АКАДЕМИЯ НАУК СССР
СИБИРСКОЕ ОТДЕЛЕНИЕ
А В Т О М Е Т Р И Я

№ 6 1989

ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

УДК 681.142.2

М. А. БЕРЕЗОВСКИЙ, А. Л. МИНКИН
(Москва)

ОПТИМИЗАЦИЯ ВЕКТОРНЫХ АРИФМЕТИЧЕСКИХ ОПЕРАТОРОВ ДЛЯ ПРОЦЕССОРА А-12

При программировании на языках высокого уровня для параллельных ЭВМ одной из важных задач является оптимизация векторных арифметических операторов *, сокращающая время их выполнения и минимизирующая объем требуемой дополнительной памяти для хранения промежуточных результатов вычислений. Такие операторы могут быть как параллельными конструкциями в векторных языках (EVAL [1], расширение Фортрана [2]), так и результатом распараллеливания циклических участков в программах, написанных на обычных последовательных языках. Рассматриваемая постановка задачи связана с созданием препроцессора к транслятору с Фортрана, автоматически векторизующего циклические участки в программах для вычисления на векторном процессоре А-12 [3]. Вопросы структуры и реализации препроцессора рассмотрены в [4]. Предлагаемые методы оптимизации учитывают специфику данного процессора, однако могут быть применены и для других параллельных ЭВМ, имеющих несколько одновременно работающих конвейерных арифметических устройств разного назначения.

В первом разделе статьи дается анализ основных особенностей архитектуры процессора А-12, позволяющий сформулировать задачи оптимизации. Во втором разделе, используя представление арифметического оператора в виде ассоциативного дерева, рассматриваются основные идеи выделения параллельно выполняемых векторных арифметических операций. Следующие два раздела посвящены собственно рассмотрению основных этапов работы алгоритма. В первом из них описывается методика разметки вершин дерева и выделения из векторного арифметического оператора скалярных подвыражений. В последнем разделе рассматривается процесс построения искомой последовательности вычислений, основанный на проведенной ранее разметке. Алгоритм второго этапа оптимизации представлен в виде логической процедуры.

1. **Определение основных задач оптимизации.** Процессор А-12 имеет два арифметических устройства конвейерного типа — сложитель и умножитель, работающие одновременно и независимо. К устройствам памяти относятся: основная память данных, блок быстрых регистров и табличная память. При работе с большими векторами главным хранилищем является основная память данных. За один такт работы А-12 может быть выполнено лишь одно обращение к этому устройству. Для выполнения бинарной операции требуются два обращения к памяти для считывания операндов и одно — для записи результата. В силу конвейерности арифметические устройства могут выдавать результат элементарных вычислений над компонентами векторов на каждом такте. Однако

* Термин «векторный арифметический оператор» подразумевает его поэлементное независимое выполнение для всех компонент векторов-операндов.

ний хранятся там же, где и исходные операнды. Для удобства дальнейшего изложения будем называть память для хранения промежуточных результатов вычислений над векторами регистрами.

В реализации препроцессора [4] процесс вычисления векторного арифметического выражения состоит из последовательности обращений к подпрограммам математической библиотеки процессора, реализующим элементарные вычисления над векторами. Набор выбранных подпрограмм и последовательность их вызова определяют эффективность вычисления арифметического выражения. Пусть требуется вычислить выражение $A = C + D * F$ (далее будем предполагать, что операции проводятся над векторами). Это выражение можно вычислять путем последовательного выполнения двух операций — умножения и сложения. При этом на каждом этапе вычислений работает по одному арифметическому устройству, требуются шесть обращений к памяти данных на одну компоненту вектора и один регистр. Рассмотрим другую последовательность вычислений. Каждый результат умножения можно не записывать в регистр, а сразу же загружать в сложитель и выполнять операцию сложения. Это потребует всего четыре обращения к памяти. Вычисления ведутся одновременно на двух арифметических устройствах, и дополнительный регистр не нужен.

Выражения, включающие две арифметические операции, назовем триадными (по числу операндов). Повышение эффективности в А-12 может быть достигнуто для следующих триад: *+, +*, *−, −*.

В связи с тем, что на время вычислений влияет количество обращений к памяти, целесообразно также рассматривать триадные операции вида **, ++, +−, −+, −−, хотя их одновременное выполнение невозможно. Операции, допустимые в одной триаде, назовем совместимыми. Первой задачей оптимизации, таким образом, является построение последовательности вычислений арифметического выражения, содержащей максимальное число триадных операций.

Рассмотрим арифметическое выражение $A = \sin(F) * ((A + B)/(C - D))$. Если его выполнять в следующей последовательности: sin, +, −, /, *, то потребуются три регистра. При другой же последовательности: +, −, /, sin, * — только два, т. е. порядок вычислений определяет количество требуемых регистров. Отсюда вторая оптимизационная задача: для арифметического выражения построить последовательность вычислений, требующую минимального числа регистров для хранения промежуточных результатов вычислений.

