Автоматическое распараллеливание последовательных программ для гибридной системы с ускорителем на основе потока данных

Арк.В. Климов, Н.Н. Левченко, А.С. Окунев, А.Л. Стемпковский

Институт проблем проектирования в микроэлектронике РАН

Рассматривается модель вычислений на основе потока данных и ее реализация в виде многоядерного сопроцессора - ускорителя вычислений. Предлагается метод автоматической трансляции последовательных программ для их выполнения на гибридной системе. Метод основан на построении полного графа алгоритма и эффективен для программ линейного класса.

1. Введение

Проблема автоматического распараллеливания программ хорошо известна. Но редко обращают внимание на то, что для различных целевых архитектур приходится решать ее поразному. Некоторых результатов удается достичь там, где целевая модель вычислений мало отличается от обычной модели исходной последовательной программы. Это, прежде всего, OpenMP, где параллелизм выражается в виде дополнений к основному последовательному коду в виде директив компилятору. Аналогичным свойством обладает модель DVM [1], для которой также есть некоторая перспектива автоматизации распараллеливания. Но уже для модели распределенных вычислений SPMD (Single Program применяется к Multiple Data), к которой относятся системы, основанные на MPI, а также PGAS-языки: UPC, CAF и т.д., разработчики распараллеливающих компиляторов сталкиваются со значительными трудностями. Похожая ситуация складывается и для модели вычислений, основанной на графических ускорителях GPU, на которых удается достичь высокой производительности. Так называемые «распараллеливающие» компиляторы для CUDA от Portland Group Inc. [2] – не что иное, как попытка прикрыть сложности и замысловатости гибридной модели программирования для GPU языком директив над обычным Фортраном или Си. Некоторый оптимизм вселяет недавняя работа [3], где для ускорителей на базе GPU автоматически распараллеливаются последовательные программы того же класса, что и в нашей работе.

Основная причина трудностей автоматизации параллельного программирования для названных моделей вычислений, на наш взгляд, коренится в самих этих моделях. Они изначально требуют *полного контроля над параллелизмом со стороны программиста.* Это, конечно, создает дополнительные возможности для повышения эффективности, но приходится платить высокую цену - увеличение трудоемкости. Причем приходится тратить силы не столько на алгоритмы или математику, сколько на оптимизацию «раскладки» уже существующих алгоритмов на имеющиеся аппаратные ресурсы: перегруппировку вычислений, агрегацию данных и на оптимальное распределение этих групп и агрегаций по пространству и времени.

Нами предлагается другой путь, когда основной контроль над параллелизмом отдается аппаратуре. А программист-математик занимается только логическими взаимосвязями внутри алгоритма, определяя, что чему логически предшествует или что без чего не может быть вычислено. Он разбивает программу вычислений на небольшие фрагменты (будем называть их узлами), выражая некоторым образом связи по данным между ними. Каждый узел принимает какие-то данные от других узлов и формирует новые данные для других. При этом инициатива взаимодействия между получателем и отправителем принадлежит отправителю. Иначе говоря, программа каждого узла должна уметь «вычислять адреса» своих получателей. Сам узел-получатель становится активным лишь тогда, когда к нему поступает вся нужная информация. Ниже модель вычислений, основанная на этих принципах, излагается более подробно.

Трудности для программиста оборачиваются и трудностями для компилятора, точнее для разработчика компилятора. Поэтому модель вычислений, облегчающая ручное параллельное программирование, открывает больше перспектив и для автоматизации распараллеливания.

Говоря о простоте программирования, мы имеем в виду абстрактную простоту решаемой задачи. Программирование на языке DFL, к сожалению, пока нельзя назвать простым: во-первых, оно слишком отличается от традиционного и потому требует переучивания, а во-вторых, недостаточно развиты инструменты, облегчающую написание и отладку программ. Но для задачи автоматизации распараллеливания эти проблемы как раз не важны, и это делает задачу автоматической трансляции последовательных программ в DFL особенно актуальной.

