УДК 004.27

Использование мелко гранулярного параллелизма в процессоре с архитектурой управления потоком данных

Н.И. Дикарев, Б.М. Шабанов, А.С. Шмелёв

Межведомственный суперкомпьютерный центр РАН,

nic@jscc.ru, shabanov@jscc.ru, guest8993@rambler.ru

***Аннотация* — Процессор с архитектурой управления потоком данных (потоковый процессор) может обеспечить значительно более высокую производительность за счет того, что поиск готовых к выполнению команд в потоковом процессоре осуществляется в окне из порядка 10000 команд, а не из 100 команд, как в современных микропроцессорах. Однако поскольку известным потоковым процессорам требуется в 2 - 3 раза больше команд для выполнения программы и ряду других недостатков они оказались не конкурентно способны по отношению к процессорам традиционной архитектуры. На примере выполнения программ перемножения матриц и пузырьковой сортировки показано, что в разрабатываемом векторном потоковом процессоре можно достичь значительно более высокой производительности за счет в 2 – 3 раза меньшего числа выполняемых команд и способности распараллеливать код на десятки мелких гранул.**

***Ключевые слова* — векторный процессор, архитектура управления потоком данных, мелкоструктурный параллелизм, скалярная производительность.**

I. Введение

Процессор с архитектурой управления потоком данных (потоковый процессор) имеет потенциал обеспечить наивысшую среди процессоров производительность за счет того, что параллелизм выполнения команд закладывается уже при составлении графа программы, и его не требуется выявлять с помощью сложной аппаратуры из последовательности команд, заданной счетчиком команд, как это имеет место в фон-неймановском процессоре. Программой в потоковых ЭВМ является ориентированный граф, узлами которого являются команды, а информация по дугам передается в виде токенов, содержащих поле данных (значение операнда) и поле контекста передаваемых данных. Этот контекст однозначно определяет, куда должен быть отправлен токен, то есть содержит номер команды приемника в графе программы, а также содержит номера индекса, итерации и запуска подпрограммы, которые позволяют в параллель выполнять различные итерации вложенных циклов и различные запуски процедур на одном и том же графе программы. Причем вне зависимости от места двухвходовой команды в графе она выдаётся на исполнение по прибытию на вход последнего из пары токенов операндов с одинаковым контекстом. После вычисления результата в исполнительном устройстве (ИУ) новые токены со значением результата отправляются на входы последующих команд согласно графу программы, а использованные токены операндов уничтожаются. При этом в потоковом процессоре в отличие от фоннеймановского отсутствует центральное устройство управления (счетчик команд), а параллелизм выполняемых команд определяется в динамике по приходу операндов на входы команд в децентрализованной схеме. А именно, устройство поиска готовых к выполнению команд – память поиска пар (готовых операндов) можно выполнить в виде многих работающих в параллель модулей, обеспечивающих работой такое же большое число ИУ.

Цель данной работы – показать, что в разрабатываемом векторном потоковом процессоре можно достичь значительно более высокую как векторную, так и скалярную производительность по сравнению с фоннеймановским процессором за счет в 2 – 3 раза меньшего числа выполняемых команд и способности распараллеливать код на десятки мелких гранул.

II. Препятствия на пути создания потокового процессора

На рис. 1 показана структурная схема потокового процессора, в которой устройство поиска готовых к выполнению команд – память поиска пар (ППП) готовых операндов расслоена на K модулей. Токены с выходов каждого из K ИУ направляются на вход нужного модуля ППП с помощью коммутатора. Буферы FIFO на входах коммутатора предотвращают блокировку выхода ИУ в случае конфликта за выход коммутатора, когда в одном такте несколько ИУ посылают токены в один и тот же модуль ППП. В этом случае на выход коммутатора проходит токен с одного из входов (с более высоким приоритетом), а остальные сохраняются в буфере FIFO на входе коммутатора для прохождения в последующие такты. Тем самым буферы FIFO используются для сглаживания неравномерности во времени прихода токенов по модулям ППП. Буферы FIFO должны быть достаточной ёмкости, поскольку переполнение любого из них из-за кольцевой структуры управления (см. рис. 1) с высокой вероятностью приводит к блокировке всех устройств в этой цепи, и процессор попадает в неработоспособное состояние.