Две сформулированные задачи оптимизации могут быть конкурирующими. Так, в выражении $A = B/C * \log(D) + \sin(F)$ для выделения триадной операции *+ потребуются три регистра; если же выполнять эти операции как последовательные бинарные, то лишь два. В случае такой конкуренции оптимизация будет осуществляться по первому направлению.

Векторные арифметические операторы могут содержать в себе скалярные подвыражения. Они вычисляются не с помощью подпрограмм, а обычными арифметическими командами. Рассмотрим выражение $A = F * B + C + D$, где F, B и D — скаляры. Если этот оператор выполнять последовательно слева направо, то потребуются две векторные операции сложения. Если же вначале выполнить действия $F * B + D$, а потом полученный результат сложить с C, то нужно одно векторное сложение и оператор фактически будет выполнен в 2 раза быстрее. Таким образом, еще одной оптимизационной задачей является выделение в выражении скалярных подвыражений максимально возможной длины.

Перечисленные оптимизационные задачи можно объединить следую-

Рис. 1. Ассоциативное дерево арифметического выражения

щей формулировкой: построить для арифметического выражения последовательность вычислений, содержащую минимальное число операций с векторами (обращений к подпрограммам математической библиотеки) и требующую минимальное число регистров. В случае конкуренции при оптимизации по этим двум направлениям предпочтение отдается первому из них.

2. **Формирование параллельных вычислений на дереве арифметического выражения.** В работе рассматриваемого алгоритма используются свойства ассоциативности и коммутативности операций + и *, но порядок вычислений, определяемый скобками, остается неизменным. Поэтому для рассмотрения алгоритмов оптимизации удобно воспользоваться представлением арифметического оператора в виде ассоциативного дерева [5]. Каждый лист дерева соответствует операнду, а внутренняя вершина — операции. Вершина, соответствующая унарной операции, имеет одного потомка, бинарной неассоциативной — двух, а ассоциативной — не менее двух. На рис. 1 представлено дерево для выражения

$$(A-B+C*D+H+(\sin(E)+F*G))* \\ *(M-N)*Z*(U+P*R-S+T+Y*Z)+W/V). \qquad (1)$$

Далее сформулируем ряд правил оптимизации на дереве, которые приводят к выделению триад. Будем называть триадообразующей вершину, соответствующую операции, которая потенциально может входить в триаду.

Алгоритм обработки арифметического оператора можно разбить на два этапа: 1) разметка дерева и выделение скалярных поддеревьев; 2) формирование собственно последовательности вычислений. Методика разметки дерева тесно связана с последующей стратегией построения последовательности вычислений. Поэтому рассмотрим вначале некоторые аспекты этого построения.

Ясно, что появлению в последовательности некоторой операции должно предшествовать появление тех операций, результаты вычисления которых используются данной. Формирование всякой новой операции сопровождается преобразованием дерева выражения: использованные вершины удаляются из дерева, а вместо них остается новый лист с указанием регистра R_i, в котором запоминается результат сформированной операции (рис. 2).

