Информатика. Построение и анализ алгоритмов.

П. В. Трифонов

8 августа 2007 г.

Оглавление

Глава 1

Введение

Современные фундаментальные и прикладные исследования и инженерная деятельность характеризуются постоянным ростом размерности и сложности решаемых задач. Несмотря на постоянное увеличение производительности вычислительных систем, среднее время расчетов, необходимых для решения типовых научно-технических задач, имеет тенденцию к возрастанию. В связи с этим возникает задача повышения эффективности вычислительных ядер существующего и вновь разрабатываемого программного обеспечения. Существует несколько путей повышения производительности вычислительных систем:

1. Применение более быстрого процессора. При этом необходимо учитывать, что в настоящее время совершенствование процессоров происходит в основном за счет реализации различных форм параллелизма, эффективное использование которых требует соответствующей модификации исходного кода программ.

2. Применение алгоритмов с уменьшенной вычислительной сложностью. Однако оказывается, что классическая теория быстрых алгоритмов не учитывает многих особенностей современных процессоров, что приводит к снижению эффективности программной реализации быстрых алгоритмов.

3. Применение параллельных вычислений. При реализации параллельных программ возникают дополнительные сложности, связанные с обеспечением синхронизации доступа к общим данным. Кроме того, не все алгоритмы допускают эффективное распараллеливание.

В большинстве случаев приходится комбинировать вышеприведенные методы повышения производительности.

Данный курс в основном посвящен классической теории быстрых алгоритмов [20, 16, 6]. При этом дополнительно рассматриваются возможности их параллельной реализации. Курс сопровождается набором лабораторных работ, которые позволяют практически ознакомиться с рассматриваемым материалом. Кроме того, слушателям предлагается выполнить курсовую работу. Выполнение курсовой работы предполагает

использование дополнительной литературы, ссылки на которую приведены в тексте задания.

Данный курс был разработан при поддержке компании Intel. Автор выражает благодарность А. Исакову, А. Дедовой и М. Димашовой за многочисленные конструктивные замечания.

Глава 2

Архитектура вычислительных систем

В данном разделе рассматриваются архитектура и принципы работы основных компонентов современных ЭВМ с точки зрения разработчика вычислительного программного обеспечения. Содержание данного раздела ни в коей мере не претендует на полноту. Приведенные здесь сведения используются в дальнейшем при исследовании практической эффективности различных алгоритмов.

2.1 Основные компоненты ЭВМ

2.1.1 Архитектура процессора

Понятие архитектуры вычислительного процессора включает в себя набор (систему) команд и компонентов процессора, доступных для использования программами, выполняемыми на данном процессоре. Наиболее важным компонентом процессора, используемым программами, является набор регистров общего назначения, в которых сохраняются промежуточные результаты вычислений. Под микроархитектурой процессора понимается совокупность технических приемов, использованных разработчиком процессора для достижения заданных требований к производительности и функциональности процессора. В рамках одной архитектуры могут использоваться различные микроархитектурные реализации. Система команд практически всех процессоров включает в себя следующие группы команд:

1. Арифметические команды, включающие инструкции сложения, вычитания и побитовых операций (И, ИЛИ, сдвиг и т.п.). Некоторые процессоры могут не иметь команд умножения и деления.

2. Команды сравнения, условного и безусловного перехода, вызова подпрограмм и возврата управления.

3. Команды пересылки данных из памяти в регистры общего назначения и обратно.

4. Системные команды, используемые операционной системой для реализации функций управления устройствами, многозадачности и т.п.

Все команды задаются кодом операции и операндами. В инструкции может быть задано непосредственно значение операнда, указан номер регистра, содержащего операнд, указан адрес ячейки оперативной памяти, в которой находится операнд, или задано выражение, с помощью которого может быть вычислен этот адрес.

Выполнение каждой команды включает в себя следующие основные этапы:

1. Выборка команды из ячейки памяти, задаваемой счетчиком инструкций. Достаточно часто оказывается, что различные инструкции занимают различное количество байт. В связи с этим устройству выборки приходится динамически определять количество выбираемых байт. Выборка чрезмерно длинных инструкций может занять существенное время, поэтому при написании или генерации машинного кода целесообразно избегать их использования. После выборки инструкции счетчик команд увеличивается на длину команды. Команды условного и безусловного перехода, вызова подпрограмм и возврата из них, а также системные команды могут модифицировать значение счетчика команд.

2. Декодирование выбранной команды, т.е. генерирация последовательности управляющих сигналов для отдельных узлов процессора, ответственных за выполнение операции, задаваемой командой. В некоторых случаях инструкция преобразуется в последовательность простейших микрокоманд, реализующих выбранную команду. Эти микрооперации обрабатываются далее аналогичным образом. Различные команды используют различные наборы узлов процессора и транслируются в последовательности микрокоманд различной длины, что может сказываться на времени их выполнения.

3. Выполнение декодированной команды (или микрокоманды) состоит в коммутации соответствующих узлов процессора и подаче на них пускового сигнала.

4. Выгрузка результатов операции в некоторое запоминающее устройство (регистр, ячейку памяти, буфер устройства и т.п).

Каждый из этих этапов может разбиваться на несколько подшагов. В связи с этим каждая команда характеризуется задержкой выполнения (latency), равной числу тактов процессора, требуемых для выполнения всех микрокоманд, составляющих данную инструкцию, и временем выполнения (throughput), равным минимальному числу тактов, которое должно пройти, прежде чем процессор будет готов повторно запустить на выполнение эту же команду. Время выполнения команд, как правило, меньше задержки их выполнения. Большинство современных процессоров использует конвейерный принцип обработки команд, т.е., например, одновременно с декодированием одной инструкции может осуществляться выборка другой инструкции. Благодаря этому в идеальном случае производительность процессора определяется временем выполнения команд. Однако в реальных программах могут встречаться инструкции

условных переходов. Они вызывают простои конвейера, т.к. до момента выполнения собственно инструкции перехода неизвестно, какие инструкции должны выполняться далее. После того, как инструкция перехода будет выполнена, пройдет некоторое количество тактов прежде чем будут подготовлены к исполнению новые инструкции. В связи с этим многие современные процессоры используют предсказание ветвлений на основе собранных статистических данных о ранее выполненных инструкциях условного перехода. Однако предсказание ветвлений не всегда бывает успешным. Поэтому целесообразно минимизировать число условных переходов в часто выполняемых фрагментах программы.