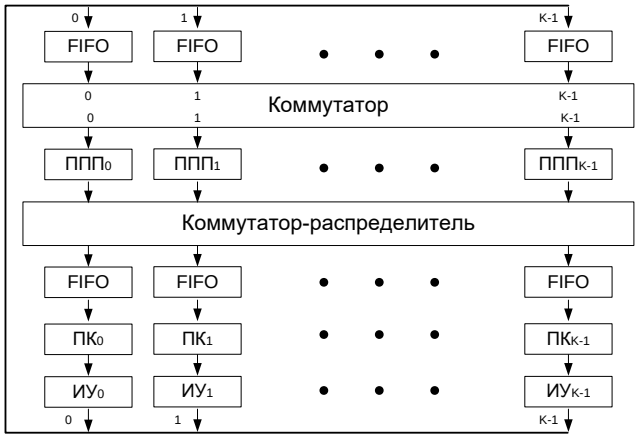

Рис. 1. Структурная схема потокового процессора

Приход в ППП первого по времени токена операнда приводит к его записи в свободную ячейку ППП, а приход второго токена операнда вызывает выдачу команды на исполнение и освобождение ячейки ППП, занятой первым операндом. Тогда готовая команда в виде пакета из двух значений операндов вместе с контекстом передается с выхода ППП через коммутатор - распределитель на вход того специализированного ИУ или группы однотипных ИУ, где она будет выполняться, предварительно проходя через буфер FIFO и модуль памяти команд (ПК). Буферы FIFO, как и аналогичные буферы на входе коммутатора, предотвращают блокировку выхода коммутатора-распределителя, если ИУ занято обслуживанием предыдущей команды. Что же касается модулей ПК, то это обычные линейно адресуемые запоминающие устройства, из которых по номеру команды в графе читается её код операции и информация о том, сколько токенов со значением результата нужно сформировать и на входы каких команд их направить.

Такой принцип работы позволяет одновременно выполнять в потоковом процессоре команды из нескольких программ, и в отличие от фон-неймановского процессора у него нет аппаратуры, отвечающей за выполнение каждого из потоков команд. Необходимо лишь чтобы ёмкость модулей ПК была достаточной для хранения команд из разных программ, а также, чтобы не переполнялись модули ППП. Заметим, что динамическая модель потокового процессора, в которой поиск готовых команд в ППП осуществляется не только по совпадению номера команды у токенов операндов, но и еще нескольких полей, например, номера индекса, итерации и запуска подпрограммы, позволяет выявлять больше параллелизма в программе [1]. Однако это приводит к сложности реализации ППП, поскольку модули ППП должны иметь не только большую ёмкость, чтобы не допускать их переполнения, но и осуществлять ассоциативный поиск. Кроме того, модули ПК и ППП должны иметь высокое быстродействие, поскольку вносимая ими задержка вместе со временем вычисления результата в ИУ и задержкой прохождения токенов через два коммутатора определяет время выполнения команды в цепочке команд, связанных зависимостью данных. Поэтому потоковые процессоры значительно проигрывают в производительности фон-неймановскому процессору на чисто последовательном коде и могут реализовать свое преимущество в производительности лишь при наличии в программе достаточного уровня параллелизма. С другой стороны, нельзя допускать избыточного параллелизма, поскольку переполнение буферов готовых команд на входах ИУ или модулей ППП приводит процессор в тупиковую ситуацию (deadlock). Для ограничения динамического параллелизма при выполнении программы приходится использовать как программные, так и аппаратные средства, что приводит к увеличению числа выполняемых команд и (или) росту аппаратных затрат.