Бинарная и унарная операции всегда порождаются одной внутрен-

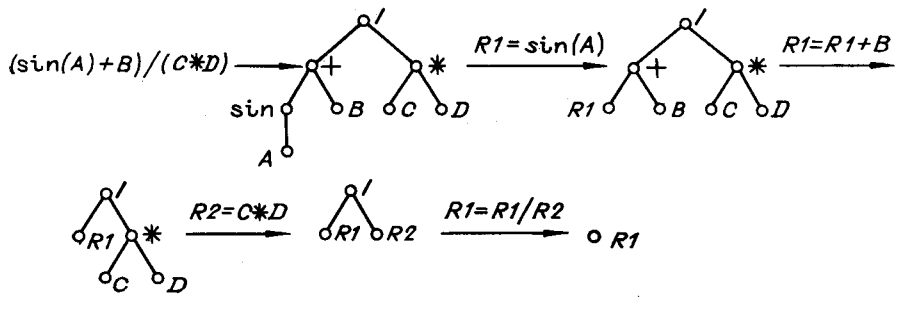

Рис. 2. Пример формирования последовательности операций на дереве

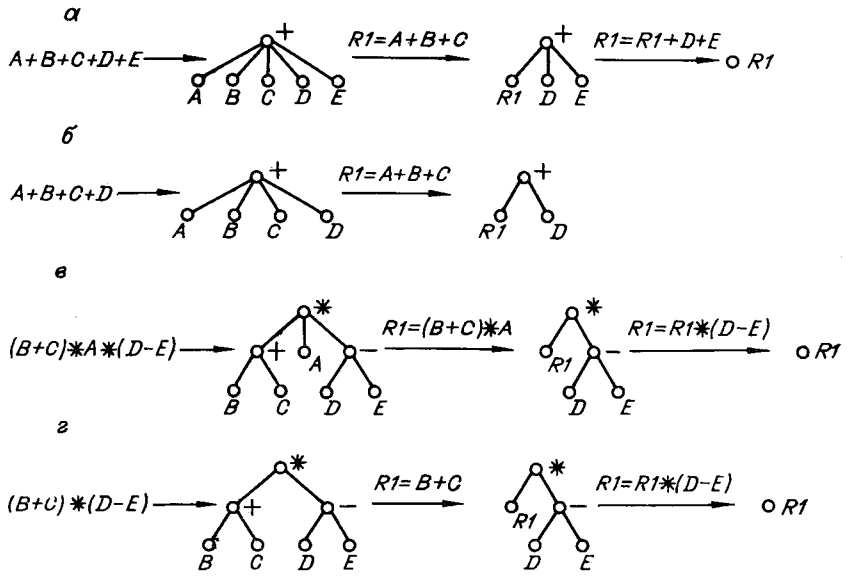

Рис. 3. Примеры формирования триад

ней вершиной. Триадообразующая вершина может соответствовать как ассоциативной, так и неассоциативной операции, поэтому триадные операции могут формироваться одной внутренней вершиной (ассоциативной) или двумя смежными (рис. 3). Прежде чем переходить к следующей вершине дерева, находясь в триадной вершине, следует выделить наибольшее число триад. Для формирования триады необходимо иметь готовыми к использованию три операнда (исходные или хранящиеся в регистрах). Естественно, что в первую очередь следует подбирать в триаду операнды из регистров, что сократит их общее требуемое число.

Если суммарное число листьев триадообразующей вершины ассоциативной операции нечетное, то они все используются при формировании триад (рис. 3,*а*), в противном случае останутся два листа (рис. 3, *б*). Всякую триадообразующую вершину, имеющую двух потомков-листьев (изначально или в результате ранее выделенных операций), назовем четной вершиной, все остальные — нечетными. Триадообразующая вершина неассоциативной операции всегда четная. Если у триадообразующей вершины ассоциативной операции нечетное число нечетных потомков, то она сама является нечетной, в остальных случаях — четной.

Рассмотрим случай формирования триад из двух смежных внутренних вершин. При этом одна из вершин является отцом по отношению к другой. Возникновение такой триады возможно при выполнении следующих условий: во-первых, операции, соответствующие двум смежным триадообразующим вершинам, должны быть совместимыми и, во-вторых, триадообразующая вершина-потомок должна быть четной. Вопрос о совместимости операции принципиально не влияет на работу алгоритма, а определяется исключительно целесообразностью совмещения тех или иных вычислений для конкретной ЭВМ. Таким образом, алгоритм является перенастраиваемым на разные триады. Если четная вершина-потомок соответствует операции, не совместимой с операцией вершины-отца, то из нее формируется бинарная операция и получается нечетная вершина. В данной реализации все операции, допустимые в триадах, совместимы друг с другом, и проверка их совместимости явно не отражена в изложенном далее алгоритме.

При формировании описываемой триады в качестве третьего операнда используется одна нечетная вершина-потомок триадообразующей вершины-отца (рис. 3, *в*). Если у вершины-отца все потомки являются четными вершинами, то, прежде чем формировать первую триаду, одна

из четных вершин трансформируется в нечетную созданием из нее бинарной операции (рис. 3, *г*).

В общем случае триадообразующая вершина имеет несколько как четных, так и нечетных потомков. При этом вначале формируются все триады из четной и нечетной вершин, а затем только из нечетных. Число триад, порождаемых одной вершиной, равно $[(N_{od}-1)/2]+N_{ev}$, где N_{od} и N_{ev} — число нечетных и четных потомков соответственно. Сама вершина будет четной, если N_{od} — четное, и нечетной — в противном случае. Таким образом, зная N_{od} и N_{ev} для триадообразующей вершины, можно определить ее тип и число порождаемых ею триад.

3. **Разметка ассоциативного дерева.** Работа рассматриваемого алгоритма осуществляется за два прохода по дереву. При первом проходе ведется специальная разметка его вершин (определение их типа и присвоение им весов), которая затем используется при втором проходе. Кроме того, на первом этапе выделяются поддеревья, соответствующие скалярным подвыражениям.

Согласно изложенным правилам, для формирования триад в вершине необходимо знать тип ее потомков. Зная типы ее потомков, можно определить тип любой вершины. Поэтому если обойти дерево в обратном порядке [6] (переход к некоторой вершине осуществляется лишь после обхода ее потомков), то можно определить тип всех вершин дерева.