Простои конвейера могут происходить как при обработке инструкций с большим временем выполнения, так и в том случае, когда очередная команда требует обращения к внешнему медленному устройству, например оперативной памяти. Для снижения потерь некоторые процессоры имеют возможность переупорядочивать команды. Это возможно в том случае, когда эти инструкции не имеют зависимостей по данным. Например, последовательность операций $A = B[i] + C; D = E + C$ может быть преобразована в $D = E + C; A = B[i] + C$, что позволит избежать простоя исполнительного устройства процессора во время загрузки из памяти значения $B[i]$, которое может производиться одновременно с вычислением $D = E+C$. Зависимости по данным делятся на истинные (например, $A = B + C; D = A * E$) и ложные ($A = B + C; B = C + D$). Для борьбы с последними процессор может динамически переименовывать регистры, используемые в обрабатываемых командах. Действительно, последний пример может быть преобразован к виду $A = B + C; E = C + D$, что дает возможность исполнять эту последовательность команд в любом порядке. В дальнейшем все ссылки на регистр B также должны быть заменены на регистр E. На микроархитектурном уровне это реализуется с помощью набора (банка) регистров, на которые динамически отображаются архитектурные регистры.

Если система команд процессора допускает множество различных схем адресаций, содержит составные инструкции и в целом является сложной и запутанной (Complicated instruction set computer, CISC), задача динамического переупорядочения команд и переименования регистров становится весьма нетривиальной, что приводит к существенному снижению производительности. Примером CISC-процессоров является семейство Intel x86. На практике многие “сложные” команды используются достаточно редко. В связи с этим получила широкое распространение концепция RISC-процессоров (reduced instruction set computer), в которых система команд сведена к абсолютному минимуму (Sun SPARC, PowerPC, PA-RISC). Это позволяет увеличить количество доступных регистров и упростить обработку команд. Некоторые CISC-процессоры содержат RISC-ядро, осуществляя динамическое преобразование потока команд.

Некоторые процессоры допускают параллельное исполнение команд, не имеющих зависимостей по данным (параллелизм на уровне машинных команд). Суперскалярные процессоры (например, Intel Pentium Pro+, AMD Athlon) не предполагают, что программа в терминах машинных команд содержит какие-либо указания о возможном параллелизме. Задача обнаружения параллелизма полностью возлагается на оборудование процессора, что приводит к его резкому усложнению. Альтернативой

этому подходу является концепция VLIW (Very large Instruction Word), состоящая в том, что каждая машинная команда задает несколько независимых операций. Задача выявления параллелизма в данном случае возлагается на компилятор, т.е. реализуется однократно на программном уровне.

2.1.2 Оперативная память

Главным требованием к оперативной памяти является обеспечение малого времени доступа к ней. Это является достаточно сложной технической задачей, поэтому реализуется иерархическая структура памяти, в которой быстродействие уменьшается с ростом объема подсистемы памяти:

1. Регистровая память. Регистры располагаются непосредственно на процессоре, могут быть подключены напрямую к его устройствам обработки, что дает возможность использовать их в качестве операндов команд.

2. Кеш-память. Кеш-память располагается на процессоре или в непосредственной близости от него, и осуществляет хранение копий отдельных недавно использованных участков оперативной памяти. Кеш-память может иметь многоуровневую структуру.

3. Оперативное запоминающее устройство. ОЗУ может рассматриваться как кеш-память для часто используемых участков виртуальной памяти системы.

4. Виртуальная память. Виртуальная память реализуется средствами операционной системы во взаимодействии с центральным процессором. С помощью данного механизма приложениям предоставляется практически неограниченный объем памяти, при этом наиболее часто используемые фрагменты (страницы) располагаются в ОЗУ, а редко используемые — выгружаются на некоторое внешнее устройство хранения данных, обычно жесткий диск. Большинство процессоров поддерживают аппаратную трансляцию адресов виртуальной памяти в адреса физической памяти. Виртуальная память позволяет также организовать безопасное выполнение нескольких процессов на одной ЭВМ.

5. Внешняя память. В некоторых случаях приходится прибегать к распределенной обработке данных, при которой может потребоваться обращение к данным физически расположенным на другой ЭВМ. В этом случае обращение к различным участкам памяти производится различными методами (Non-uniform memory access, NUMA).

Эффективная реализация обменов между различными типами оперативной памяти базируется на следующих основных правилах: обращаться к медленной памяти как можно реже, переносить за обмен как можно больше и при каждом переносе данных в более быструю память обрабатывать их как можно дольше. Эти правила предполагают, что выполняется свойство локальности вычислений, состоящее в том, что

близкие команды в потоке инструкций используют данные, расположенные в близких ячейках виртуальной памяти. Нарушение этого свойства приводит к резкому снижению производительности ЭВМ.

2.2 Параллельные вычисления

2.2.1 Классификация параллельных систем

В данном разделе рассмотрены несколько подходов к классификации вычислительных систем. Необходимо учитывать, что в большинстве современных систем могут присутствовать признаки сразу нескольких классов.

Классификация Флинна

Наиболее известной классификацией вычислительных систем является классификация Флинна [7, 1]:

1. SISD (single instruction stream/single data stream) —одиночный поток команд и одиночный поток данных. К этому классу относятся последовательные компьютерные системы, которые имеют один центральный процессор, способный обрабатывать только один поток последовательно исполняемых инструкций. В настоящее время практически все высокопроизводительные системы имеют более одного центрального процессора, однако каждый из них выполняет несвязанные потоки инструкций, что делает такие системы комплексами SISD-систем, действующих на разных пространствах данных. Для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка. Примерами компьютеров с архитектурой SISD могут служить процессоры x86, не поддерживающие набор инструкций MMX.

2. MISD (multiple instruction stream / single data stream) –– множественный поток команд и одиночный поток данных. Теоретически в этом типе машин множество инструкций должно выполняться над единственным потоком данных. В настоящее время отсутствует единое мнение о том, какие архитектуры процессоров могут быть отнесены к этому классу. Среди возможных вариантов рассматриваются суперскалярные процессоры, сигнальные процессоры, системы с аппаратной избыточностью и т.п.

3. SIMD (single instruction stream / multiple data stream) —- одиночный поток команд и множественный поток данных. Эти системы обычно имеют некоторое количество процессоров, которые могут выполнять одну и ту же инструкцию относительно разных данных в жесткой конфигурации. Единственная инструкция параллельно выполняется над многими элементами данных. Примерами SIMD-машин являются

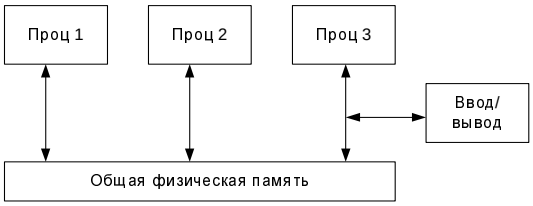


Рис. 2.1: SMP-система

системы CPP DAP, Gamma II и Quadrics Apemille. Другим подклассом SIMD-систем являются векторные компьютеры. Векторные компьютеры манипулируют массивами сходных данных подобно тому, как скалярные машины обрабатывают отдельные элементы таких массивов. Это делается за счет использования специально сконструированных векторных центральных процессоров. Когда данные обрабатываются посредством векторных модулей, результаты могут быть выданы на один, два или три такта частотогенератора (такт частотогенератора является основным временным параметром системы). При работе в векторном режиме векторные процессоры обрабатывают данные практически параллельно, что делает их в несколько раз более быстрыми, чем при работе в скалярном режиме. Примерами векторных процессоров являются Intel-совместимые процессоры, поддерживающие наборы инструкций MMX, SSE[2,3].