Другая причина сложности аппаратной реализации потокового процессора связана с использованием коммутатора, с помощью которого осуществляется пересылка токенов с выходов каждого из K ИУ на входы любого из K модулей ППП. Такой коммутатор должен иметь высокую пропускную способность по каждому из входов и одновременно малую задержку передачи до выхода, что при передаче пакетов малой длины (токенов) трудно выполнимо уже при K >16. Если же учесть, что в потоковом процессоре приходится выполнять в 2 - 3 раза больше команд на программах научных задач по сравнению с фон-неймановским процессором [2], то оказывается, что преимущества в реальной производительности у него фактически нет. Заметим, что сам принцип работы потокового процессора приводит к появлению избыточности в числе выполняемых команд, поскольку, как правило, одна команда формирует не более двух токенов для передачи результата. Если же результат используется в качестве операнда более двух раз, то в граф программы приходится вводить команды дублирования, единственное назначение которых – указать на входы каких ещё команд следует послать значение результата. Кроме того, при выполнении ветвления каждое из значений данных, используемых в ветвях программы, необходимо направить в нужную ветвь с помощью отдельной команды переключателя, в то время как в традиционном процессоре требуется лишь одна команда условного перехода. Однако главная причина избыточности числа команд у потокового процессора связана с обработкой массивов данных.

В общем случае потоковому процессору не нужны другие устройства памяти кроме ППП. При создании массива A его элементы A(i) поступают в ППП в виде токенов с различными значениями индекса i в поле контекста. Каждый из этих элементов либо находит себе пару в ППП, и команда выдается на выполнение, либо записывается в ППП и ждет второго операнда. Однако при увеличении размера обрабатываемых массивов естественный параллелизм у многих задач может быть столь велик, что обрабатывать все их элементы в параллель невозможно, поскольку для этого не хватит ресурсов ни в какой реальной системе [3]. Для ограничения параллелизма элементы массивов приходится хранить в ППП на входах команд синхрониза-

ции, с помощью которых выдается разрешение на продолжение вычислений, что и приводит к появлению большого числа избыточных команд. Кроме того, фактически статическое хранение массивов в ППП требует многократного увеличения ёмкости этой памяти, что недопустимо как из-за роста аппаратных затрат для осуществления ассоциативного поиска, так и снижения её быстродействия. Поэтому потоковый процессор часто содержит обычную память для хранения массивов, и эта функция снимается с ППП, но тогда для исключения конфликтов информационной зависимости необходима синхронизация обращений к его отдельным элементам по записи и чтению, что приводит к дополнительной аппаратуре и (или) к появлению избыточных команд.

Отмеченные выше сложности аппаратной реализации потокового процессора привели к тому, что ни в одном из многочисленных проектов по его созданию, пик которых за рубежом пришелся на 80-е годы прошлого века, не удалось достичь более высокой производительности по сравнению с фон-неймановским процессором, и к концу 2000-х годов таких проектов практически не осталось. Заметим лишь, что это относится к разработке универсальных потоковых процессоров, а не специализированных, поскольку, например, потоковые процессоры цифровой обработки сигналов не только разрабатывались, но и выпускались [4]. Это объясняется тем, что при их разработке отсутствуют такие проблемы создания универсальных потоковых процессоров, как избыточный параллелизм и необходимость хранения массивов в памяти большой ёмкости. И ещё одно уточнение – таких проектов не осталось за рубежом, но они есть в России, об одном из них пойдёт речь ниже.

III. Векторный потоковый процессор

Разрабатываемый в МСЦ РАН потоковый процессор является векторным, и само наличие векторной обработки позволяет в сотни раз уменьшить число выполняемых команд, поскольку одна векторная команда заменяет собой цикл с независимыми итерациями с числом итераций VL, где VL – длина вектора. Для записи результатов векторных команд и хранения массивов в разрабатываемом векторном потоковом процессоре (ВПП) используется линейно адресуемая память, содержащая два уровня - память векторов (ПВ) большой емкости, реализованную на микросхемах динамической памяти, и быструю локальную память векторов (ЛПВ) значительно меньшей емкости, размещенную на процессорном кристалле. Распределение ресурса ПВ и ЛПВ в ВПП осуществляется на аппаратном уровне, и в качестве единицы фрагментации используется вектор с фиксированным числом слов VLmax=256. В этом случае входящее в состав ВПП устройство распределения памяти ведет список свободных векторов для ПВ и ЛПВ, и выделяет для записи результата векторной команды свободный вектор из затребованного списка. Тогда адрес начального элемента вектора вместе с фактической длиной VL, которая меньше или равна VLmax, и битом уровня памяти составляют аппаратный указатель (имя) вектора. Этот указатель однозначно определяет вектор для его использования в качестве операнда и передается в поле данных токена на входы последующих команд согласно графу программы. После выполнения последней из этих команд адрес вектора возвращается обратно в список свободных векторов. В результате размещение векторов и массивов в ПВ происходит динамически по мере их создания (уничтожения) и без участия операционной системы. Тем самым, несмотря на использование в ВПП линейно адресуемой памяти работа с этой памятью приближена к принципу работы потокового процессора. Что же касается больших массивов, то их приходится хранить в ПВ в виде «векторов указателей», то есть векторов, элементами которых являются указатели векторов подмассивов, как это показано на рис. 2.