Рассмотрим теперь методику построения последовательности вычислений оператора, требующей минимального числа регистров. В работе [5] предложен алгоритм решения такой задачи для скалярного арифметического оператора с бинарными операциями. Алгоритм состоит из процесса присвоения вершинам дерева оператора определенных значений веса W (числовая метка) и последующего построения на их основе оптимального порядка вычислений (кода на Ассемблере). Присвоение веса вершинам ведется при обходе дерева в обратном порядке. Для всех листьев $W=0$. Пусть W_l, W_r — веса левого и правого потомков внутренней размечаемой вершины, тогда если $W_l=W_r$, то $W=W_r+1$, иначе $W=\max(W_l, W_r)$. Вес всякой внутренней вершины при этом равен числу регистров, необходимых для вычисления по поддереву, корнем которого она является, а вес корня дерева — числу регистров, требуемых для всего выражения. Искомая последовательность вычислений формируется при движении по дереву в прямом порядке так, что всякая посещаемая внутренняя вершина порождает соответствующую команду (операцию, операнды и место записи результата). При этом для любой внутренней вершины, если веса ее потомков разные, прежде обходят потомка с большим весом. Если же веса потомков одинаковые, то порядок их прохождения не важен.

Распространение описанного алгоритма на случай унарных операций не составляет труда. Для этого достаточно добавить разметку вершины указанной операции: если вес потомка — 0, то вес вершины — 1, в противном случае ей присваивается вес потомка. Расширим теперь данную методику на случай триадных операций. Будем рассматривать триадообразующую вершину, соответствующую ассоциативной операции. (Неассоциативная операция — частный случай.) Результат будет достигнут, если для триадообразующих вершин удастся сформировать триады так, чтобы их вес W (число требуемых регистров) минимально превышал наибольший из весов потомков. При построении триады важен тип входящих в нее вершин, а вершины одного типа равноправны. Следовательно, всякий раз выбирая в триаду потомков определенного типа, нужно также учитывать их вес.

Для формирования триады необходимо иметь готовыми к использованию три операнда. Однако в отличие от первой триады, сформированной в пределах некоторой вершины, во всех последующих требуются не три, а два новых операнда, так как в качестве третьего всегда берется результат предыдущей сформированной триады, хранящийся в регистре.

Рассмотрим, как влияют на W веса потомков, составляющих первую

триаду (W_{max} — наибольший из веса потомков, входящих в конкретную триаду). Очевидно, если $W_{max}=0$, то $W=1$ и порядок подбора по весам потомков не важен. Для случая $W_{max}>0$ имеем следующее:

1. Если триада формируется из нечетных потомков одного веса, то

4. В остальных случаях $W \geqslant W_{max}$. Исключением является случай нечетного потомка с весом 1 и четного — с весом 2, у которого, в свою очередь, есть оба потомка с весом по 1.

Определим теперь, как влияет вес потомков на W при формировании всех последующих триад рассматриваемой вершины:

1. Если в триаду входят два нечетных потомка одного веса, то $W \geqslant W_{max}+2$.

2. Если в триаду входят два нечетных потомка с разным весом, то $W \geqslant W_{max}+1$.

3. Если в триаду входит четный потомок, то $W \geqslant W_{max}+1$.

Исключение составляет случай, когда четный потомок имеет двух потомков с весом 1 каждый.

Основываясь на приведенных правилах, по количеству потомков того или иного веса и типа можно определить вес размечаемой вершины, а также признак S существенности подбора в триады потомков по весу. Если исходя из анализа веса всех потомков W превышает наибольший из них на минимально возможную в рассматриваемом случае величину, то подбор потомков по весу существен и признаку S присваивается вес, с которым в триаде должно быть не более определенного числа вершин. В противном случае $S=0$.