4. MIMD (multiple instruction stream / multiple data stream) —— множественный поток команд и множественный поток данных. Эти машины параллельно выполняют несколько потоков инструкций над различными потоками данных. В отличие от упомянутых выше многопроцессорных SISD-машин, команды и данные связаны, потому что они представляют различные части одной и той же задачи. Например, MIMD-системы могут параллельно выполнять множество подзадач с целью сокращения времени выполнения основной задачи. Большое разнообразие попадающих в данный класс систем делает классификацию Флинна не полностью адекватной.

Классификация по способу организации памяти

Параллельные компьютеры могут быть дополнительно классифицированы по способу организации памяти. SMP (symmetric multiprocessing) —— симметричная многопроцессорная архитектура. Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами (см. рис. 2.1). Память служит, в частности, для передачи сообщений между процессорами, при этом все вычислительные устройства при обращении к ней имеют равные права и одну и ту же адресацию для всех ячеек памяти. Поэтому SMP-архитектура называется

симметричной. Последнее обстоятельство позволяет очень эффективно обмениваться данными с другими вычислительными устройствами. Наиболее известными SMP-системами являются SMP-серверы и рабочие станции на базе процессоров Intel, AMD, IBM, HP и др. В настоящее время стали доступны многоядерные SMP-системы, содержащие несколько вычислительных ядер в одной микросхеме.

Основные преимущества SMP-систем:

- простота и универсальность для программирования. Архитектура SMP не накладывает ограничений на модель программирования, используемую при создании приложения: обычно используется модель параллельных ветвей, когда все процессоры работают независимо друг от друга. Однако можно реализовать и модели, использующие межпроцессорный обмен. Использование общей памяти увеличивает скорость такого обмена, пользователь также имеет доступ сразу ко всему объему памяти. Для SMP-систем существуют довольно эффективные средства автоматического распараллеливания;

- простота эксплуатации. Как правило, SMP-системы используют систему кондиционирования, основанную на воздушном охлаждении, что облегчает их техническое обслуживание;

- относительно невысокая цена.

Недостатки:

- системы с общей памятью плохо масштабируются. Этот существенный недостаток SMP-систем не позволяет считать их по-настоящему перспективными. Причиной плохой масштабируемости является то, что в данный момент шина способна обрабатывать только одну транзакцию, вследствие чего возникают проблемы разрешения конфликтов при одновременном обращении нескольких процессоров к одним и тем же областям общей физической памяти. Вычислительные элементы начинают друг другу мешать. Когда произойдет такой конфликт, зависит от скорости связи и от количества вычислительных элементов. В настоящее время конфликты могут происходить при наличии 8-24 процессоров. Кроме того, системная шина имеет ограниченную (хоть и высокую) пропускную способность (ПС) и ограниченное число слотов. Все это очевидно препятствует увеличению производительности при увеличении числа процессоров и числа подключаемых пользователей. В реальных системах можно задействовать не более 32 процессоров.

Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры. При работе с SMP-системами используют так называемую парадигму программирования с разделяемой памятью (shared memory paradigm).

MPP (massive parallel processing) – массивно-параллельная архитектура. Главная особенность такой архитектуры состоит в том, что память физически разделена.

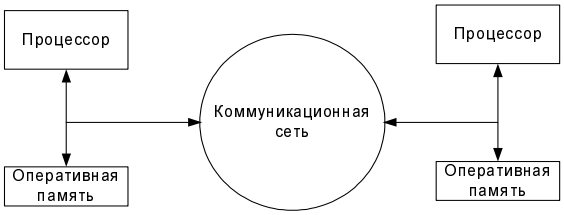


Рис. 2.2: MPP-система

В этом случае система строится из отдельных модулей, содержащих процессор[1], локальный банк операционной памяти (ОП), коммуникационные процессоры (рутеры) или сетевые адаптеры, иногда – жесткие диски и/или другие устройства ввода/вывода. По сути, такие модули представляют собой полнофункциональные компьютеры (см. рис. 2.2). Доступ к банку ОП из данного модуля имеют только процессоры (ЦП) из этого же модуля. Модули соединяются специальными коммуникационными каналами. Пользователь может определить логический номер процессора, к которому он подключен, и организовать обмен сообщениями с другими процессорами. Используются два варианта организации работы операционной системы (ОС) на машинах MPP-архитектуры. В одном полноценная ОС работает только на управляющей машине (front-end), на каждом отдельном модуле функционирует сильно урезанный вариант ОС, обеспечивающий работу только расположенной в нем ветви параллельного приложения. Во втором варианте на каждом модуле работает полноценная UNIX-подобная ОС, устанавливаемая отдельно.

Главным преимуществом систем с раздельной памятью является хорошая масштабируемость: в отличие от SMP-систем, в машинах с раздельной памятью каждый процессор имеет доступ только к своей локальной памяти, в связи с чем не возникает необходимости в потактовой синхронизации процессоров. Практически все рекорды по производительности на сегодня устанавливаются на машинах именно такой архитектуры, состоящих из нескольких тысяч процессоров (ASCI Red, ASCI Blue Pacific).

Недостатки:

- отсутствие общей памяти заметно снижает скорость межпроцессорного обмена, поскольку нет общей среды для хранения данных, предназначенных для обмена между процессорами. Требуется специальная техника программирования для реализации обмена сообщениями между процессорами;
- каждый процессор может использовать только ограниченный объем локального банка памяти;

[1]В некоторых случаях модули представляют собой SMP-системы, оснащенные несколькими процессорами. Но в этом случае уже нельзя говорить о чистой MPP-системе.

- вследствие указанных архитектурных недостатков требуются значительные усилия для того, чтобы максимально использовать системные ресурсы. Именно этим определяется высокая цена программного обеспечения для массивно-параллельных систем с раздельной памятью

