ISSN 2079-3316 **ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ** **т. 9**, №4(39), с. 493–508

ББК 32.973.2
УДК 004.4' 24

А. В. Климов

К автоматическому порождению программ трафаретных вычислений с улучшенной временной локальностью

Аннотация. Трафаретные (stencil) алгоритмы широко используются в задачах математического моделирования на регулярных сетках, эволюции клеточных автоматов (типа игры «жизнь»), обработки изображений, анализа последовательностей и т.п. Такие алгоритмы хорошо параллелятся, но обычные подходы к распараллеливанию имеют низкую временную локальность, что ограничивает их масштабируемость. Избавление от этого недостатка возможно при использовании различных схем переупорядочения обработки точек, когда пространство разбивается на небольшие области, помещающиеся в кэш, в которых удается продвинутся сразу на несколько итераций. Однако, такие программы трудно писать и отлаживать. Есть несложный метод пирамид, но он плохо масштабируется, поскольку влечет дублирование вычислений.

Наш подход состоит в использовании более сложных схем переупорядочения без дублирования, для которых код может генерироваться автоматически из относительно несложной спецификации схемы. При этом сами схемы задаются путем назначения функций распределения вычислительных узлов по пространству и времени. В статье излагается подход, и рассматриваются на простом примере различные варианты кода, порождаемые по различным функциям распределения.

Ключевые слова и фразы: трафаретные алгоритмы, параллельные вычисления, автоматизация распараллеливания, временная локальность, кэш-память, метод пирамид, потоковая модель вычислений, планирование вычислений, распределение вычислений.

Введение

Трафаретные (stencil) алгоритмы широко используются в задачах математического моделирования на регулярных сетках, эволюции клеточных автоматов (типа игры «жизнь»), обработки изображений, анализа последовательностей и т.п. В них итерационно вычисляемые значения в точках пространства зависят от значений в некоторой

Работа выполнена при поддержке РФФИ (гранты №№ 17-07-00324, 17-07-00478).

© А. В. Климов, 2018
© Институт проблем проектирования в микроэлектронике РАН, 2018
© Программные системы: теория и приложения (дизайн), 2018

10.25209/2079-3316-2018-9-4-493-508

heat1.c

```
1         for (t=1; t<=M; t++) {
2           for (i=1; i<N; i++)
3             B[i] = 0.33333*(A[i-1]+A[i]+A[i+1]);
4           for (i=1; i<N; i++)
5             A[i] = B[i] ;
6           }
```

Рис. 1. Простой трафаретный алгоритм Heat1 на языке C

окрестности на предыдущей итерации. В англоязычной литературе их называют Iterative Stencil Loops (ISL) [1]. Примером может служить следующий код на C (рис. 1).

Такие алгоритмы хорошо параллелятся, но обычные подходы к распараллеливанию упираются в низкую временную локальность. *Временная локальность* это свойство программы, которое характеризуется вероятностью того, что некое обращение к памяти в ближайшем будущем будет выполняться по тому же адресу, куда было выполнено последнее обращение. Обратная к ней величина выражает ожидаемое число обращений по другим различным адресам до следующего обращения по тому же адресу. Если эта величина не превосходит размер кэша, то с большой вероятностью данная ячейка будет удержана в кэше до нового обращения к ней. Таким образом, временная локальность характеризуется тем объемом кэша (меньше – лучше), при котором попадания в кэш начинают существенно превалировать над промахами. Или, при фиксированном размере кэша – долей попаданий (точнее, тем, насколько эта доля близка к 1).

В трафаретной программе на каждой итерации прочитывается весь объем данных задачи, тогда как объем вычислений, приходящихся в среднем на один элемент считанных данных, невелик (в данном примере – 3 Flop). И если весь объем данных не помещается в кэш, то производительность ограничивается не арифметикой, а пропускной способностью канала связи процессора с памятью (так называемая *стена памяти*).

Для улучшения временной локальности могут применяться различные схемы переупорядочения обработки точек: пространство разбивается на небольшие области, помещающиеся в кэш, для которых удается продвинутся сразу на несколько итераций. Тогда, с одной стороны, увеличивается объем вычислений на одно читаемое из памяти значение, а с другой – многие промежуточные значения «проживают» свою жизнь в кэше вообще без записи в память. Однако, такие программы трудно писать и отлаживать. Есть относительно несложный

классический метод пирамид [2–4], но он приводит к дублированию части вычислений. Объем этого дублирования быстро растет с ростом высоты пирамиды (числа объединяемых ярусов) и числа процессоров, что также мешает полноценному масштабированию.