На рис. 4 представлена логическая процедура определения веса триадообразующей вершины и признака подбора по весу потомков в триады, построенная на основе приведенных правил и возможных исключений из них.

```
{node — номер размечаемой вершины дерева}
{W[node] — вес, присваиваемый размечаемой вершине}
{S[node]  — признак выбора сыновей node в триаду. S[node]=0, если выбор произволен,}
{S[node] > 0— вес, который в триаде должен иметь лишь один сын}
{W_max — максимальный из веса всех потомков вершины}
{W_max_od — максимальный из веса всех нечетных потомков вершины}
{N_od — число нечетных потомков вершины}
{N_ev — число четных потомков вершины}
{N(W) — число потомков вершины, имеющих вес W}
{N_od(W) — число нечетных потомков вершины, имеющих вес W}
procedure LABELING;
begin
  if (N_od = 0) then
    if (N(W_max) = 1) then
      begin W[node] : = W_max; S[node] : = 0; end
    else
      begin W[node] : = W_max + 1; S[node] := 0; end;
      else if (W_max = 0) then
    if ((N_ev = 0) AND (N_od = 2)) then
      begin W[node] := 0; S[node] := 0; end
      else
      begin W[node] := 1; S[node] := 0; end
```

```
if (N_od (W_max — 1) — MOD_2(N_od — 1) < Σ_{i<W_max-1} N_od(i)) then
    begin W[node] := W_max; S[node] := W_max — 1; end
  else
    begin W[node] := W_max + 1; S[node] := 0; end
else if (W_max = W_max-od + 1) then
  if (W_max = 2) AND (у четного сына с весом W_max каждый из двух оставшихся
    потомков имеет вес не менее 1) AND (N_od(1) > N_od(0)) then
    begin W[node] := 3; S[node]:= 0; end
  else if (N_od(W_max)— 1 — MOD_2(N_od — 1) < Σ_{i<W_max} N_od(i)) then
    begin W[node] := W_max; S[node]:= W_max-od; end
  else
    begin W[node] := W_max + 1; S[node] := 0; end
else if (W_max > W_max-od + 1) then
  begin W[node] := W_max; S[node] := 0; end
else if (N(W_max) > 1) then
  if (W_max > W_max-od) then
  begin W[node] := W_max + 1; S[node] := 0; end
  else if (N_od(W_max) — 1 — MOD_2(N_od — 1) < Σ_{1<W_max} N_od(i)) then
  begin W[node] := W_max + 1; S[node] := W_max; end
  else
  begin W[node] := W_max + 2; S[node] := 0; end
end;
```

Рис. 4. Логическая процедура разметки триадообразующей вершины

На рис. 5 приведен пример разметки дерева для выражения (1).

Как уже сообщалось, в векторном выражении целесообразно выделять как можно полнее скалярные подвыражения. Всякая арифметическая операция будет векторной лишь в том случае, если хотя бы один из ее операндов является вектором. Следовательно, достаточно проанализировать потомков вершины, чтобы определить является ли порождаемая вершиной операция векторной или скалярной. Если обнаружено, что некоторое поддерево соответствует скалярному подвыражению, то оно удаляется из дерева, а вместо него в дереве остается лист, указывающий имя дополнительной скалярной переменной, в которой будет запоминаться результат вычисления выделенного скалярного подвыражения. Выделение скалярных подвыражений приведено на рис. 6 для выражения (1) при условии, что переменные A, B, E, F, G, M, N, P, R, T, Y, Z являются скалярами.

Итак, на первом этапе работы алгоритма при проходе дерева арифметического оператора в обратном порядке решаются следующие задачи: установление типа вершин, присвоение вершинам веса, определение для триадообразующих вершин критерия подбора потомков в триады, выделение скалярных подвыражений. Все эти задачи решаются за один проход по дереву.

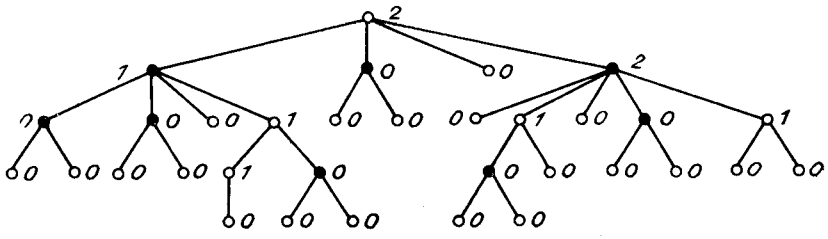

Рис. 5. Размеченное дерево:
цифры около вершин определяют их вес; ● — четная, ⊙ — нечетная вершина

4. **Построение оптимальной последовательности вычислений.** Процесс выделения искомой последовательности вычислений осуществляется при прямом проходе по дереву, начиная с его корня. На рис. 7, 8 представлены логические процедуры и функции, осуществляющие необходимые построения. Результатом их работы является последовательность псевдокоманд, указывающих виды операций, операнды и место записи результата. Эта информация достаточна для формирования обращений к подпрограммам математической библиотеки, вычисляющим векторный арифметический оператор. Алгоритм использует проведенную ранее разметку и на ее основе определяет порядок прохождения вершин дерева и подбор их в команды.

Процедуры работают с деревом, каждой вершине которого присвоен номер. По номеру определяются семантика вершины (вид операции или имя операнда) и проведенная разметка. Семантическая информация хранится в символьном виде.

Всякая внутренняя вершина проходит начальную обработку через основную процедуру CODE_GENERATOR (см. рис. 7), получающую в качестве аргументов номер обрабатываемой вершины и наименьший в момент вызова номер i свободного регистра. Методика обработки вершины определяется ее видом (унарная, бинарная или триадообразующая), а для вершины последнего из видов — ее типом (четная, нечетная), типом ее потомков и признаком S подбора их в триады.