2.2.2 Параллельные алгоритмы

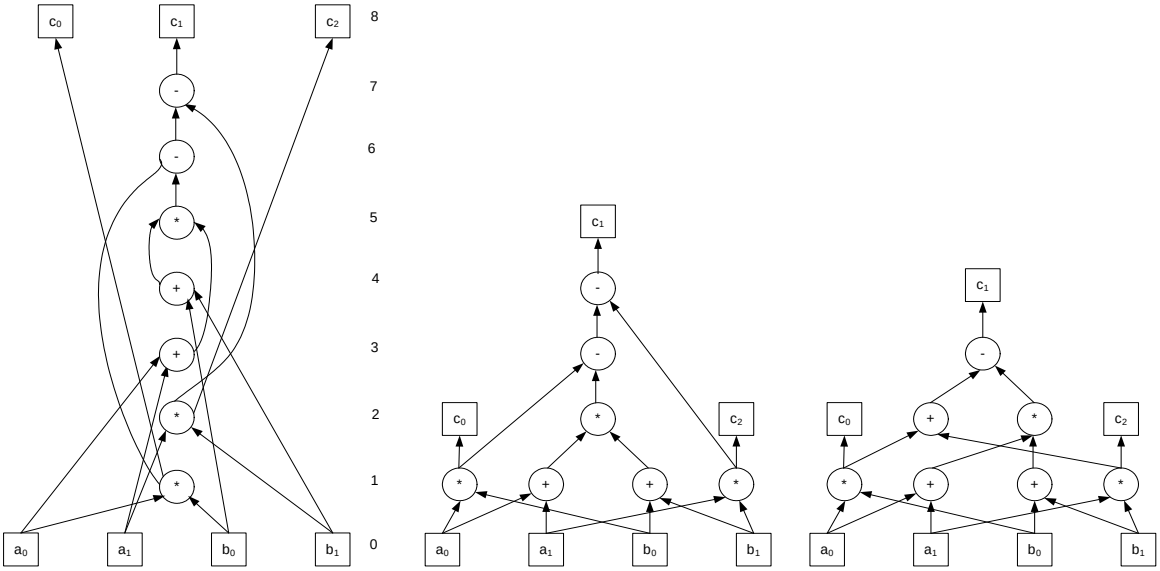
Под параллельными алгоритмами понимаются алгоритмы, некоторая часть которых может выполняться одновременно на различных вычислительных устройствах. Это определение включает в себя все возможные типы параллелизма, описанные в разделе 2.2.1. Всякий алгоритм, не содержащий условных операторов, может быть представлен в виде последовательности операций. Сопоставим каждой операции узел в графе. Если результат операции A используется операцией B, проведем дугу от A к B. В полученном ориентированном ациклическом графе вершины без входящих узлов соответствуют исходным данным, а вершины без исходящих узлов — результатам вычислений. Реальные программы могут содержать условные операторы или иным образом изменять свое поведение в зависимости от своих параметров, т.е. граф алгоритма может быть недетерминированным. Но для большинства вычислительных алгоритмов условные операторы могут быть погружены в некоторые макрооперации, а графы, получаемые для различных значений параметров, имеют сходную структуру. Поэтому в дальнейшем будут рассматриваться детерминированные графы.

Разметим узлы графа числами $v(A_i)$ таким образом, что если существует дуга из A_i в A_j, то $v(A_i) < v(A_j)$. Размеченный таким образом граф носит название *строгой параллельной формой* алгоритма. Она может быть построена следующим образом:

1. Выбрать произвольное подмножество вершин, не имеющих входящих дуг, пометить их индексом 0.
2. Удалить помеченные вершины.
3. Выбрать произвольное подмножество вершин, не имеющих входящих дуг, пометить их индексом 1.
4. Удалить помеченные вершины.
5. Действовать аналогично до исчерпания всех вершин графа.

Строгая параллельная форма не может содержать дуг между узлами с одинаковой меткой $v(A_i)$. Существует строгая параллельная форма, в которой максимальная из длин путей, оканчивающихся в вершине с индексом k, равна k. Среди таких параллельных форм существует такая, для которой все входные вершины находятся в группе с одним индексом, равным 0. Данная параллельная форма носит название *канонической параллельной формы*. Каноническая параллельная форма единственна.

Пример 2.1. Рассмотрим следующий алгоритм (Карацубы) вычисления произведения многочленов первой степени $c_0 + c_1x + c_2x^2 = (a_0 + a_1x)(b_0 + b_1x)$:



(a) Строгая неканоническая форма (b) Каноническая параллельная форма (c) КПФ модифицированного алгоритма

Рис. 2.3: Параллельные формы алгоритма Карацубы

1. $c_0 = a_0 b_0$
2. $c_2 = a_1 b_1$
3. $t_1 = a_0 + a_1$
4. $t_2 = b_0 + b_1$
5. $t_3 = t_1 t_2$
6. $t_4 = t_3 - c_0$
7. $c_1 = t_4 - c_2$

Заметим, что последние два шага могут быть заменены на $t_4 = c_2 + c_0$ и $c_1 = t_3 - t_4$. На рисунке 2.3 представлены несколько параллельных форм этого алгоритма.

Ярусом параллельной формы называется группа вершин с одинаковыми номерами. Предполагая, что все операции выполняются за одно и то же время и в наличии имеется достаточно большое число процессоров, получим, что номер яруса указывает момент окончания соответствующих операций. Максимальное число узлов в ярусе называется шириной параллельной формы. Число ярусов называется высотой параллельной формы. Параллельная форма минимальной высоты называется максимальной. Данное название связано с тем, что уменьшение высоты формы достигается, как правило, за счет увеличения ее ширины. Каждая параллельная форма соответствует определенной реализации заданного алгоритма. Например, параллельная форма, представленная на

рисунке 2.3(a), соответствует реализации алгоритма Карацубы на последовательном компьютере, в то время как форма на рисунке 2.3(b) соответствует реализации на ЭВМ с четырьмя независимыми исполнительными устройствами.

Удобной теоретической моделью для анализа параллельных алгоритмов является концепция *неограниченного параллелизма*. В рамках данной концепции предполагается, что имеется бесконечно большое число процессоров, способных выполнять любые операции, требуемые алгоритмом Процессоры работают синхронно и способны обмениваться информацией мгновенно и без конфликтов. Ясно, что в этом случае минимальное время выполнения достигается при использовании алгоритма с минимальной высотой.

Теорема 2.1. *Пусть функция существенно зависит от n переменных и представлена как суперпозиция конечного числа операций, имеющих не более p аргументов. Высота алгоритма вычисления этой функции с помощью тех же операций не может быть меньше* $\log_p n$

Доказательство. Пусть на нулевом ярусе параллельной формы расположены вершины, соответствующие входным переменным алгоритма. На верхнем ярусе должна находиться единственная вершина, соответствующая выходной переменной. В каждую вершину графа может входить не более p дуг. Если параллельная форма содержит s ярусов, то нижний из них не может иметь более p^s вершин. С другой стороны, известно, что он содержит ровно n вершин. Таким образом, $n \leq p^s$ и $s \geq \log_p n$. □

Реальной производительностью вычислительного устройства или их системы называется количество операций, реально выполненных в среднем за единицу времени при решении некоторой фиксированной задачи. *Пиковой производительностью* называется максимальное число операций, которое может быть выполнено тем же устройством/системой за единицу времени при отсутствии связей между функциональными узлами. Стоимостью работы называется время *последовательной* реализации всех рассматриваемых операций на заданном устройстве. Загруженностью устройства на данном отрезке времени называется отношение стоимости реально выполненной работы к максимально возможной стоимости. В параллельных вычислительных системах часто встречается ситуация, при которой в силу структуры используемого алгоритма невозможно загрузить полезной работой некоторые исполнительные устройства. Если система состоит из l устройств, имеющих пиковые производительности $\pi_1, \ldots, \pi_l$, работающих с загруженностями $p_1, \ldots, p_l$, то реальная производительность r системы равна

$$r = \sum_{i=1}^{l} p_i \pi_i \tag{2.1}$$

Целесообразность применения параллельных вычислительных систем для решения конкретной задачи определяется уменьшением времени вычислений, достигаемым за счет

их использования. Достигнутое ускорение может быть найдено как

$$R = \frac{\sum_{i=1}^{l} p_i \pi_i}{\max_i \pi_i} \tag{2.2}$$

Таким образом, при использовании l вычислительных устройств максимальное возможное ускорение равно l. Оно достигается при полной загрузке всех устройств, что требует использования соответствующего алгоритма.