Различие в моделях целевых архитектур порождает и существенные различия в способах анализа и компиляции программ. Существующие компиляторы распараллеливают циклы, пытаясь выяснить про каждый цикл, может ли он выполняться как параллельный. Для этого проверяется, нет ли зависимостей между операциями чтения и записи, мешающих параллельному выполнению цикла. Цикл не может выполняться как параллельный, если на разных его итерациях производится доступ к одному и тому же элементу массива, и хотя бы один из этих доступов является записью. При этом важно обнаружить сам факт наличия или отсутствия зависимости, а какие конкретно операции пишут и читают, и в какие элементы – не важно. Поэтому, такой анализ имеет право быть неполным [4, гл.11]. Более развитые компиляторы пытаются преобразовать циклы так, чтобы хотя бы один (желательно, внешний) цикл стал параллельным.

Наш подход основан на переводе программы в модель вычислений, в которой неявный параллелизм извлекается динамически по принципу готовности данных. Для перевода в такую модель не требуется дополнительной информации, кроме той, которую хороший анализатор способен извлечь из исходной программы – точное описание графа потоковых (истинных) зависимостей между экземплярами операций с памятью. Построение этого описания – наиболее сложная часть процесса трансляции в DFL, который, в сущности, является одной из форм представления этого графа. Выполнение этого описания требует известных накладных расходов, на минимизацию которых и нацелена архитектура предлагаемого вычислителя.

Статья имеет следующую структуру. В разделе 2 приводится описание гибридной модели вычислений, в которой обычная последовательная программа взаимодействует с программой на языке потока данных. Потоковая часть во многом подобна классической модели dataflow [5], но имеет и важное отличие, состоящее в наличие контекста, который полностью контролируется (задается) программистом. Описываются основные черты языка программирования DFL, соответствующего этой модели вычислений. В разделе 3 описывается архитектура вычислителя, способного выполнять программу на DFL. В разделе 4 обсуждается трансляция последовательных программ в эту модель вычислений. При этом накладываются определенные ограничения на допустимые исходные программы, и вводится понятие графа алгоритма, описание которого может быть автоматически построено для любой допустимой программы. По описанию графа уже легко строится программа на DFL. В разделе 5 работа компилятора демонстрируется на примере программы решения двумерной задачи Пуассона методом Якоби.

2. Модель вычислений

Вычисление, рассматриваемое как целое, разбивается на *вычислительные элементы*, каждый из которых является экземпляром одного из *типовых узлов*. Между элементами возникают связи по данным: одни элементы вычисляют некоторые данные, другие эти данные используют. Интерфейс и поведение типового узла задаются в его описании.

Конечный набор описаний типовых узлов и является программой на языке DFL. Описание типового узла состоит из *заголовка* и *программы узла*. В заголовке указываются *входы* (имя и тип каждого) и *контекст*, задаваемый как список имен полей целого типа. В программах узлов для записи вычислений используется подмножество языка Паскаль. В качестве аргументов могут использоваться только значения входов, полей контекста, а также *констант*, а результатом работы может быть только посылка некоторого количества данных на входы других узлов (или хост-процессор).

Данные между узлами передаются в виде *токенов*. Оператор посылки токена имеет вид:

$$v \rightarrow N.p\{e_1,\ldots,e_k\}$$

где v – посылаемое значение, N – имя узла, $\underline{p}$ – имя входа, e_i – целые выражения, в совокупности задающие контекст целевого виртуального узла. Читается: послать значение v на вход p виртуального узла $N\{e_1,\ldots,e_k\}$. Или короче: послать v на $N.p\{e_1,\ldots,e_k\}$. Виртуальным узлом мы называем экземпляр типового узла с конкретным набором значений полей контекста. Несколько токенов, направленных на разные входы одного и того же виртуального узла, должны встретиться, и когда соберутся токены для всех входов, образуется пакет – задание на выполнение программы узла. Созданные пакеты друг от друга не зависят и могут выполняться в произвольном порядке или в параллель. Для каждого пакета выполняется программа соответствующего типового узла, в результате чего могут появиться новые токены.