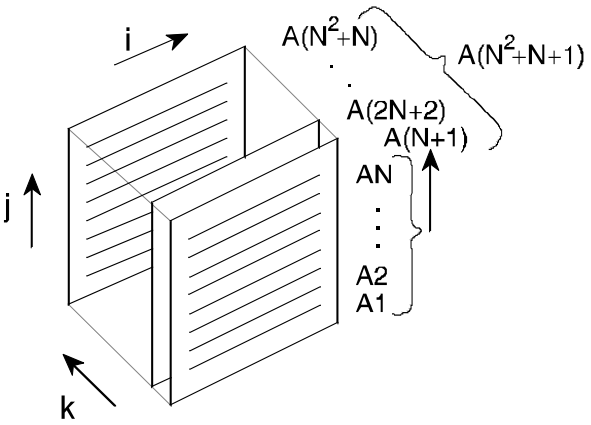

Рис. 2. Представление трехмерного массива векторами-указателями

Заметим, что нумерация векторов A1, A2, AN, A(N+1) и других на рис. 2 не имеет никакого отношения к физическим адресам этих векторов в ПВ. Как уже отмечалось выше, задача определения адреса для записи вектора результата в ПВ или ЛПВ решается аппаратно, путем ведения списков свободных векторов. Такое хранение массивов позволяет одной командой формирование потока (ФП) выдать указатели всех векторов строк, содержащиеся в векторе-указателе матрицы, и выполнить над всеми строками матрицы одинаковые векторные операции [5]. Тем самым удается векторизовать не только самый внутренний, но и следующий из вложенных циклов в программе, что позволяет значительно сократить адресные и другие вычисления, выполняемые в скалярных ИУ (СИУ) ВПП, и соответственно повысить производительность векторных ИУ (ВИУ) относительно СИУ. Например, в программе перемножения матриц число команд в цикле по вычислению столбца матрицы результата (подпрограмма SGENV [6]) в ВВП уменьшается в 2,7 раза. Так в векторном процессоре CRAY Y-MP в каждой итерации этого цикла нужно выполнить 8 команд (3 векторных и 5 скалярных), в то время как в ВПП 3 команды, одна из которых скалярная [5]. Значительное уменьшение общего числа выполняемых команд и особенно скалярных команд в ВПП достигается за счет использования команд ФП, которые позволяет исключить из тела цикла команды обращения к памяти по

чтению векторов Ak в матрице A и элементов B(k,j) в матрице B, а также команды вычисления начальных адресов для чтения векторов Ak и увеличения номера итерации k на 1. В результате производительность каждого из двух ВИУ в ВПП можно поднять до 128 флоп в такт и уменьшить общее число скалярных и векторных ИУ, так же, как и модулей ППП до 4. Тогда в качестве коммутатора (см. рис. 1) можно использовать матричный коммутатор 4x4 с малой задержкой, высокой пропускной способностью и небольшими аппаратными затратами.

Как показали результаты моделирования программы перемножения матриц, полученные на VHDL модели ВПП и приведенные в [5], программа автоматически распараллеливается на вычисление нескольких столбцов матрицы результата, так чтобы обеспечить полную загрузку сумматоров и умножителей в векторном АЛУ. Дело в том, что цикл, выполняемый в подпрограмме SGENV, имеет зависимость по данным, когда вектор, созданный сумматором в предыдущей итерации цикла, используется в качестве его операнда в следующей итерации. Поэтому итерации данного цикла приходится выполнять последовательно, что приводит к большому периоду следования команд векторного сложения в ВВП и, как следствие, его низкой производительности. Здесь проявляется известный недостаток потоковой архитектуры, заключающийся в низкой производительности при выполнении последовательных команд.