Наш подход состоит в использовании более сложных схем переупорядочения без дублирования, для которых код может генерироваться автоматически из относительно несложной спецификации схемы [5]. В качестве языка описания алгоритма используется язык потоковой модели вычислений с индексированием, где порядок вычислений ограничен только наличием зависимостей по данным. Такая форма представления программы может быть получена автоматически из классической последовательной программы (при соблюдении некоторых ограничений). Далее, задавая тот или иной порядок выполнения узлов, можно автоматически порождать различные программы, в том числе параллельные.

Порядок предлагается задавать посредством назначения функций распределения вычислительных узлов по пространству и времени. В статье будет изложен общий подход, и на простом будет показано, как по разным функциям распределения могут порождаться различные варианты кода. Данный подход может быть применен и в более широком классе программ, построенных в виде линейных циклов (где границы циклов и индексные выражения линейны). Для статьи выбраны трафаретные программы, поскольку они имеют простую структуру и позволяют наглядно показать позитивный результат применения метода.

Статья имеет следующую структуру. В разделе 1 описывается потоковая модель вычислений, в которой должна быть выражена исходная программа. Для представления может использоваться графический язык, который иллюстрируется и разъясняется на рабочем примере в разделе 2. В разделе 3 вводится понятие о функциях распределения, а в разделе 4 демонстрируется на рабочем примере их использование для порождения вариантов кода с улучшенной временной локальностью. В разделе 5 обсуждаются особенности нашего подхода в сравнении с работами других авторов.

1. Потоковая модель вычислений с явным индексированием

В потоковой модели вычисления разбиваются на семантически мотивированные фрагменты, выполнение которых организуется по принципу готовности данных: когда все входные данные некоторого фрагмента готовы, он может выполняться, и его выполнение приводит к вычислению новых данных, которые посылаются на входы других

фрагментов в форме сообщений, называемых токенами. Таким образом, программа задает только вычислительный граф (зависимостей по данным), а порядок выполнения выстраивается автоматически в динамике.

В нашей версии потоковой модели вычислений важную роль играют индексы. Описывая в программе узел, мы фактически задаем семейство экземпляров узла, различаемых индексами и называемых также виртуальными узлами. В описании узла задается список имен индексов (обычно – целочисленных). В посылаемом токене помимо значения данного указывается имя узла и конкретные значения всех индексов. Всякий виртуальный узел (с конкретными индексами) будет активирован, когда на каждый его вход придет хотя бы один токен. Пока этого не случилось, пришедшие токены ожидают прихода недостающих токенов в памяти процессора сопоставления (ППС) [**6**]. В момент активации все участвующие в ней токены (по одному на вход) удаляются из ППС, а информация из них переходит в пакет, передаваемый в исполнительное устройство (ИУ) для выполнения программы узла. Выполнение может быть отложено до появления свободного ИУ. В программе узла могут содержаться команды порождения и отправки новых токенов на другие виртуальные узлы. Это — стандартный сценарий. Есть и другие возможности, но в примерах данной статьи они не используются.

2. Рабочий пример

В качестве рабочего примера рассмотрим задачу решения нестационарного уравнения теплопроводности классическим методом явной схемы. Чтобы упростить изложение будем рассматривать одномерный случай с использованием простой трехточечной схемы. Это не сильно снижает общность, поскольку таковой может служить моделью и для многомерной задачи с одномерной декомпозицией. Соответствующий код на языке C уже был приведен на рис. 1.

По этому коду автоматически строится потоковый граф алгоритма, в графической форме ([**7**]) показанный на рис. 2. Граф получен в предположении, что $M \geqslant 1$ и $N \geqslant 2$, входом является весь массив `A[0:N]`, а выходом – его часть `A[1:N-1]`.

На каждый оператор присваивания в потоковом графе обычно возникает отдельный узел, имя которого образуется из имени переменной в левой части. У него столько входов, сколько вхождений переменных (возможно, с индексами) есть в правой части, и один выход. На каждый вход идут стрелки от выходов тех узлов, значение которых может читаться в этом вхождении, в начале стрелки ставится условие такого

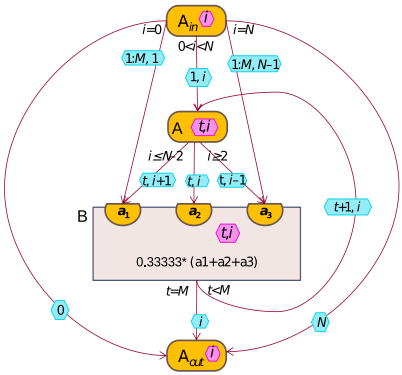


Рис. 2. Потоковый граф алгоритма Heat1

чтения. Исключение составляют операторы переприсваивания, типа A=B, порождающие *транзитные* узлы, которые могут быть либо устранены, либо перестроены. Здесь мы поставили узел A (сдвинув его на 1 по оси t) перед узлом B, чтобы не дублировать развилку. Также вводятся отдельные узлы для входных и выходных значений – здесь это $\mathrm{A_{in}}$ и $\mathrm{A_{out}}$.

