIGPLAN conference on PLDI, June 1991.*

[Wolfe] M.E. Wolfe, Improving locality and parallelism in nested loops, Ph.D. dissertation, August 1992, Stanford University