Для иллюстрации модели вычислений приведем простой пример: вычисление массива частичных сумм произведений двух массивов. На Рис. 1 слева приводится спецификация алгоритма в виде фрагмента на Фортране, а справа – соответствующий фрагмент на DFL, состоящий из одного трехвходового узла P.

Здесь параллелизма нет: узел P{i+1} не может работать, пока узел P{i} не отправил ему свою частичную сумму s. Но можно делать умножения параллельно, если выделить его в отдельный узел

```
REAL A(N),B(N),C(N),S
S = 0.0
DO I = 1,N
  S = S + A(I) * B(I)
  C(I) = S
ENDDO
```

```
node P(s:real,a:real,b:real) {i};
 var v:real;
 begin
   v := s+a*b;
   v -> P.s{i+1};
   v -> C.s{i};
 end;
```

Слева – фортран, справа – DFL. Предполагается, что исходные данные засылаются на входы a и b узлов P{i}, i=1,…,N, а результаты будут на C.s{i}. Кроме того, на P.s{1} надо послать значение 0.0.

Рис. 1. Фрагмент программы вычисления частичных сумм произведений двух векторов

Параллельная система работает в паре с обычным процессором, будем называть его хост-процессором, или просто хостом. На нем в рамках обычной ОС запускается последовательная программа, в которой некоторые сложные счетные подзадачи решаются на параллельной подсистеме. Хост-программа посылает исходные данные для подзадачи в виде токенов на параллельную подсистему, дожидается приема результирующих токенов и продолжает последовательную работу. Во время ожидания хост-процессор может заниматься счетом других подзадач, если они есть. Параллельная программа в подсистему должна быть загружена заранее.

На уровне DFL прием токенов из хоста не отличается от приема любых других токенов: в токене должен быть задан код принимающего узла, контекст и значение. Токены-результаты, поступающие на хост-процессор, немного отличаются от стандартных: их целевой узел имеет имя с суффиксом _out, единственный вход и пустую программу.

В хост-программе посылка входных токенов выражается специальным оператором типа

SEND X,Y

у которого аргумент – список имен массивов (или переменных). Для каждого элемента заданного массива X формируется токен со значением в качестве данного, именем узла X_in и индексами элемента в качестве контекста. Результаты принимаются оператором типа

RECEIVE A(N),B(N+1),N

в котором тоже задан список массивов или простых переменных, но при имени массива обязательно должно быть указано в скобках выражение – количество принимаемых токенов. В данном случае это количество N также принимается отдельным токеном. Для каждого элемента списка X(m) в параллельной подсистеме должны быть созданы m токенов, направленных на узел X_out с некоторым контекстом {i,j,…}. Каждый такой токен передается на хост, где его значение записывается в элемент X(i,j,…) и подсчитывается их количество. Число размерно-

стей должно совпадать с числом полей контекста. Массив X считается принятым после поступления ровно *m* таких токенов. Токены-результаты для разных массивов и простых переменных могут приходить в любом порядке. Выполнение оператора RECEIVE завершается, когда все указанные в нем массивы и переменные будут приняты. Описанные правила взаимодействия хост-процессора с параллельной потоковой подсистемой образуют основу гибридной модели вычислений.

3. Архитектура сопроцессора

Архитектура вычислителя для данной модели вычислений была предложена в [6] и получила свое дальнейшее развитие в [7]. Здесь мы применяем его как сопроцессор для ускорения счетных подзадач. Вычислитель состоит из некоторого числа ядер K, соединенных коммутатором (Рис. 2). К этому же коммутатору подключен блок ввода-вывода. Ядро состоит из модуля ассоциативной памяти (МАП) и исполнительного устройства (ИУ). Токены через коммуникационную сеть приходят в МАП и могут там находиться в ожидании других токенов. МАП состоит из ассоциативной памяти (АП) ключей, где хранятся ключи со ссылками на наборы токенов, и памяти для наборов токенов (ПТ). Ключ токена состоит из имени (кода) узла и контекста. В аппаратной реализации он упаковывается в одно 64-битное слово. Входящий токен сравнивается в АП по ключу со всеми присутствующими ключами, и если обнаруживается совпадение, то его данное вносится в соответствующий хранимый набор на место указанного в нем входа. Если набор оказывается полным, выполняется процедура образования пакета. Если совпадения ключа входящего токена с хранимым ключом не обнаружено, в АП заносится новый ключ и в ПТ отводится место для нового набора, куда помещается сам токен. Если узел одновходовой, то сразу выполняется процедура образования пакета.