Теорема 2.2. *Пусть высота параллельной формы равна s и всего в алгоритме выполняется N операций. Максимально возможное ускорение вычислений при их параллельной реализации по сравнению с последовательной равно N/s*

Доказательство. Пусть система состоит из l устройств пиковой производительности π. Предположим, что за время T реализации алгоритма на i-м устройстве выполняется N_i операций. Следовательно, загруженность этого устройства равна $\frac{N_i}{\pi T}$. Таким образом, ускорение системы равно $R = \frac{\sum_{i=1}^{l} \frac{N_i}{\pi T}\pi}{\pi} = \frac{N}{\pi T}$. Время реализации одного яруса пареллельной формы равно $1/\pi$. Поэтому время реализации алгоритма удовлетворяет $T \geq s/\pi$, откуда и вытекает утверждение теоремы. □

Минимально возможное число l устройств, необходимое для достижения предельного ускорения, равно ширине алгоритма. В реальности редко удается обеспечить одинаковое время выполнения всех операций параллельной формы, что снижает производительность системы.

Предположим, что по каким-либо причинам n операций из N имеющихся в алгоритме приходится выполнять последовательно, т.е. соответствующие ярусы параллельной формы имеют ширину 1. Отношение $\beta = \frac{n}{N}$ называется долей последовательных вычислений.

Теорема 2.3 (Закон Амдала)**.** *Пусть система состоит из l одинаковых универсальных вычислительных устройств. Предположим, что при выполнении параллельной части алгоритма все устройства загружены полностью. Тогда максимально возможное ускорение равно*

$$R = \frac{l}{\beta l + (1 - \beta)} \tag{2.3}$$

Доказательство. Пусть π — пиковая производительность отдельного вычислительного устройства. Если всего выполняется N операций, βN из них выполняются последовательно, например, на первом устройстве, и $(1 - \beta)N$ операций выполняются параллельно на l устройствах по $(1 - \beta)N/l$ на каждом. Всего алгоритм реализуется за время $T_1 = \frac{\beta N + (1-\beta)N/l}{\pi}$. На параллельной части алгоритма работают как первое, так и остальные устройства в течение $T_i = \frac{(1-\beta)N/l}{\pi}, i = 2..l$. Таким образом, $\rho_1 = 1$ и $\rho_i = \frac{(1-\beta)N/l}{\beta N + (1-\beta)N/l}, i \geq 2$. Согласно (2.2), получим $R = 1 + \sum_{i=2}^{l} \frac{1-\beta}{\beta l + (1-\beta)} = \frac{(l-1)(1-\beta) + \beta l + (1-\beta)}{\beta l + (1-\beta)} = \frac{l}{l\beta + (1-\beta)}$ □

Приведенные теоремы позволяют оценить максимально возможный выигрыш, достижимый за счет распараллеливания вычислений. Известно достаточно большое число алгоритмов, построенных согласно концепции неограниченного параллелизма. Однако большинство таких алгоритмов характеризуется численной неустойчивостью, огромными требованиями к числу процессоров и сложной схемой обмена данными, что делает эти алгоритмы практически бесполезными. Тем не менее, полученные результаты могут использоваться в качестве ориентира при разработке практических параллельных алгоритмов.

В связи с этим возникает необходимость поиска *внутреннего параллелизма* в существующих вычислительных алгоритмах. Если построена параллельная форма некоторого вычислительного алгоритма и ее ярусы имеют достаточно большую ширину, существует принципиальная возможность его параллельной реализации. Поиск хорошей параллельной формы может потребовать некоторых алгебраических преобразований, как это было сделано при построении параллельной формы на рисунке 2.3(c), где за счет использования свойств операции сложения удалось уменьшить число ярусов.

Многие вычислительные алгоритмы компьютерной алгебры строятся по принципу "разделяй и властвуй", т.е. исходная задача разбивается на p аналогичных подзадач меньшей размерности, которые решаются независимо, после чего результаты их решений объединяются с помощью какой-либо сравнительно простой операции. Разбиение задачи на подзадачи может быть продолжено рекурсивно. Таким образом, получается дерево разбиения, которое может рассматриваться как граф алгоритма. На k-ом уровне разбиения ширина графа равна p^k. Полная загрузка всех процессоров будет обеспечена[2], если выполняется условие $l|p^k$.

2.3 Реализация вычислительных алгоритмов

В данном разделе рассматриваются основные приемы, которые могут использоваться для повышения эффективности использования имеющихся аппаратных средств.

2.3.1 Влияние характеристик процессора на скорость вычислений

Основными характеристиками, влияющими на скорость выполнения вычислительного алгоритма на заданной ЭВМ, являются:

1. Количество различных операций преобразования данных, доступа к памяти и управления состоянием процессора. Различные операции требуют разного числа тактов для своего исполнения, способны одновременно выполнять операции над разными объемами данных и требуют задействования различных узлов процессора.

2. Последовательность адресов в оперативной памяти, к которым осуществляется доступ при выполнении программы. Наличие кеш-памяти позволяет существенно

[2]Здесь не учитываются операции, связанные с объединением результатов решения различных подзадач. Как правило, они имеют достаточно малую сложность.

снизить задержку при обращении к некоторым блокам данных. Если при исполнении программы систематически производится обращение к данным, отсутствующим в кеше, производительность процессора оказывается ограниченной производительностью подсистемы памяти.

3. Количество используемых регистров процессора. Увеличение их числа позволяет снизить число обращений к оперативной памяти.

4. Объем данных, обрабатываемых при выполнении одной инструкции. При обработке больших массивов данных часто оказывается, что одна и та же операция последовательно применяется ко всем элементам массива.

5. Количество инструкций, запускаемых одновременно на исполнение. Многие процессоры способны одновременно исполнять несколько инструкций, если между ними отсутствуют зависимости по данным.

6. Предсказуемость условных переходов. Обработка команд в процессоре производится в несколько этапов по конвейерному принципу. При плохой предсказуемости условных переходов возможны простои отдельных элементов конвейера.

Проиллюстрируем эти принципы с помощью нескольких простейших программ. В большинстве случаев нижеприведенные программы не имеют никакой полезной функциональности, а служат лишь для исследования характеристик ЭВМ. Во всех случаях в данном разделе указывается приведенное время выполнения соответствующей функции, т.е. среднее время обработки одного элемента входных данных.