Функция ARG (см. рис. 8) возвращает имя операнда для формируемой команды, соответствующее вершине, номер которой передан в функцию как параметр. Если вершина — лист, то имя содержится в семантическом поле вершины. Если вершина внутренняя, то именем является адрес регистра R_i, который будет содержать результат вычисления в этой вершине. При этом производится вызов процедуры CODE_GENERATOR с передачей ей в качестве параметров номеров этой вершины и регистра i. Так как при этом используется i-й регистр, то номер i первого свободного регистра увеличивается на 1 (оператор $i := i + 1$).

Если триадообразующая вершина является четной, то вызывается процедура EVEN, используемая для формирования бинарной операции и преобразования этой вершины в нечетную. Причем если у этой вершины более двух потомков, то она повторно попадает на обработку в рекурсивно вызываемую процедуру CODE_GENERATOR уже как нечетная.

Если триадообразующая вершина является нечетной, то в ее пределах формируются триадные операции. При формировании триад используется методика, рассмотренная в п. 2. Кроме того, если важен

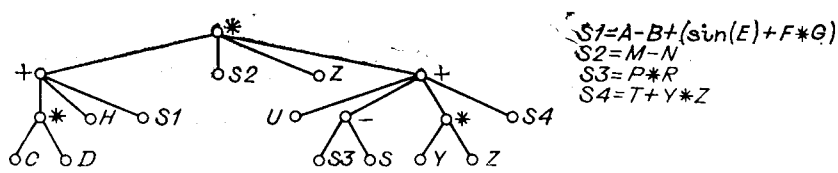

Рис. 6. Выделение скалярных подвыражений:
S1, *S2*, *S3*, *S4* — временные скалярные переменные

```
{node — номер вершины дерева}
{C[node] — семантика node (вид операции или имя операнда)}
{W[node]— вес node}
{S[node]— признак выбора сыновей node в триаду. S[node] =0, если выбор произволен,}
{S[node] > 0 — вес, который в триаде должен иметь лишь один сын}
{i — наименьший номер свободного регистра}
{Ri — i-регистр}
{son_k — k-й сын вершины node}
procedure CODE_GENERATOR (node : integer; var i : integer);
var i_old : integer;
    op1, op2, op3 : char;
begin
  if (C[node] — унарная операция) then WRITE (C[node], ARG(son_1, i), 'Ri')
  else if (C[node] — бинарная операция) then
    begin
      if (W[son_1] > W[son_2]) then
        begin op1 := ARG(son_1, i); op2 := ARG(son_2, i); end
      else
        begin op2 := ARG(son_2, i); op1 := ARG(son_1, i); end;
    WRITE (C[node], op1, op2, 'Ri');
  end
else if (C[node] — триадообразующая операция) then
  begin
    if (node — четная вершина) then
      begin EVEN (node, i, op1, op2);WRITE (C[node], op1, op2, 'Ri'); end
    else if (node — нечетная вершина) then
      begin
        i_old := i;
        выбрать сына son_m с максимальным значением веса W[son_m];
        if (у вершины node только четные сыновья) then
          begin
            EVEN (son_m, i, op1, op2); WRITE (C[node], op1, op2, 'Ri');
            i := i_old + 1; EVEN_AND_ODD (node, i);
          end
      else if (у вершины node только нечетные сыновья) then
        begin
          if (S[node] = 0) then
        begin
          WRITE (C[node], C[node], ARG(son_m, i), ARG(ANY(node), i),
            ARG(ANY(node), i), 'Ri');
          i := i_old + 1; ODD_IN_ANY_ORDER(node, i);
        end
  else if (S[node] > 0) then
    begin
      WRITE (C[node], C[node], ARG(son_m, i), ARG(SEQ(node), i),
        ARG(SLT(node), i), 'Ri');
      i := i_old + 1; ODD_IN_DEF_ORDER (node, i);
    end;
  end
else if (у вершины node и четные, и нечетные сыновья) then
  begin
    if (son_m — четная вершина) then
      begin
        EVEN (son_m, i, op1, op2);
        if (S[node] = 0) then op3 := ARG(ANY(node), i)
        else if (S[node] > 0) then op3 := ARG(SEQ(node,) i);
        WRITE (C[son_m], C[node], op1, op2, op3, 'Ri');
      end
      else if (son_m — нечетная вершина) then
        begin
          op3 := ARG(son_m, i);
          выбрать четного сына son_e;
          EVEN (son_e, i, op1, op2);
          WRITE (C[son_e], C[node], op1, op2, op3, 'Ri');
        end;
      i := i_old + 1; EVEN_AND_ODD(node, i);
      if (у node остались нечетные сыновья) then
          if (S[node] = 0) then ODD_IN_ANY_ORDER(node, i)
          else if (S[node] > 0) then ODD_IN_DEF_ORDER(node, i);
        end;
      end;
    end;
end;
```