Все узлы здесь имеют индекс из одного или двух компонентов – по числу циклов, объемлющих соответствующий оператор. При записи виртуальных узлов индексы записываются в угловых скобках, а на схеме – в шестиугольниках. Обычный узел изображается прямоугольником с одним или несколькими именованными входами, имеющими форму усеченного желтого круга на краю прямоугольника. Если узел транзитный (одновходовый и без вычислений), то он может быть изображен просто овалом того же цвета.

Граф-программа работает следующим образом. На входные узлы $\mathrm{A_{in}}\langle i\rangle$ поступают токены, содержащие начальное состояние, с i от 0 до N. Когда токены есть на всех входах виртуального узла, возникает его активация. При этом участвующие в ней токены стираются, а узел вычисляет результат, зависящий от входов и индекса (если задана формула, иначе это просто значение первого входа), и посылает токены с этим результатом по исходящим стрелкам. Каждый экземпляр узла A_{in} подает свое значение через узел-посредник A на три входа узла B в зависимости от условий на индекс i. Условия вписаны у

оснований стрелок. У конца стрелки указан также индекс получателя на фоне голубого шестиугольника. Выражения в условии и индексе вычисляются в контексте отправителя.

Обратим внимание на стрелки с условиями $i = 0$ и $i = N$. В них индекс t задан с диапазоном $1 : M$. Это множественный токен, посылаемый виртуально на все узлы-получатели с индексом t в указанном диапазоне. Но формально это один токен с кратностью – либо бесконечной (`##`), либо заданной числом (`#M`). Здесь кратность `#M` определяется диапазоном `1:M`.

Каждый узел $\mathrm{B}\langle t, i\rangle$, получив три входных значения, вычисляет результат по своей формуле и передает его узлу $\mathrm{A}\langle t, i\rangle$, увеличивая на 1 индекс t. Узел A раздает значение снова на три входа узла B, меняя должным образом индекс i. При $t = M$, узел $\mathrm{B}\langle t, i\rangle$ подает результат на выходной узел $\mathrm{A}_{\mathrm{out}}\langle i\rangle$.

3. Функции распределения

В соответствии с потоковой моделью вычислений каждый узел может выполняться, когда доставлены все его аргументы. Будем говорить, что узел Y зависит от узла X, обозначая это как $X \to Y$, если узел X создает токен, используемый в активации Y. Транзитивное замыкание этого отношения (обозначаемое как $\twoheadrightarrow$) создает базовый (строгий) частичный порядок на множестве всех виртуальных узлов, которые возникают в данной задаче. Любой линейный порядок, уточняющий этот базовый порядок, является допустимым. Параллелизм это тоже некоторый частичный порядок, уточняющий базовый. Разным уточненным порядкам соответствуют разные программы вычисления данного потокового графа.

Параллельные программы, помимо порядка, характеризуются и способом распределения вычислительных узлов по вычислительным ядрам (процессорам) или тредам. Его можно задать функцией распределения по пространству Π, которая зависит от узла (как семейства) и его индексов и вырабатывает номер ядра или треда. Аргумент-узел будем записывать нижним индексом (опуская, если он несуществен или ясен из контекста), а индексы узла – как обычные аргументы: $\Pi_A(t, i)$.

Аналогично, порядок вычислений можно задать функцией распределения по времени Θ, которая также зависит от узла и его индексов и выдает номер этапа, на котором этот узел надо выполнять.

Составление законченной программы обхода потокового графа (последовательной или параллельной) тоже является способом описания порядка его вычислений. Но мы считаем, что составить программу обхода на некотором языке программирования человеку сложнее,

чем выписать функцию распределения. Поэтому считаем актуальной задачу автоматической генерации программы по заданным функциям распределения по пространству и времени.

Часто может рассматриваться многомерное пространство процессоров/тредов, когда процессоры расположены в узлах многомерной решетки, и номер каждого задается вектором целых чисел. В этом случае функция Π вырабатывает векторные значения. Аналогично время можно считать многомерным, задавая фукнцию Θ как векторную. В таком случае подразумевается лексикографический порядок. Мы также будем использовать комбинированную многомерную функцию Ψ, все компоненты которой поделены на *временные* и *пространственные*. Последние будем выделять подчеркиванием.