Сложность различных арифметических операций

Программа ArithmeticComplexity из прилагаемого комплекта позволяет оценить время выполнения различных арифметических операций. Программа выполняет заданное число раз различные операции над элементами массивов целых чисел и чисел с плавающей точкой с двойной точностью. Результаты замеров времени выполнения различных инструкций на различных процессорах приведены в таблице 2.1. Все результаты были получены с помощью одного и того же исполнимого модуля, скомпилированного с помощью Microsoft Visual C++ 8.0.

Точное число тактов, требуемых на предварительную обработку и выполнение различных инструкций, можно узнать из технической документации для соответствующего процессора. Каждая инструкция характеризуется временем (latency) и задержкой (throughput) обработки. Время обработки инструкции равно числу тактов, требуемых для выполнения всех необходимых микроопераций, задаваемых инструкцией. Задержка обработки равна числу тактов, по истечении которых исполнительные устройства процессора готовы к обработке следующей такой же инструкции. В таблице 2.2 приведены характеристики некоторых инструкций, использованных в данном исследовании, для процессоров Intel Pentium 4+ [15]. Необходимо учитывать, что эти

Таблица 2.1: Время выполнения арифметических операций, нс

Тип аргумента	int				double		
Операция	^	+	*	/	+	*	/
Pentium 4 2,5 ГГц	0,78	0,86	5,6	19	2	2,8	15
CoreDuo 1,8 ГГц	1,1	1,1	2,2	4,2	1,7	2,7	18
Pentium M 1,8 ГГц	1,1	1,1	2,2	8,7	1,7	2,7	17
Athlon 64X2 2,5 ГГц	1,4	1,4	1,6	17	1,8	1,8	8,3

Таблица 2.2: Число тактов, необходимых для исполнения инструкций на процессорах Intel P4+

Инструкция	XOR	ADD	MUL	DIV	FADD	FMUL	FDIV (double)
Задержка обработки	0,5–1	0,5–1	3–18	22–70	5–6	7–8	38–40
Время обработки	0,5	0,5	1–5	23–30	1	2	38–40

величины могут существенно отличаться у различных процессоров, а также зависеть от местоположения обрабатываемых данных (в регистрах, кеше или в памяти), а иногда и собственно от данных. Кроме того, некоторые инструкции могут выполняться параллельно.

Однако полученные результаты позволяют сделать некоторые выводы относительно возможных направлений оптимизации вычислительных алгоритмов:

1. При обработке целочисленных данных целесообразно минимизировать число умножений. При этом допустим некоторое увеличение числа сложений.

2. При обработке данных с плавающей точкой необходимо одновременно уменьшать как число умножений, так и число сложений.

3. Необходимо минимизировать число операций деления.

Предсказание ветвлений

Обработка команд в современных процессорах осуществляется поэтапно на основе конвейерного принципа. При обнаружении команды условного ветвления может оказаться, что операция, определяющая соответствующее условие, еще не выполнена до конца. В этом случае невозможно определить, какая из ветвей программы будет выполняться дальше. До недавнего времени подобные ситуации приводили к тому, что обработка дальнейших команд приостанавливалась до момента вычисления условного перехода. Ввиду того, что длина конвейера может быть весьма велика, это приводило к значительным потерям производительности. Данная проблема может быть решена путем сбора статистической информации о том, исполнение какой ветви условного перехода более вероятно. Данная информация может быть получена, например, во время предыдущих проходов данного участка кода или из “подсказки”, вставленной компилятором в исполнимый код. В этом случае процессор может продолжить обработку

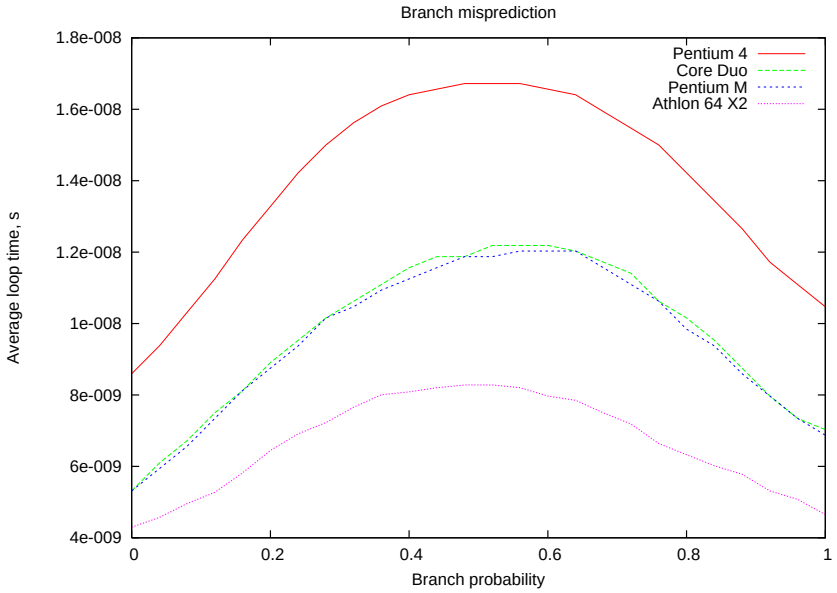


Рис. 2.4: Влияние вероятности перехода на время выполнения

команд предсказанной ветви программы. В тот момент, когда становится доступным значение условного выражения, проверяется правильность предсказания. Если оно оказалось неверным, результаты обработки ошибочно выбранных команд уничтожаются, и начинается обработка альтернативной ветви программы. Это приводит к простою исполнительных блоков процессора. Таким образом, наихудшим вариантом является условный переход, обе ветви которого равновероятны.

Программа BranchPrediction иллюстрирует работу механизма предсказания ветвлений. Программа состоит в последовательном вычислении псевдослучайных чисел. В зависимости от значения очередного случайного числа выполняется сложная операция вычисления остатка от деления на то или иное целое число.

```
1  for (unsigned i=0;i<Count;i++)
2      {
3          unsigned OldState=RNGState;
4          RNGState=(RNGState*1664525+1013904223);
5          if (RNGState<BranchThreshold)
6              Sum+=(OldState%12345);
7          else
8              Sum+=(OldState%12346);
9
10     };
```

Рисунок 2.4 иллюстрирует время выполнения одной итерации данного цикла в зависимости от вероятности ветвления. Видно, что для всех рассматриваемых процессоров приближение вероятности условного перехода к $0,5$ приводит к двукратному снижению производительности. Это означает, что каждый неверно предсказанный

переход приводит к увеличению времени выполнения соответствующих инструкций примерно в 4 раза.

В связи с этим целесообразно удалять операции условного перехода из часто выполняемых участков программы. Например, если переход выполняется для четных элементов некоторого массива, целесообразно создать два цикла, отдельно обрабатывающих четные и нечетные элементы.