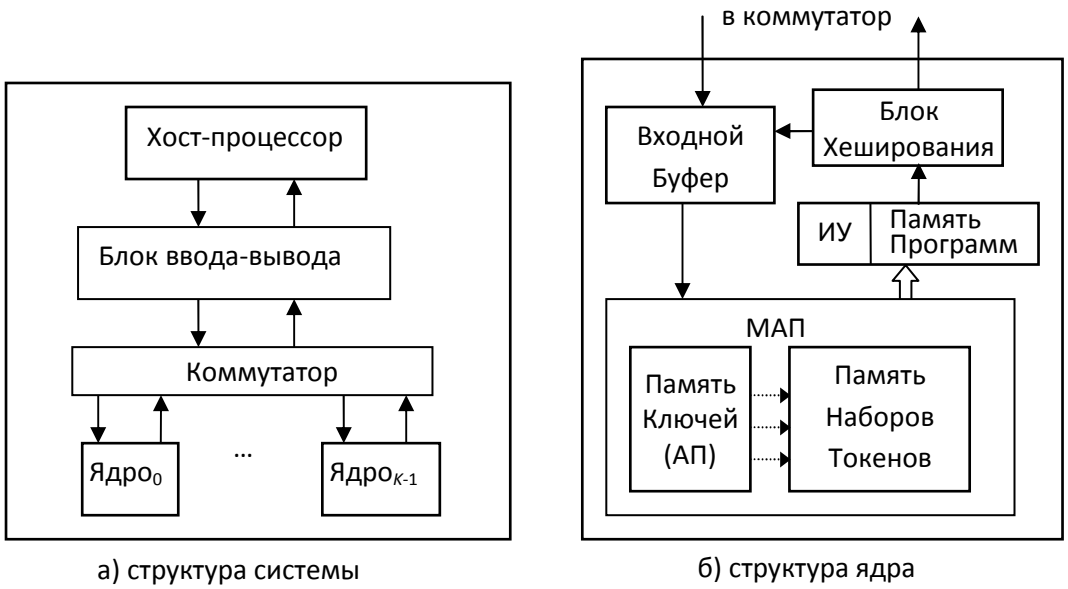

Рис. 2. Архитектура потокового вычислителя

Токены с общим ключом должны оказаться в одном и том же МАП. Для этого номер целевого МАП формируется при создании токена посредством вычисления определенной функции распределения, зависящей только от ключа. Это принципиальный момент – этим обеспечивается возможность эффективной и распределенной реализации ассоциативного поиска. Стандартная функция распределения обеспечивает достаточно хорошее перемешивание, но разрушает потенциальную локальность. Пользователь может указать или задать свою функцию распределения. Если при хорошем балансе нагрузки удается обеспечить локальность (чтобы токены чаще посылались в свое ядро), то сокращается нагрузка на коммутатор и, как следствие, повышается общая производительность системы. Перед выходом на коммутатор номер вычисленного

МАП сравнивается с номером данного ядра и в случае совпадения токен направляется во входной буфер, минуя коммутатор.

Сформированный пакет, состоящий из токенов набора и их общего ключа, поступает через буфер пакетов в ИУ. Обычно на каждый вход приходит по одному токену. После формирования пакета набор токенов и ключ удаляются из МАП. Пока набор токенов не полон, он продолжает храниться в МАП в памяти наборов токенов, а общий ключ токенов – в памяти ключей.

Для каждого поступающего пакета в ИУ выполняется программа соответствующего узла. При этом могут образоваться новые токены, которые через буфер и блок хеширования (БХ) поступают на вход коммуникационной сети.

В памяти программ каждого ИУ хранятся копии программ и констант. Они загружаются туда из хост-процессора перед началом работы (на схеме соответствующий канал не показан).