На множестве значений функции Ψ определим *частичный лексикографический порядок* $\succ$ следующим образом. Для сравнения двух векторов надо найти первый (слева) компонент, в котором они различаются. Если этот компонент временной, то значения сравнимы и больше тот вектор, у которого этот компонент больше. В противном случае значения несравнимы (равные также считаем несравнимыми).

Функция Ψ (также как и функция Θ) должна быть монотонной. Это значит, что при наличии связи

$$\mathrm{A}\langle I\rangle \twoheadrightarrow \mathrm{B}\langle J\rangle$$

должно выполняться:

$$\Psi_{\mathrm{B}}(J) \succ \Psi_{\mathrm{A}}(I).$$

Из двух отдельных функций Π и Θ можно составить функцию Ψ двумя способами: как $\langle \underline{\Pi}, \Theta\rangle$ и как $\langle \Theta, \underline{\Pi}\rangle$. В первом случае мы имеем набор независимых асинхронных параллельных процессов, а во втором – процессы, синхронизуемые глобальным барьером после каждого этапа. В первом случае относительный во времени порядок выполнения разных процессов может (и должен) ограничиваться асинхронно фактическими связями между узлами. Попросту говоря, одни процессы будут ожидать прихода токенов от других процессов.

Если задана функция Ψ, то функции Π и Θ можно получить стиранием, соответственно, всех временных или всех пространственных компонентов.

4. Порождение программ по функциям распределения

Наш подход состоит в том, что функции распределения задаются пользователем – программистом или отдельным экспертом по распралалеливанию, а специальный компилятор порождает в соответствии с ними программу (последовательно-параллельную)

обхода вычислительного графа. Чтобы предложить хорошие функции распределения (обеспечивающие хорошие распаралеливание, эффективность и масштабируемость), такой эксперт должен, конечно, понимать взаимосвязи внутри алгоритма.

Простейший вариант функций распределения для примера на рис. 2 это $\Pi(t,i) = 0$ и $\Theta_{\mathrm{B}}(t,i) = \langle t,i\rangle$. (Для транзитных узлов собственное распределение задавать не будем, чтобы не загромождать изложение). Это значит, что для всех узлов используется только один процессор, и все узлы B выполняются упорядоченно по возрастанию номера итерации t, а внутри итераций по возрастанию индекса i. Эта схема реализуется, например, простым кодом, показанным на рис. 1.

Рассмотрим задачу порождения такого или подобного кода автоматически из графа алгоритма и заданной функции распределения. Во-первых, нам понадобится описание множества (индексов) всех исполнямых виртуальных узлов B. Здесь это многогранник

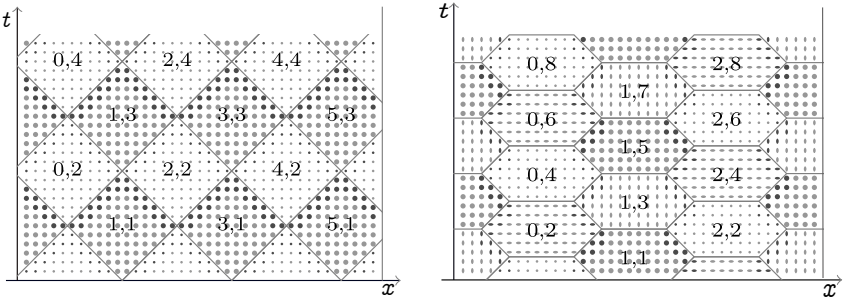
$$D_{\mathrm{B}} = \{(t,i) | 1 \leqslant i \leqslant N-1 \wedge 1 \leqslant t \leqslant M\}.$$

Оно либо задано изначально (при порождении графа из кода на C эти области также порождаются), либо его придется определять по заданным областям для входных узлов (что непросто, но решаемо, хотя и не всегда). В программе в соответствии с этими областями для каждого входа каждого узла заводится массив с размерностью, равной размерности узла. Будем называть их виртуальными.

Далее генерируется [8] код обхода областей в порядке, предписанном функцией Θ. Внутрь вставляется операция узла, которая берет элементы из своих входных массивов и помещает результат во входные массивы адресатов. Это уже работоспособный код, но в нем используются огромные виртуальные массивы (здесь двумерные) с однократным присваиванием в каждый элемент.