Однако в ВПП этот недостаток компенсируется возможностью осуществлять поиск готовых команд в гораздо большем окне (в 100 раз) по сравнению с лучшими процессорами традиционной архитектуры, что позволяет не останавливать работу умножителя, поставляющего вектора операнды на другой вход сумматора. В результате происходит инициирование следующей подпрограммы SGENV, затем ещё одной и так до тех пор, пока векторный сумматор не станет выполнять команды из нескольких подпрограмм, то есть вычислять в параллель несколько столбцов матрицы результата, что устраняет простои в его работе, и производительность ВПП приближается к пиковой. При этом число команд, ожидающих парного операнда у сумматора в ВПП, растёт пропорционально его производительности, и при производительности ВПП равной 256 флоп в такт полная загрузка векторного сумматора достигается при одновременном выполнении 20 подпрограмм SGENV, а число токенов в ППП достигает 3300. Кроме того, в [5] было показано, что в ВПП можно не только повысить производительность процессора до 256 флоп в такт, что в 4 раза выше, чем у последнего векторного процессора NEC SX-ACE, но сохранять ее при значительно меньших размерах обрабатываемых массивов. Последние изменения, внесенных в схему и VHDL модель ВПП, заключались в замене раздельных сумматоров и умножителей с плавающей запятой в векторном блоке процессора на сдвоенные сумматоры и умножители (FMA) и были направлены на уменьшение числа портов доступа от этих АЛУ к ЛПВ. Это позволило уменьшить число конфликтов занятого банка в ЛПВ и повысить производительность ВПП на программе перемножения матриц размером 512x512 с 222 до 243 флоп в такт или до 95% от пиковой производительности в 256 флоп в такт.

IV. Использование мелко гранулярного параллелизма на скалярной обработке в ВПП

Покажем, что число выполняемых команд в ВПП можно сократить в несколько раз и значительно повысить производительность не только на хорошо векторизуемой программе, какой является программа перемножения матриц, но и на чисто скалярном коде, например, когда между итерациями внутреннего цикла есть зависимость по данным. В качестве примера такой чисто последовательной программы будет рассмотрена программа пузырьковой сортировки. Простой алгоритм этой программы хорошо известен, также как и то, что число операций при его выполнении растет как $O(N^2)$, в то время как существуют гораздо более эффективные алгоритмы сортировки с числом операций $O(N \log_2(N))$, где N – длина массива. Поэтому практическая ценность рассматриваемой ниже программы не в целесообразности её непосредственного использования, хотя при малых N она, по-видимому, есть, а в том, чтобы показать, как её можно распараллелить для получения более высокой производительности в ВПП. Причем невозможно сделать то же самое на одном процессоре фон-неймановской архитектуры.

Граф программы пузырьковой сортировки для ВПП приведен на рис. 3. Эта программа содержит два вложенных цикла. Во внутреннем цикле производится поочередное сравнение элементов массива a(i) с a(i+1) и их перестановка, если a(i)>a(i+1). В результате, при изменении индекса i от 0 до N-1, где N – длина массива, наибольший элемент массива перемещается в крайне правую позицию (N-1). Далее во внешнем цикле длина массива уменьшается на 1, и процесс повторяется N-1 раз.