Рис. 7. Основная процедура генерации кода CODE_GENERATOR

```
function ARG (node : integer; var i : integer) : char;
begin
  if (вершина node — лист) then ARG := C[node]
  else begin ARG := 'Ri'; CODE_GENERATOR (node, i); i := i + 1; end;
end;
function ANY (node : integer) : integer;
begin
  из сыновей node выбрать любого нечетного сына son_k;
  ANY := son_k;
end;
function SEQ (node : integer) : integer;
begin
  if (среди нечетных сыновей node есть с заданным весом S[node]) then
    begin
      из нечетных сыновей node выбрать сына son_k с весом S[node];
      SEQ := son_k;
    end;
  else SEQ := 0;
end;
function SLT (node : integer) : integer;
begin
  из нечетных сыновей node выбрать сына son_k с весом меньше S[node];
  SLT := son_k;
end;
procedure EVEN (node : integer; var i : integer; var op1, op2 : char);
var t : integer;
begin
  if (у вершины node два сына) then
    if (W[son_1] > W[son_2]) then
      begin op1 := ARG(son_1, i); op2 := ARG(son_2, i); end
    else
      begin op2 := ARG(son_2, i); op1 := ARG(son_1, i); end
  else
    begin
      пометить вершину node как нечетную;
      t := SEQ(node); op1 := ARG(node, i); op2 := ARG(t, i);
    end;
end;
procedure ODD_IN_ANY_ORDER (node : integer; var i : integer);
var i_old : integer;
begin
  i_old := i;
  while (у node есть сыновья) do
    begin
      WRITE (C[node], C[node], ARG(ANY(node), i), ARG(ANY(node), i),'Ri'', Ri');
      i := i_old;
    end;
end;
procedure ODD_IN_DEF_ORDER (node : integer; var i : integer);
var i_old, son : integer;
begin
i_old := i; son := SEQ(node);
  while ((son < > 0) AND (у node есть сыновья)) do
    begin
      WRITE (C[node], C[node], ARG(son, i), ARG(SLT(node), i), 'Ri', 'Ri');
      i := i_old; son := SEQ(node);
    end;
  if (у node есть сыновья) then ODD_IN_ANY_ORDER (node, i);
end;
procedure ODD_AND_EVEN (node : integer; var i : integer);
var i_old : integer;
    op1, op2 : char;
begin
  i_old := i;
  while (у node есть четные сыновья) do
    begin
      из сыновей node выбрать четного сына son_e;
      EVEN (son_e, i, op1, op2); WRITE (C[son_e], C[node], op1, op2, 'Ri', 'Ri');
      i := i_old;
    end;
end;
```

Рис. 8. Функции и процедуры, используемые процедурой CODE_GENERATOR

подбор потомков по весу ($S > 0$), то они выбираются так, чтобы количество потомков с весом S в триаде было не больше определенного числа. Для выбора потомков определенного веса используются функции ANY, SEQ и SLT. Функция ANY выбирает нечетного потомка с любым весом, SEQ — с весом, равным признаку выбора S, и функция SLT — с весом, меньшим S (см. рис. 8).

В первую триаду выбираются либо три нечетных потомка, либо один четный и один нечетный. Причем для этой триады всегда берется потомок с не меньшим, чем у всех других потомков вершины, весом. Для формирования последующих триад вызывается одна из процедур: ODD_IN_DEF_ORDER, ODD_IN_ANY_ORDER или ODD_AND_EVEN (см. рис. 8). Результат вычисления первой триады записывается в регистр с номером i, полученным процедурой CODE_GENERATOR в качестве аргумента, а сам номер при этом увеличивается на единицу (операторы $i := i\_old + 1$ на рис. 7, где i_old содержит исходное значение i). Результат вычисления всех последующих триад, порожденных текущей вершиной, записывается в тот же самый регистр, а далее он участвует в этих триадах и в качестве одного из аргументов (см. рис. 3). Важно отметить, что если подбор по весу существен, то для каждой входящей в триаду группы потомков дальнейшая обработка производится в порядке убывания их веса. Это гарантирует, что число регистров, необходимое для вычисления значения текущей вершины, будет равно ее весу.

Если у текущей вершины только нечетные потомки и подбор по весу важен, то вызывается процедура ODD_IN_DEF_ORDER; если же он не важен, то — ODD_IN_ANY_ORDER. Если у вершины есть четные потомки, то вызывается процедура ODD_AND_EVEN (см. рис. 7).

Процедура ODD_IN_ANY_ORDER формирует все возможные триады в пределах данной вершины из нечетных потомков вне зависимости от их веса. Для выбора потомков в триады используется функция ANY. Так как вызов функции ARG может менять значение номера регистра i, то он запоминается в локальной переменной i_old, которая используется для его восстановления после каждой сформированной триады.