4. Трансляция последовательных программ в DFL

В языке DFL программа узла, вычисляющая значение, должна уметь разослать его всем узлам, которые в нем нуждаются. Поэтому для перевода заданной последовательной программы P в язык DFL необходимо для каждой операции записи (то есть присваивания элементу массива или простой переменной) определить все операции чтения (в правых частях операторов присваивания), в которых читается записываемое здесь значение. Фактически нужно построить описание графа алгоритма [8, гл.6], которое в параметрическом виде описывает любой граф вычисления программы, определяемый по прогону программы при конкретных входных данных. Граф вычисления состоит из вершин двух типов. Вершины первого типа (записи) соответствуют исполнениям операций записи в память, а второго типа (чтения) операциям чтения из памяти. Из вершины-записи идет дуга в вершину-чтение, если эта операция чтения считывает из памяти именно то значение, которое было записано данной операцией записи.

Отдельную операцию записи или чтения можно идентифицировать парой $(M,\boldsymbol{I})$, где M – место в программе, где эта операция находится, а $\boldsymbol{I}$ – набор значений параметров внешних (для этого места) циклов. Используя эти пары как идентификаторы вершин, определим отображение

$$\boldsymbol{F}: (R, \boldsymbol{I}_R) \rightarrow (W, \boldsymbol{J}_W) \qquad (1)$$

которое по идентификатору чтения $(R, \boldsymbol{I}_R)$ выдает идентификатор записи $(W, \boldsymbol{J}_W)$, записавшей читаемое значение (или знак, обозначающий, что записей не было, а читается исходное значение).

Но для перевода в DFL нам требуется обратное: для каждой операции записи найти все (их может быть несколько или ни одной) операции чтения, которые именно это значение читают, то есть нужно многозначное отображение

$$\boldsymbol{G}: (W, \boldsymbol{J}_W) \text{ -> } \{ (R, \boldsymbol{I}_R) \} \qquad (2)$$

которая по идентификатору записи $(W,\boldsymbol{J}_W)$ выдает множество идентификаторов чтений $\{(R,\boldsymbol{I}_R)\}$, читающих записываемое значение.

К сожалению, выразить в явной форме эти отображения удается далеко не всегда. Но есть хорошо определенный класс программ, для которых это возможно: это *линейные* программы [8, гл.6]. Они могут содержать следующие конструкции:

- операторы присваивания с линейными индексными выражениями во всех обращениях к массивам;
- правильно вложенные циклы do-enddo с единичным шагом, линейными выражениями для верхней и нижней границы и без преждевременных выходов из цикла;
- правильно вложенные условные операторы if-then-else-endif с условиями вида e=0 или e>0, где e – линейное выражение.

Выражения называются *линейными*, если они являются линейными формами с целыми коэффициентами, где в качестве переменных используются лишь только параметры внешних циклов и некоторые фиксированные параметры, которые определяются вне участка, подлежащего анализу и преобразованию, и в нем не изменяются.

Пока мы рассматриваем только строго линейные программы. Более того, счетный участок целиком должен быть представлен подпрограммой, принимающей входные данные и возвращающей результаты через параметры. Впоследствии некоторые из ограничений могут быть

сняты. Предполагаемая общая схема обработки исходной программы такая: транслятор сканирует программу, ища участки, удовлетворяющие условиям допустимости, и переводит их в DFL. А на их место вставляется новый код, который передает данные на входные узлы и принимает результаты с выходных. По желанию программист может отметить сам нужные счетные участки директивами или ограничить область поиска таких участков. Сама внешняя программа выполняется на хост-процессоре, а ее преобразованная параллельная часть – на параллельном сопроцессоре. Таким образом, в худшем случае, наш транслятор ничего не сделает, и программа останется последовательной. Но на практике вычислительные ядра очень многих реальных программ удовлетворяют условиям допустимости.