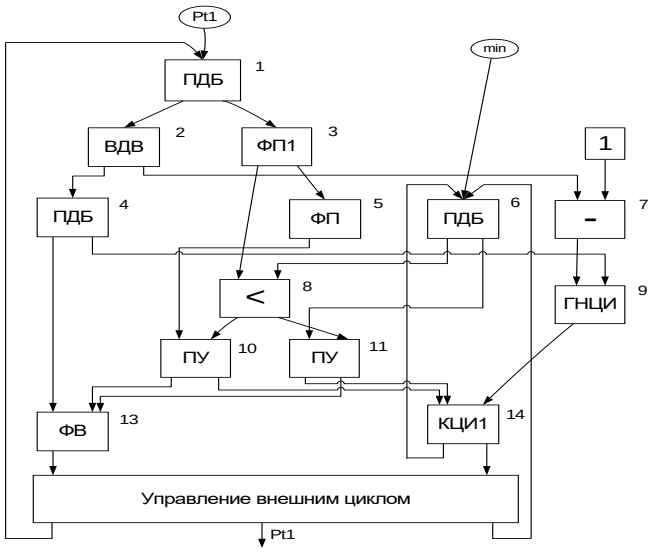

Рис. 3. Граф программы пузырьковой сортировки для ВПП

Выполнение программы в ВВП начинается с прихода на вход команды дублирования (ПДБ) 1 токена с указателем сортируемого массива (вектора) Pt1. Этот указатель определяет нахождение всех элементов массива в ПВ – обычной линейно адресуемой памяти ВПП, и содержит два параметра: адрес начального (0-го) элемента массива в ПВ и число элементов (длину вектора). При выполнении команды 1 в СИУ формируются два токена с указателем Pt1, которые посылаются на входы команд 2 и 3. Командой 2 выдаётся длина вектора, а команда ФП 3 выполняется в ВИУ. Эта команда производит чтение элементов массива в ПВ и формирует N токенов из элементов a(i), причем индекс i записан в соответствующее поле контекста каждого токена. Эти токены передаются по левому выходу команды 3 (см. рис. 3), а по правому выходу передаётся одиночный токен с указателем массива Pt1 на вход ещё одной команды ФП 5. Тем самым для формирования второй серии токенов с элементами a(i) используется ещё одна групповая команда ФП вместо N команд дублирования.

Тогда в каждой итерации внутреннего цикла в ВПП выполняются лишь 5 команд. Это команда 8, сравнивающая элемент a(i) с максимальным элементом из итерации i-1 (для i=0 это минимальное число min из диапазона сортируемых чисел, поступающее из внешней машины). Затем команды передача условная (ПУ) 10 и 11 передают соответственно элемент a(i) и максимальный элемент из предыдущей итерации на свой левый выход, если результат сравнения true, и на правый выход, если результат сравнения false. Команда КЦИ1 (конец цикла по итерации) 14 посылает максимальный из сравниваемых элементов со своего левого входа на вход команды ПДБ 6 для выполнения следующей итерации (увеличив на 1 значение индекса в контексте токена), если это не последняя итерация и индекс не равен N-1. В результате выполнения N итераций внутреннего цикла максимальный из элементов a(i) поступает в блок графа «Управление внешним циклом» с правого выхода команды 14. При этом с выхода команды формирование вектора (ФВ) 13 поступает указатель нового вектора Pt2, в который записаны остальные элементы массива в порядке их поступления на вход команды 13. Тогда для перехода к следующей итерации внешнего цикла требуется удалить 0-й элемент вектора Pt2, содержащий значение min, что достигается сдвигом вектора на 1 элемент влево с записью в крайне правый элемент максимального элемента. Результат команды сдвига нужно записать по адресу исходного массива Pt1 и, уменьшив длину вектора на 1, послать указатель вектора на вход команды ПДБ 1, а на вход команды ПДБ 6 - послать токен со значением min. Тогда после выполнения N-1 итерации внешнего цикла вектор Pt1 будет содержать результат сортировки, и его указатель направляется во внешнюю машину.

Таким образом, каждая итерация внутреннего цикла в ВПП требует выполнения команд 6, 8, 10, 11 и 14, причем остальные команды, показанные на рисунке, выдаются на выполнение лишь один раз на все итерации внутреннего цикла. Для сравнения фон-неймановскому процессору на одну операцию сравнения и перестановки двух элементов массива требуется 12 команд, если a(i)>a(i+1), и 8 команд, если a(i)<a(i+1) [7]. Поскольку первое условие встречается значительно чаще, то для выполнения программы пузырьковой сортировки ВВП требуется лишь 42% от числа команд, выполняемых традиционным процессором. Правда на одну операцию сравнения и перестановки современному процессору, способному выполнять до четырёх команд в такт, при времени выборки элемента из памяти в 2 такта требуется 6 тактов [7], а по результатам моделирования ВВП – 28 тактов. Конечно, 6 тактов это минимально возможное время, необходимое на базовую операцию сравнения и перестановки современному процессору. Если учесть, что время выборки из оперативной памяти с учётом промахов в кэш будет больше двух тактов и есть потери времени из-за неверно предсказанного перехода при проверке условия a(i)>a(i+1), то среднее время на операцию сравнения и перестановки должно быть не менее 8 – 10 тактов. Но и в этом случае, несмотря на меньшее число выполняемых команд в ВВП, его производительность при выполнении базовой операции оказывается примерно в 3 раза ниже, чем у процессора традиционной архитектуры.