Кеш-память

Большинство современных процессоров оборудованы кеш-памятью, в которой хранятся копии часто используемые блоки данных из оперативной памяти. Кеш-память организована в виде совокупности кеш-линий, содержащих фиксированное число байт (например, $L = 16$). При обращении к данным, отсутствующим в кеше, одновременно загружается весь L-байтный блок, содержащий затребованные данные. Если оказывается, что затребованный блок данных пересекает границу блока из L байт, загружаются оба блока. Работа кеша основывается на предположении о том, что смежные участки программы обрабатывают смежные блоки данных. Невыполнение этой гипотезы может привести к дезорганизации работы кеш-памяти и резкому падению производительности.

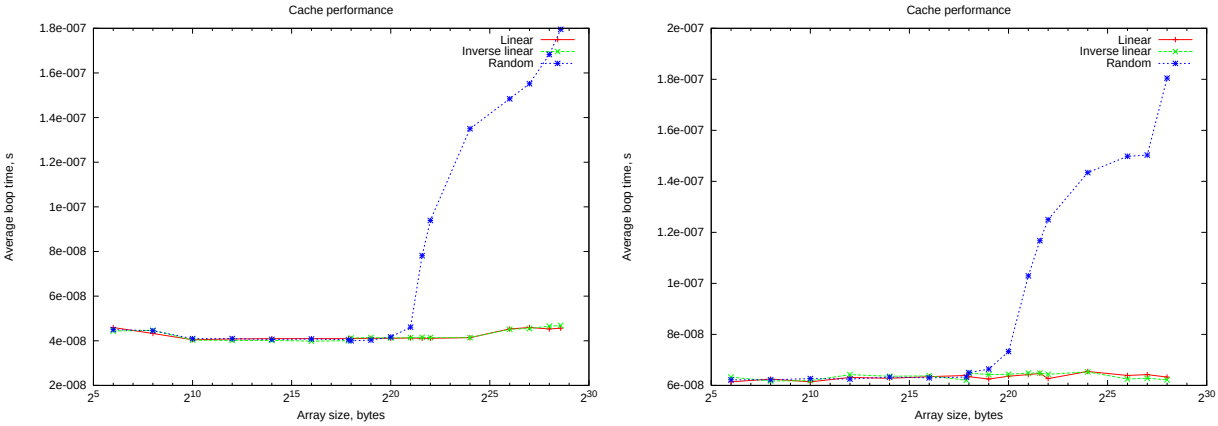
Рассмотрим программу, осуществляющую суммирование элементов целочисленного массива в линейном возрастающем, линейном убывающем и псевдослучайном порядке. Рисунок 2.5 иллюстрирует время выполнения одной итерации цикла суммирования на различных процессорах в зависимости от размера массива. Видно, что скорость линейной обработки массива практически не зависит от его размера. Но если обращение к элементам массива происходит в псевдослучайном порядке, время доступа к памяти катастрофически возрастает. Ввиду того, что в данной программе используется достаточно сложный генератор псевдослучайных чисел, на приведенных графиках не виден эффект от кеша первого уровня.

Таким образом, при обработке больших массивов данных необходимо избегать хаотических обращений к памяти.

Суперскалярная архитектура процессора

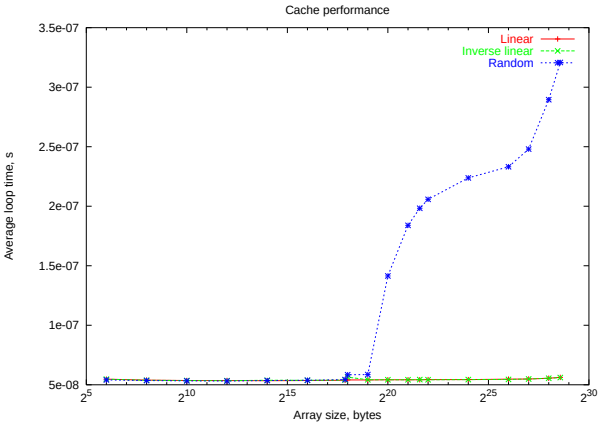
Большинство современных процессоров способны одновременно исполнять несколько команд, если между ними нет зависимостей по данным. Многие компиляторы способны автоматически упорядочивать инструкции для обеспечения их параллельного выполнения. Однако во многих случаях возможности параллельного исполнения инструкций ограничены структурой реализуемого алгоритма. В связи с этим целесообразно учитывать возможность параллельного исполнения инструкций на этапе разработки алгоритма.

Рассмотрим задачу вычисления значения многочлена в некоторой точке. Оптимальным способом решения этой задачи на последовательной машине является

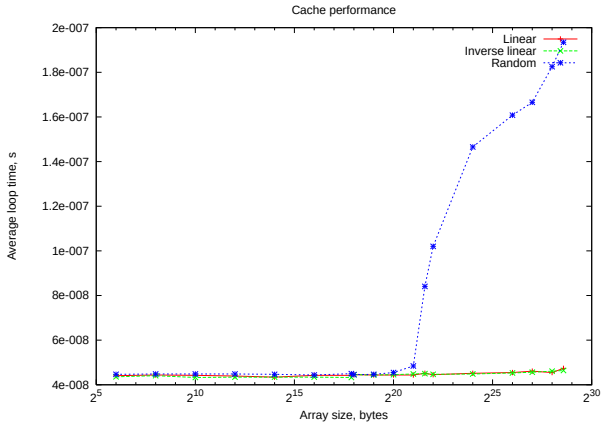


(a) Centrino Duo, 2 МБ кеш

(b) Athlon 64X2, 2*512 КБ кеш



(c) Pentium 4, 512 КБ кеш



(d) Pentium M, 2 МБ кеш

Рис. 2.5: Влияние порядка обращения к памяти на производительность

схема Горнера, т.е.

$$f(x) = f_0 + x(f_1 + x(f_2 + x(f_3 + \ldots + xf_n)\ldots) \tag{2.4}$$

При ее непосредственной реализации на суперскалярном процессоре параллельное исполнение инструкций практически исключается. Преобразуем это выражение следующим образом:

$$f(x) = \sum_{j=0}^{d-1} x^j \underbrace{\sum_{i=0}^{\lfloor n/d \rfloor - 1} f_{di+j} x^{di}}_{A_j^{(0)}} = \sum_{j=0}^{d-1} x^j \left(f_j + y(f_{j+d} + y(f_{j+2d} + \ldots + yf_{j+\lfloor n/d \rfloor d})\ldots)\right), \tag{2.5}$$

где $y = x^d$. Это выражение дает возможность параллельно вычислять значения

$$A_j^{(i)} = yA_j^{(i+1)} + f_{di+j}, j = 0..d-1, i = 0..\lfloor n/d \rfloor, A_j^{(\lfloor n/d \rfloor + 1)} = 0 \tag{2.6}$$