Дальше включается важный механизм построения *отображения* (он является предметом будущих исследований), которое экономно отобразит все виртуальные масссивы задачи на реальные массивы в памяти процессоров. На этом этапе будет выявлено, что три входа можно без дублирования хранить в одном массиве, и что достаточно хранить значения только двух смежных итераций. Это будет реализовано либо как два одномерных массива, либо как один двумерный массив размера 2 по измерению t. Копирование одного массива в другой целесобразно исключить, меняя их ролями между итерациями.

Как выше было отмечено, данная реализация имеет низкую временную локальность, связанную с тем, что на каждой итерации весь массив A прочитывается целиком. Чтобы этого избежать, будем



(*а*) $a = (x + t)/d$, $b = (x - t)/d$, $d = 12$

(*б*) $a = (x + t_1)/d$, $b = (x - t_1)/d$, $t_1 = m(t/h) + t \bmod h + (k/2)$, $k = 8$, $h = 3$, $m = k + h - 1$, $d = 2m$

Рис. 3. Варианты распределений вычислительных узлов по пространству ($\Pi = a + b$) и времени ($\Theta = a - b$) в задаче Heat1

использовать другие способы обхода вычислительного графа, при которых для части массива выполняются сразу несколько итераций.

Например, можно обходить блоками, показанными на рис. 3*а*,3*б*. Под каждым вариантом указаны свои формулы для величин $a = a(t, x)$ и $b = b(t, x)$, участвующих в построении функций распределения $\Pi(\mathrm{t}, \mathrm{x}) = a + b$ и $\Theta(\mathrm{t}, \mathrm{x}) = a - b$ (деление всюду целочисленное). Эти функции и порождают разбиение на блоки. Каждый блок это множество узлов $\mathrm{B}\langle t, x\rangle$, имеющих одинаковые значения функций Π и Θ (они показаны внутри блоков).

Для удобства восприятия узлы в разных блоках отрисованы по-разному, причем в алгоритме порождения рисунка для этого честно использованы указанные функции. Нетрудно убедиться, что при вычислении каждого блока вся необходимая внешняя информация уже вычислена ранее, то есть при меньших значениях функции Θ.

Вариант рис. 3*а* задают такие функции распределения узлов $\mathrm{B}\langle t, x\rangle$:

$$\Pi(t, x) = p(t, x) = \lfloor (x + t)/d \rfloor + \lfloor (x - t)/d \rfloor$$
$$\Theta(t, x) = s(t, x) = \lfloor (x + t)/d \rfloor - \lfloor (x - t)/d \rfloor$$

Подразумевается, что внутри блоков выполняется обход как в простейшем варианте. Но можно задать обход и явно, например определив комбинированную векторную функцию Ψ:

$$\Psi(t, x) = \langle s(t, x), \underline{p(t, x)}, t, x\rangle,$$

heat1a.c

```
1     for (s=0; s<=S_Max; s++)
2       #pragma omp parallel for
3       for (p=-1; p<=P_Max; p++}
4          if ((s+p)%2==0)
5            for (t=max(1,6*s-5); t<=min(M,6*s+5); t++)
6              for (int x=max(1,max(6*p+6*s-t,6*p-6*s+t));
7                       x<=min(N-1,min(6*p+6*s-t+11,6*p-6*s+t+11)); x++)
8                 A[t,x] = 0.33333*(A[t-1,x-1]+A[t-1,x]+A[t-1,x+1]);
```

Рис. 4. Блочный последовательно-параллельный вариант согласно рис. 3*а* с общим двумерным массивом

где функции s и p определены выше.

В таком случае будет порождена последовательно-параллельная программа (рис. 4), которая для каждого *этапа* s параллельно выполняет блоки, приписанные разным *местам* p, а внутри блоков порядок соответствует исходному. Очевидно, что этот код обладает заметно более высокой временной локальностью, чем вариант на рис. 1.

Эффективность данного кода можно улучшить, если цикл по t разбить на два: где $t < 6 * s$ и где $t \geq 6 * s$, что упростит границы цикла по x. Этого же эффекта можно добиться, если в определении функции распределения Ψ перед p вставить $\lfloor 2t/d \rfloor$.

Для избавления от огромного общего двумерного массива `A` выполним отображение, склеив все четные (по t) слои в один слой и все нечетные в один слой. Формально: отобразим элемент `A[t,x]` на элемент `A1[t%2+2*x]` одномерного массива `A1` размера $2 * N + 2$. Нетрудно доказать, что такое склеивание корректно при любом допустимом порядке: никакие склеиваемые ячейки никогда не используются одновременно. Темным цветом на рис. 4 показаны точки, значения которых сохраняются при переходе от этапа к этапу (при реализации в MPI эти участки будут пересылаться между процессами).