Причина в том, что на цепочках с последовательным выполнением команд (так выполняются 4 из 5 команд внутреннего цикла, показанного на рис. 3) производительность потоковых процессоров существенно ниже. Однако, если длина сортируемого массива в рассматриваемой программе больше аппаратной длины вектора ВПП (256 элементов), то данный массив будет храниться в ПВ и обрабатываться в программе блоками (векторами) с числом элементов равным 256, упакованными в вектор-указатель (см. рис. 2). При этом исходный внутренний цикл в описанной выше программе пузырьковой сортировки для ВПП будет заменен двумя вложенными циклами, которые будут вести обработку длинного одномерного массива, последовательно обрабатывая вектора с числом элементов меньше или равным 256. Тогда по мере перехода к сортировке следующего вектора данных из вектора-указателя длинного массива можно начинать сортировку ещё и только что записанного частично отсортированного вектора результата, то есть переходить к следующей итерации внешнего цикла в исходной программе. Тем самым в ВПП за счет возможности одновременного выполнения мелких гранул (блоков с числом элементов, меньше или равных 256) можно распараллелить скалярную обработку в программе пузырьковой сортировки, так чтобы СИУ могло работать без простоев.

Заметим, что такое распараллеливание алгоритма сортировки невозможно осуществить в современных суперскалярных процессорах, которые за счет поиска готовых команд в окне из примерно 100 команд могут выполнять в параллель команды из нескольких итераций самого внутреннего цикла, но не из нескольких вложенных циклов одновременно как в данном случае.

Не даст эффекта такое распараллеливание алгоритма и в существующих многопроцессорных системах с общей памятью из-за слишком большого времени на создание и запуск следующего потока.

Моделирование ВПП, которое содержит 2 СИУ, одно из которых выполняет двухвходовые команды, а другое – команды с одним операндом, показало, что такая модифицированная программа сортировки позволяет вести в параллель сортировку до 8 векторов на массиве из 16 векторов по 128 элементов в каждом. При этом среднее время на одну операцию сравнения и перестановки составило 4,53 такта, а требуемая ёмкость ППП - 2600 токенов. Тем самым, производительность ВПП на программе пузырьковой сортировки будет примерно в два раза выше, чем у суперскалярного процессора традиционной архитектуры, и её можно повысить ещё в несколько раз за счет увеличения числа СИУ.

V. Заключение

Таким образом, результаты моделирования ВПП показали, что в нём действительно можно достичь значительно более высокой производительности за счет в 2 – 3 раза меньшего числа выполняемых команд и способности распараллеливать код на десятки мелких гранул. Такое распараллеливание, как показано выше, позволяет компенсировать существенный недостаток потоковых процессоров, заключающийся в низкой производительности при выполнении цепочек из последовательных команд. При этом дополнительный параллелизм может быть выявлен как аппаратурой потокового процессора, например, в программе перемножения матриц, так и модификацией алгоритма, как в программе пузырьковой сортировки. Причем использование мелко структурного параллелизма с выполнением команд по готовности данных открывает перед программистом новые возможности распараллеливания программ, недоступные для применения в процессорах традиционной архитектуры.

Поддержка

Работа выполнена при частичной поддержке гранта РФФИ 13-07-01124.

Литература

[1] Dataflow Computers: Their History and Future. Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin War, 2008, John Wiley & Sons, Inc.

[2] Papadopoulos G.V., Traub K.R. Multithreading: A Revisionist View of Dataflow Architectures // Proc. 18-th Ann. Symp. on Computer Architecture. 1991. P. 342–351.