В работе [9] введено понятие дерева выбора, с помощью которого возможно описать граф потоковых зависимостей любой линейной программы. Там же представлен алгоритм получения такого графа. Имея граф, нетрудно произвести трансляцию в DFL. Каждый оператор присваивания переходит в отдельный узел. Элементы массивов в правой части превращаются во входы. Контекстом узла становится список переменных всех внешних циклов данного оператора. В начало узла помещается сам оператор присваивания, где элементы массивов правой части заменены именами соответствующих входов, а левая часть – локальной переменной. Дерево выбора, описывающее множество получателей данной операции записи, превращается в составной оператор, который в зависимости от заданных условий выполняет рассылку вычисленного значения на входы других узлов.

5. Пример трансляции с Фортрана в DFL

На Рис. 3 вверху слева показан пример исходной линейной подпрограммы на Фортране, которая решает двумерную задачу Пуассона простым итерационным методом Якоби. Начальное приближение поступает на вход задачи как двумерный массив A(0:L,0:L). В нем же выполняются итерации с использованием вспомогательного массива B(0:L,0:L) и возвращается результат. Крайние элементы не пересчитываются и используются как постоянные граничные условия. Остальные фрагменты (серые) – результат трансляции. Слева в центре – вариант подпрограммы, которым подменяется исходная. Здесь язык Фортран расширен операторами LOADCONST, которыми заносятся постоянные значения L и M в память констант всех ядер, а также SEND и RECEIVE, которые, соответственно, посылают исходные данные в параллельную подсистему и принимают ее результаты.

На Рис. 3 внизу и справа показан перевод тела этой подпрограммы в DFL. В подпрограмме два оператора присваивания, назовем их B1 и A1, каждый вложен в тройной цикл. Соответственно, узлы B1 и A1 имеет контекст из трех полей. Тело каждого узла содержит образ самого оператора присваивания, за которым следуют операторы рассылки вычисленного значения. Узел A_in принимает входные элементы и рассылает их куда требуется, а через узел A_out происходит возврат результатов.

```
SUBROUTINE JAC(A,L,M)
REAL A(0:L,0:L),B(0:L,0:L),X,Y
IF M.GE.1 THEN
IF L.GE.2 THEN
DO IT = 1,M
 DO J = 1,L-1
  DO I = 1,L-1
   B(I,J)=(A(I-1,J)+A(I,J-1)+
          A(I+1,J)+A(I,J+1))/4
  ENDDO
 ENDDO
 DO  J = 1, L-1
  DO I = 1, L-1
   A(I, J) =  B(I,J)
  ENDDO
 ENDDO
ENDDO
ENDIF
ENDIF
END
```

```
SUBROUTINE JAC(A,L,M)
REAL(8) A(N,N)
LOADCONST L,M
SEND A
RECEIVE A((N+1)*(N+1))
END
```

```
vconst int L,M;

node B_1(A_1:real,A_2:real,A_3:real,
                 A_4:real) {IT,J,I};
   (((A_1+A_2)+A_3)+A_4)/4 → A_1.B_1{IT,J,I};

node A_1(B_1:real) F3{IT,J,I};
 if IT=M then
   B_1 → A_out{I,J}
 else begin
   if J<>(L-1) then
    B_1 → B_1.A_2{IT+1,J+1,I};
   if I>=2 then
    B_1 → B_1.A_3{IT+1,J,I-1};
   if J>=2 then
    B_1 → B_1.A_4{IT+1,J-1,I};
   if I<>(L-1) then
    B_1 → B_1.A_1{IT+1,J,I+1};
 end;
```

```
node A_in(A:real32) {I_0,I_1};
 var IT:int;
 begin
  if I_1=L then begin
   A → A_out{I_0,L};
   if I_0>=1 then
    if I_0<>L then
     for IT := 1 to M do
      A → B_1.A_4{IT,L-1,I_0};
  end
  else
   if L=1 then A → A_out{I_0,I_1}
   else
    if M>=1 then begin
     if I_1=0 then A → A_out{I_0,0};
     if I_0=L then
      if I_1>=1 then begin
       for IT := 1 to M do
        A → B_1.A_3{IT,I_1,L-1};
       A → A_out{L,I_1};
      end else begin
       if I_0<>0 then
        if I_1<>L-1 then
         A → B_1.A_2{1,I_1+1,I_0};
       if I_1=0 then
        for IT := 2 to M do
         if I_0>=1 then
          A → B_1.A_2{IT,1,I_0}
        else begin
         if I_0=0 then begin
          for IT := 2 to M do
           A → B_1.A_1{IT,I_1,1};
          A → A_out{0,I_1};
         end else begin
          if I_1>=2 then
           A → B_1.A_4{1,I_1-1,I_0};
          if I_0>=2 then
           A → B_1.A_3{1,I_1,I_0-1};
         end;
         if I_0<>L-1 then
          A → B_1.A_1{1,I_1,I_0+1};
        end;
       end
      end
     else
      A → A_out{I_0,I_1};
 end;

node A_out(A:real) {I_0,I_1};
```