Выполнив вручную указанные преобразования, мы измерили производительность полученного кода на настольном компьютере с 4-ядерным процессором Intel Core i7-2600 3.4GHz. Использовался компилятор GCC 6.3.0 (под Windows в рамках MINGW) с опциями `-O2 -fopenmp`. Тестировались два варианта:

(*1*) исходный как на рис. 1 с распараллеливанием обоих внутренних циклов,

(*2*) преобразованный.

Были использованы следующие значения параметров: $N = 2000000$, $M = 5000$, $d = 300$ (вместо $d = 12$ на рис. 3*а*). Параметр N выбран так, чтобы объем данных задачи не помещался в кэш L3.

Таблица 1. Масштабирование (GFlops)

Число тредов:	1	2	3	4	5	6	7	8
Вариант 1:	0.99	1.07	1.07	1.07	1.05	1.05	1.00	0.92
Вариант 2:	3.33	6.56	9.55	10.80	10.55	11.72	12.35	13.55

Результаты при различном числе тредов приведены в таблице 1. Исходный вариант демонстрирует очень низкую производительность при полном отсутствии распараллеливания из-за работы на пределе скорости доступа к памяти. Улучшенный вариант показывает более чем втрое большую скорость на одном треде и хорошее масштабирование до 3-х тредов, которое дальше ослабляется.

Заметим, что для получения такого ускорения ключевую роль сыграло отображение, при котором результаты итерации t записывались поверх результатов итерации $t-2$. В противном случае (если бы использовались большие двумерные массивы) все промежуточные значения рано или поздно все равно пришлось бы выталкивать из кэша в основную память, несмотря на то, что они уже не нужны.

Однако, посыл и вклад статьи не в том, что мы написали более эффективный код для данной задачи, а в том, что намечен путь для порождения таких кодов в автоматическом режиме по указанию подходящей функции распределения. Если для одномерного случая написать такой код руками было несложно, то для аналогичного 2-мерного случая это уже гораздо труднее, а для 3-мерного почти невозможно. Но машине и компилятору с несложной подсказкой от человека это будет вполне под силу.

5. Аналогичные работы

За последние десятилетия вышло много работ [9–12], авторы которых решают задачу автоматического порождения оптимального расписания (schedule) и размещения (placement) для параллельного выполнения узлов потокового графа. При этом исходят из линейной (аффинной) структуры графа (семейства его вершин и ребер описываются линейными функциями и ограничениями), а сами расписание и размещение ищут в форме линейных функций. Также при этом большое внимание уделяется скорости работы алгоритмов построения расписаний [13], поскольку придется в чисто автоматическом режиме обрабатывать большие объемы кода.

Вопросы генерации кода для расписаний и размещений, заданных аффинными функциями также хорошо проработаны [11][12][8]. Однако, функции распределения по пространству и времени (те же placement и

schedule) могут выбираться из более широкого класса функций (чем чисто аффинные), например, они могут содержать в себе деление с остатком (причем не только на заключительной стадии как в [9]), и, возможно, логические операции над двоичными числами. В частности, обсуждаемые в данной статье функции распределения не являются чисто аффинными. При этом требования столь разнообразны, а разнообразие возможных решений столь велико, что не представляется возможным описать их формально для алгоритмов оптимизации. С другой стороны, структуры вычислительных графов написанных человеком программ не очень сложны и вполне доступны их автору для выбора хороших функций распределения. Во всяком случае это проще, чем описывать обход графа в виде вложенных циклов. Важно, что ошибки в записи функций распределения могут в худшем случае привести к не очень эффективному коду (или отказу в его порождении), но не могут нарушить правильность алгоритма (при условии, что генератор кода корректен).

В работах [1][14] авторы акцентируют внимание на выборе хороших способов распределения, дающих высокую эффективность. Свои решения они реализуют вручную. Наличие инструмента автоматической генерации может помочь перебирать и экспериментально сравнивать большое число разных решений.

Непроцедурный язык НОРМА [15] автоматически и притом достаточно эффективно транслируется в разные параллельные платформы (OpenMP, MPI, CUDA и их комбинации). При этом также используются линейные функции расписания, индуцируемые так называемой параллельно-ярусной формой, которая допускает параллелизм только в пределах каждого яруса. Для трафаретных алгоритмов ярусами будут отдельные итерации. Предоставление пользователю возможности задавать расписания более широкого класса могло бы дать дальнейшее улучшение эффективности генерируемых кодов.