В процедуре ODD_IN_DEF_ORDER учитывается вес потомков, поэтому для их выбора используются функции SEQ и SLT. Если в какой-то момент у всех оставшихся потомков вершины вес меньше S, то для формирования остальных триад вызывается процедура ODD_IN_ANY_ORDER.

Процедура ODD_AND_EVEN выбирает для триады одного четного потомка, а в качестве третьего операнда, как и две предыдущие процедуры, использует регистр, содержащий результаты вычисления в текущей вершине. Для обработки четной вершины вызывается процедура EVEN. Если после использования всех четных потомков у вершины еще остались нечетные потомки, то для формирования остальных триад в зависимости от признака S вызывается процедура ODD_IN_ANY_ORDER или ODD_IN_DEF_ORDER.

Все перечисленные процедуры и функции, а также переменная i должны быть объявлены в некоторой содержащей их главной программе. Из нее инициализируется работа алгоритма командой CODE_GENERATOR (root, i), где root — номер корня дерева, а переменной i предварительно присвоено значение 1. Последовательность сгенерированных после этого команд определит искомый порядок вычислений. Результат вычисления всего выражения (корня дерева) запишется в первый регистр.

На рис. 9 показан пример последовательности команд, построенный для размеченного ранее дерева (см. рис. 4).

Корректность работы алгоритма вытекает из следующих утверждений. При первом проходе по дереву отсекаются максимально возможные скалярные поддеревья, что сокращает общее число команд вычисления

Номера полей						Номера полей					
1	2	3	4	5	6	1	2	3	4	5	6
*	—	P	R	S	R1	sin		E			R2
*	+	Y	Z	R1	R1	*	+	F	G	R2	R2
/		W	V		R2	*	+	C	D	R2	R2
+	+	R2	U	R1	R1	—	+	A	B	R2	R2
+	*	R1	T	Z	R1	+	*	R2	H	R1	R1
—	*	M	N	R1	R1						

Рис. 9. Последовательность команд. Значение полей команд: 1, 2 — операции; 3, 4, 5 — операнды; 6 — результат

с векторами. По индукции доказывается, что вызов процедуры CODE_GENERATOR (node, *i*) правильно вычисляет значение в вершине node, помещая его в *i*-й регистр. В процессе разметки дерева каждой вершине присваивается минимальный возможный вес, при этом для триадообразующих вершин учитывается последующая методика формирования триад. Соответственно и корню дерева присваивается минимально возможный вес. Так как вес корня дерева равен необходимому для вычисления его значения числу регистров, то тем самым это число минимально. Процедуры ODD_IN_ANY_ORDER, ODD_IN_DEF_ORDER и ODD_AND_EVEN, прежде чем вернуть управление, формируют для текущей вершины максимальное число триад. В результате во всем дереве строится максимальное число триад.

Заключение. В работе рассмотрены вопросы оптимизации арифметических векторных операторов применительно к векторному процессору А-12. Проанализированы основные направления такой оптимизации. Предложен алгоритм, решающий поставленные задачи. Алгоритм может быть использован для оптимизации применительно к ЭВМ схожей архитектуры (например, GRAY-1, FPS-164) и позволяет перенастройку на различные допустимые сочетания операций, входящих в триады.

В ходе своей работы алгоритм использует свойства ассоциативности и коммутативности арифметических операций. Однако при необходимости можно явно определить желаемую последовательность вычислений, так как последовательность действий, задаваемая скобками, сохраняется.

Описанный алгоритм реализован на языке Паскаль и апробирован как составная часть программной разработки препроцессора, векторизующего циклические участки пользовательских программ.

СПИСОК ЛИТЕРАТУРЫ

1. **Мучник В. Б., Шафаренко А. В.** EVAL — язык для программирования параллельных компьютеров.— Новосибирск, 1985.— (Препр/ИАиЭ СО АН СССР; 281).
2. **Меткалф М. Оптимизация в Фортране.**— М.: Мир, 1986.
3. **Бродский И. И., Козлачков В. А., Коршевер И. И. и др.** Высокопроизводительный периферийный векторный процессор А-12 // Вопросы кибернетики.— 1985.— № 104.
4. **Березовский М. А., Минкин А. Л.** Оптимизирующий препроцессор программ на Фортране для матричного процессора А-12 // Автометрия.— 1988.— № 2.
5. **Ахо А., Ульман Д.** Теория синтаксического анализа, перевода и компиляции.— М.: Мир, 1978.— Т. 2.
6. **Кнут Д.** Искусство программирования для ЭВМ.— М.: Мир, 1976.— Т. 1.

Поступила в редакцию 10 февраля 1988 г.