Рис. 3. Пример трансляции подпрограммы метода Якоби

Следует отметить, что в основной части полученной программы нет циклов. Есть несколько циклов во входном узле A_in, каждый из которых рассылает одно значение сразу на много узлов. Но исходные циклы исчезли. Вместо них в хост-процессоре создается поток элементов входных массивов, которым активируется сразу множество узлов. Одни узлы через данные передают активность другим узлам; этим порождается тот же объем вычислений, который в исходной программе порождался циклами.

При выполнении на системе с достаточно большим числом ядер в нашем примере достигается ускорение порядка L^2, если не учитывать время на пересылки. При этом совмещение вычислений с обменами между ядрами происходит автоматически, благодаря отсутствию барьерных синхронизаций.

6. Выводы

Наш компилятор выявляет потоковые зависимости алгоритма и переводит программу в такую форму, в которой зависимости выражены явно. При этом никаких явных указаний на параллелизм в результирующем коде нет. Программа всякого узла активируется по готовности входных данных независимо от других узлов. Таким образом, автоматическое распараллеливание достигается совместными усилиями компилятора и исполняющей системы. Компилятор лишь выявляет полный граф потоковых зависимостей и выражает его на языке DFL. Это отличает наш подход от традиционных распараллеливающих компиляторов, которые стремятся выявить параллелизм на стадии компиляции, используя модели вычислений (Open MP, MPI, CUDA и др.), требующие явно выраженного параллелизма. И это позволяет нам, в конечном счете, извлекать из программ больше параллелизма, чем это делают традиционные распараллеливающие компиляторы.

Список литературы

1. Бахтин В.А., Клинов М.С., Крюков В.А., Поддерюгина Н.В.. Автоматическое распараллеливание последовательных программ для многоядерных кластеров. //Труды Международной научной конференции "Научный сервис в сети Интернет: суперкомпьютерные центры и задачи", сентябрь 2010 г., г. Новороссийск. - М.: Изд-во МГУ, 2010, с.12-15.
2. The Portland Group. PGI Accelerator Programming Model for Fortran & C. November 2010. URL: http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf (дата обращения: 15.02.2011).
3. Baskaran, M.M., Ramanujam, J., Sadayappan, P. Automatic C-to-CUDA Code Generation for Affine Programs. // In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, Springer, Heidelberg, 2010, pp. 244-263.
4. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: пер. с англ. М: «И.Д. Вильямс», 2008.
5. Arvind, S. The evolution of dataflow architectures: from static dataflow to P-RISC. // International Journal of High Speed Computing. 1993. V. 5, no. 2, pp. 125-153.
6. Бурцев В.С. Выбор новой системы организации выполнения высокопараллельных вычислительных процессов, примеры возможных архитектурных решений построения суперЭВМ // В сб. Бурцев В.С. Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ, ИВВС РАН, М.:1997, с.41-78.
7. Стемпковский А.Л., Левченко Н.Н., Окунев А.С, Цветков В.В. Параллельная потоковая вычислительная система — дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // "Информационные технологии" №10, 2008, с.2–7.
8. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб: БХВ-Петербург, 2004.
9. Климов Арк.В. Использование деревьев выбора для описания состояний в распараллеливающем компиляторе. // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность. Труды Всероссийской суперкомпьютерной конференции, М.: Изд-во МГУ, 2009, с.238-240.