[3] Culler D.E. and Arvind, Resource Requirements of Dataflow Programs // Proc. 15-th Ann. Symp. on Computer Architecture. 1988. P. 141–150.

[4] Terada H. et al., DDMP's: Self-Timed Super-Pipelined Data-Driven Multimedia Processors // Proceedings of the IEEE. 1999. V. 87. № 2. P. 282–296.

[5] Дикарев Н.И., Шабанов Б.М., Шмелёв А.С. Векторный потоковый процессор: оценка производительности // Известия ЮФУ. Технические науки. 2014. № 12 (161). С. 36–46. URL: http://izv-tn.tti.sfedu.ru/wp-content/uploads/2014/12/4.pdf (дата обращения 29.04.2016)

[6] Hake J.F. and Homberg W. The Impact of Memory Organization on the Performance of Matrix Calculations // Parallel Computing, 1991, V. 17, № 2/3, P. 311–327.

[7] Smith J.E. and Sohi G.S. The Microarhitecture of Superscalar Processors // Proceedings of the IEEE, 1995, V. 83, № 12, P. 1609–1624.

The use of fine-grained parallelism in dataflow processor

N.I. Dikarev, B.M. Shabanov, A.S. Shmelev

Joined Supercomputer Center of RAS,

nic@jscc.ru, shabanov@jscc.ru, guest8993@rambler.ru

***Keywords* — vector processor, dataflow architecture, fine-grained parallelism, scalar performance.**

Abstract

Dataflow processor can offer the highest performance among scalar microprocessors due to natural parallelism of a dataflow graph as a program. Program in a dataflow computer is a directed graph where nodes are instructions and arches are data dependencies among the nodes. The data is conveyed from one instruction to another in packets called tokens. Each token contains data field (operand) and context field. The context field consists of the destination instruction number in graph, its index, iteration and subroutine numbers, thus different iterations of nested loops and different subroutine calls of the same graph can be executed concurrently. Two-input instructions are issued to execution as soon as both tokens with the same context arrive. After calculating the results in the functional unit new tokens are sent to the inputs of the following instructions according to the graph of the program. Once used tokens are destroyed.

Unlike von-Neumann processor, dataflow processor has no program counter, and the matching store dynamically reveals instruction parallelism according to operand availability. It is possible to interleave matching store to many units so as many independent functional units can be loaded.

In this paper we show that considerably higher performance on vector and scalar codes could be reached in vec-

tor dataflow processor due to 2-3 times less number of executed instructions and the ability to parallelize the code on fine grains. High performance is confirmed by the results of simulation in matrix multiplication and bubble sort programs.

Exploiting of fine-grained parallelism in vector dataflow processor is possible due to considerably wider window of execution, which holds 10 thousand instructions as compared with 100 instructions in modern superscalar processors.

REFERENCES

[1] Dataflow computers: Their history and future. Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin War, 2008, John Wiley & Sons, Inc.

[2] Papadopoulos G.V., Traub K.R. Multithreading: A revisionist view of dataflow architectures // Proc. 18-th Ann. Symp. on Computer Architecture, 1991, P. 342–351.

[3] Culler D.E. and Arvind, Resource requirements of dataflow programs // Proc. 15-th Ann. Symp. on Computer Architecture, 1988, P. 141–150.

[4] Terada H. et al., DDMP's: Self-Timed Super-Pipelined Data-Driven Multimedia Processors // Proceedings of the IEEE. 1999. V. 87. No. 2. P. 282–296.

[5] Dikarev N.I., Shabanov B.M., Shmelev A.S. Vector dataflow processor: performance evaluation // Izvestiya SFedU. Engineering Sciences, 2014. no. 12 (161), pp. 36–46. URL: http://izv-tn.tti.sfedu.ru/wp-content/uploads/2014/12/4.pdf (29.04.2016) (in Russian).

[6] Hake J.F. and Homberg W. The impact of memory organization on the performance of matrix calculations // Parallel Computing, 1991, V. 17, No. 2/3, P. 311–327.

[7] Smith J.E. and Sohi G.S. The Microarhitecture of Superscalar Processors // Proc. IEEE. 1995. V. 83, No. 12. P. 1609–1624.