6. Заключение

Целью работы на данном этапе было продемонстривать возможность и принципы автоматического синтеза параллельных программ по спецификации в форме потокового графа и задаваемым пользователем распределением по пространству и времени в форме функции распределения. Такой синтез возможен прежде всего для программ трафаретного типа, а также для более широкого класса программ из циклов со статической структурой по управлению (static control program). Осуществление такого синтеза потребует наличия развитой библиотеки разрешающих процедур в области целочисленной линейной алгебры и логики.

Список литературы

[1] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, M. D. Santambrogio. “A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops”, *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD’16, ACM, New York, NY, USA, 2016, pp. 77:1–77:8. doi ↑494,504

[2] В. А. Вальковский. “Параллельное выполнение циклов. Метод пирамид”, *Кибернетика*, **19**:5 (1983), pp. 51–55. ↑495

[3] Д. Л. Головашкин, А. В. Кочуров. “Решение сеточных уравнений на графических вычислительных устройствах. Метод пирамид”, *Вычислительные технологии*, **17**:3 (2012), pp. 55–69. URL ↑495

[4] С. А. Малышева, Д. Л. Головашкин. «Реализация разностного решения уравнения Максвелла на графическом процессоре методом пирамид», *Компьютерная оптика*, **40**:2 (2016), с. 179–187. doi ↑495

[5] А. В. Климов. «О парадигме универсального языка параллельного программирования», *Языки программирования и компиляторы-2017. Труды конференции*, PLC–2017 (3–5 апреля 2017), ЮФУ, Ростов-на-Дону, 2017, с. 141–146. * ↑495

[6] N. N. Levchenko, A. S. Okunev, D. N. Zmeev. “Approaches to the development of various sets of nodes and blocks for the PDCS matching processor”, *Proceedings of IEEE East-West Design & Test Symposium*, EWDTS’2017 (Novi Sad, Serbia, Sept. 29–Oct. 2, 2017). doi ↑496

[7] А. В. Климов, А. С. Окунев. «Графический потоковый метаязык для асинхронного распределенного программирования», *Проблемы разработки перспективных микро- и наноэлектронных систем*. Т. II, МЭС-2016 (3–7 октября 2016 года), ИППМ РАН, М., 2016, с. 151–158. * ↑496

[8] C. Bastoul. “Efficient code generation for automatic parallelization and optimization”, *Proceedings of the Second International Conference on Parallel and Distributed Computing*, ISPDC’03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 23–30. doi ↑500,503

[9] P. Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”, *International Journal of Parallel Programming*, **21**:6 (1992), pp. 389–420. doi ↑503,504

[10] P. Feautrier. “Toward automatic distribution”, *Parallel Processing Letters*, **4**:3 (1994), pp. 233–244. doi ↑503

[11] M. Griebl. *Automatic parallelization of loop programs for distributed memory architectures*, Habilitation thesis, University of Passau, 2004. doi URL ↑503

[12] U. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*, PhD thesis, The Ohio State University, 2008. URL ↑503

[13] P. Feautrier. “Scalable and structured scheduling”, *International Journal of Parallel Programming*, **34**:5 (2006), pp. 459–487. doi ↑503

[14] R. Strzodka, M. Shaheen, D. Pajak, H.-P. Seidel. “Cache accurate time skewing in iterative stencil computations”, *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP’11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 571–581. ↑504

[15] А.Н. Андрианов, Т.П. Баранова, А.Б. Бугеря, К.Н. Ефимкин. «Трансляция непроцедурного языка НОРМА для графических процессоров», *Препринты ИПМ им. М.В.Келдыша*, 2016, 073, 24 с. ↑504

Поступила в редакцию 03.07.2018
Переработана 19.12.2018
Опубликована 30.12.2018

Рекомендовал к публикации *д.ф.-м.н. С. В. Знаменский*

Пример ссылки на эту публикацию:

А. В. Климов. «К автоматическому порождению программ трафаретных вычислений с улучшенной временной локальностью». *Программные системы: теория и приложения*, 2018, **9**:4(39), с. 493–508.

10.25209/2079-3316-2018-9-4-493-508

http://psta.psiras.ru/read/psta2018_4_493-508.pdf

Об авторе:

Аркадий Валентинович Климов

Научные интересы: нетрадиционные модели и языки (параллельного) программирования (потоковые, функциональные, порождающие и т.п.); методы анализа, преобразования и порождения программ; математические алгоритмы.

0000-0002-7030-1517
e-mail: arkady.klimov@gmail.com

UDC 004.4' 24

Arkady Klimov. *Towards automatic generation of stencil programs with enhanced temporal locality.*

Abstract. Stencil algorithms are widely used in areas of mathematical modeling on regular grids, the evolution of cellular automata (such as the game "life"), image processing, sequence analysis, etc. Such algorithms are well parallelized, but the usual approaches to parallelization have low temporal locality, which limits their scalability. It is possible to get rid of this drawback by using different re-ordering schemes for processing points, when the space is divided into small areas that fit into the cache, in which it is possible to advance by several iterations at once. However, such programs are difficult to write and debug. There is a simple pyramid method, but it doesn't scale well due to some duplication of calculations.

Our approach is to use more sophisticated reordering schemes without duplication, for which code can be generated automatically from a relatively simple scheme specification. In this case, the schemes themselves are defined by specifying distribution functions that distribute the computational points over space and time. This article outlines the approach and considers, for a sample source, various code variants generated from different distribution functions. (*In Russian*).

Key words and phrases: stencil algorithms, parallel computing, auto-parallelization, temporal locality, cache memory, pyramid method, dataflow computation model, placement, scheduling.

2010 *Mathematics Subject Classification*: 97P40; 97P50, 97R40

References

[1] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, M. D. Santambrogio. "A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops", *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD'16, ACM, New York, NY, USA, 2016, pp. 77:1–77:8. ↑494,504

[2] V. A. Val'kovskiy. "Parallel execution cycles. Method of the pyramids", *Kibernetika*, **19**:5 (1983), pp. 51–55 (in Russian). ↑495

[3] D. L. Golovashkin, A. V. Kochurov. "Solving finite-difference equations on GPU. The pyramid method", *Vychislitel'nyye tekhnologii*, **17**:3 (2012), pp. 55–69 (in Russian). URL ↑495

[4] S. A. Malysheva, D. L. Golovashkin. "Implementation of the FDTD algorithm on GPU using a pyramid method", *Komp'yuternaya optika*, **40**:2 (2016), pp. 179–187 (in Russian). ↑495

[5] A. V. Klimov. "On the paradigm of a universal parallel programing language", *Yazyki programmirovaniya i kompilyatory-2017. Trudy konferentsii*, PLC–2017 (3–5 aprelya 2017), YuFU, Rostov-na-Donu, 2017, pp. 141–146 (in Russian). ↑495

© A. V. Klimov, 2018
© Institute for Design Problems in Microelectronics, 2018
© Program Systems: Theory and Applications (design), 2018

10.25209/2079-3316-2018-9-4-493-508

[6] N. N. Levchenko, A. S. Okunev, D. N. Zmeev. “Approaches to the development of various sets of nodes and blocks for the PDCS matching processor”, *Proceedings of IEEE East-West Design & Test Symposium*, EWDTS’2017 (Novi Sad, Serbia, Sept. 29–Oct. 2, 2017). ↑496

[7] A. V. Klimov, A. S. Okunev. “A graphical dataflow meta-language for asynchronous distributed programming”, *Problemy razrabotki perspektivnykh mikro- i nanoelektronnykh sistem.* V. II, M·YeS-2016 (3–7 oktyabrya 2016 goda), IPPM RAN, M., 2016, pp. 151–158 (in Russian). ↑496

[8] C. Bastoul. “Efficient code generation for automatic parallelization and optimization”, *Proceedings of the Second International Conference on Parallel and Distributed Computing*, ISPDC’03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 23–30. ↑500,503

[9] P. Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”, *International Journal of Parallel Programming*, **21**:6 (1992), pp. 389–420. ↑503,504

[10] P. Feautrier. “Toward automatic distribution”, *Parallel Processing Letters*, **4**:3 (1994), pp. 233–244. ↑503

[11] M. Griebl. *Automatic parallelization of loop programs for distributed memory architectures*, Habilitation thesis, University of Passau, 2004. ↑503

[12] U. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*, PhD thesis, The Ohio State University, 2008. ↑503

[13] P. Feautrier. “Scalable and structured scheduling”, *International Journal of Parallel Programming*, **34**:5 (2006), pp. 459–487. ↑503

[14] R. Strzodka, M. Shaheen, D. Pajak, H.-P. Seidel. “Cache accurate time skewing in iterative stencil computations”, *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP’11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 571–581. ↑504

[15] A. N. Andrianov, T. P. Baranova, A. B. Bugerya, K. N. Efimkin. “Nonprocedural NORMA language translation for GPUs”, *Keldysh Institute preprints*, 2016, 073, 24 pp. ↑504

Sample citation of this publication:

Arkady Klimov. “Towards automatic generation of stencil programs with enhanced temporal locality”. *Program Systems: Theory and Applications*, 2018, **9**:4(39), pp. 493–508. (*In Russian*).

10.25209/2079-3316-2018-9-4-493-508

http://psta.psiras.ru/read/psta2018_4_493-508.pdf