Заметим, что общее число арифметических операций при этом несколько возрастает. Вычисление $A_j^{(i)}$ может быть организовано в виде цикла по j. Многие процессоры способны обеспечить параллельное выполнение инструкций даже в этом случае. Однако организация цикла связана с весьма значительными накладными расходами. Поэтому может быть целесообразно выписать в явном виде операторы вычисления $A_j^{(i)}$ для всех j. Данный прием носит название разворачивания цикла. Например, вышеописанный алгоритм с 3-кратным разворачиванием цикла может быть реализован следующим образом:

```
1  unsigned Horner3(unsigned Degree,unsigned x,const unsigned* pCoeffs)
2  {
3      unsigned Tail=(Degree/3)*3;
4      unsigned Temp0=pCoeffs[Tail];
5      unsigned Temp1=pCoeffs[Tail+1];
6      unsigned Temp2=pCoeffs[Tail+2];
7      unsigned y=x*x*x;
8      for (int i=Tail-3;i>=0;i-=3)
9      {
10         Temp0=Temp0*y+pCoeffs[i];
11         Temp1=Temp1*y+pCoeffs[i+1];
12         Temp2=Temp2*y+pCoeffs[i+2];
13     };
14     return Temp0+x*(Temp1+x*Temp2);
15 };
```

Заметим, что данное преобразование стало возможным только после преобразования исходного алгоритма. На рисунке 2.6 приведены графики зависимости времени выполнения программ, реализующих описанный алгоритм. Кривые, соответствующие программам, в которых было реализовано d-кратное разворачивание цикла, обозначены

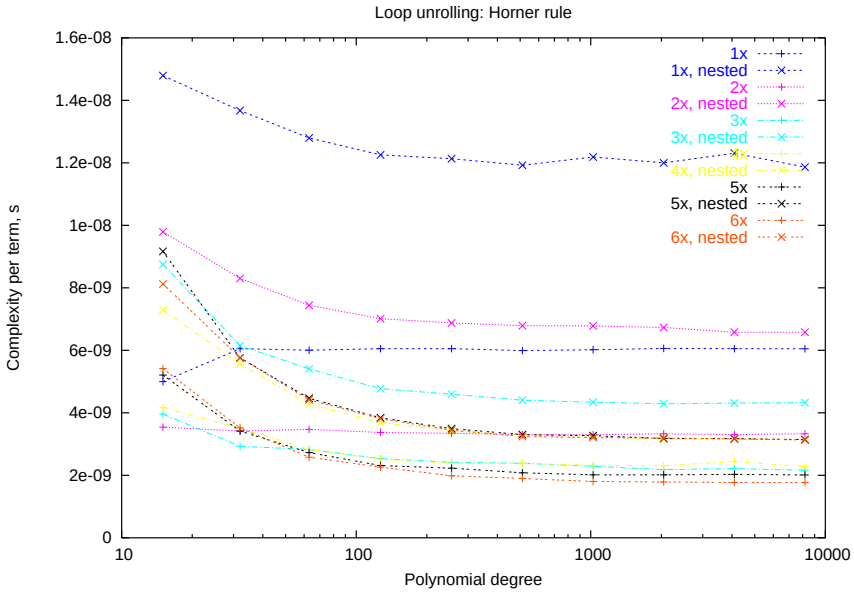


Рис. 2.6: Вычисление значения многочлена на суперскалярном процессоре Pentium 4

на графике как dx. Кривые, соответствующие программам с вложенным циклом по j (см. (2.6)), обозначены как $dx, nested$. Заметим, что кривая $1x$ соответствует непосредственной реализации схемы Горнера, а кривая $1x, nested$ соответствует программе, в которой во внутреннем цикле по i имеется всего одна итерация. Сопоставляя эти графики, можно сделать вывод о величине издержек, связанных с организацией дополнительного цикла. Заметим, что практически максимальная производительность достигается при трехкратной раскрутке цикла. Дальнейшая раскрутка позволяет незначительно увеличить быстродействие за счет снижения накладных затрат при организации цикла. При использовании вложенного цикла увеличения производительности при $d > 3$ не наблюдается. Это согласуется с тем, что процессор, использованный в данном исследовании, способен параллельно исполнять три инструкции.

Необходимо также отметить, что вышеприведенная реализация предполагает, что массив, содержащий коэффициенты многочлена, имеет по крайней мере два нулевых элемента в конце. В реальности это не всегда удается гарантировать. Кроме того, это приводит к увеличению числа операций. В связи с этим при использовании подобных приемов приходится добавлять фрагменты, обеспечивающие обработку “неполных” итераций.

Векторизованная обработка

Большинство современных процессоров поддерживают одновременную обработку нескольких единиц данных одной инструкцией (Single Instruction Multiple Data, SIMD). Некоторые компиляторы позволяют обнаружить в исходном коде фрагменты, которые

могут быть реализованы с помощью SIMD-инструкций. Однако это удается сделать только в простейших случаях. В более сложных случаях программисту приходится в явном виде выписывать последовательность машинных команд, реализующих требуемые действия. Многие компиляторы имеют встроенные функции (intrinsics), которые при компиляции отображаются в соответствующие машинные команды. Альтернативой является использование специализированных библиотек, содержащих оптимизированную реализацию часто используемых математических функций. Их преимуществом является автоматическое определение типа используемого процессора, точный учет его архитектурных особенностей и простота использования. Основным недостатком являются значительные издержки, связанные с вызовом библиотечных функций, а также необходимость включения в дистрибутив программного продукта значительных по объему библиотек, из которых могут реально использоваться только несколько функций. Кроме того, библиотеки, разработанные одним производителем процессоров могут некорректно или недостаточно полно использовать возможности процессоров, выпускаемых другим производителем.

Рассмотрим задачу вычисления скалярного произведения $\sum_{i=0}^{n-1} x_i y_i$ двух векторов шестнадцатибитовых целых чисел и 64-битных чисел с плавающей точкой. Соответствующий алгоритм будет реализован непосредственно на языке C++, с помощью встроенных функций SSE2 и путем обращения к библиотеке Intel Integrated Performance Primitives Library.

Непосредственная реализация функции вычисления скалярного произведения векторов имеет следующий вид:

```
1 //скалярное произведение целочисленных векторов
2 __int64 DotProduct(unsigned Size,const __int16* pA,const __int16* pB)
3 {
4     __int64 Sum=0;
5     for (unsigned i=0;i<Size;i++)
6         Sum+=int(pA[i])*int(pB[i]);
7     return Sum;
8 };
```

Заметим, что данная функция возвращает 64-битное целое число. Реализация на базе SSE2 использует встроенный тип __m128i, который может рассматриваться как объединение одного 128-битного целого числа, двух 64-битных чисел, 4 32-битных, 8 16-битных или 16 8-битных чисел. При обнаружении переменной данного типа компилятор автоматически обеспечивает выравнивание ее по 16-байтной границе (т.е. ее адрес при размещении в памяти делается кратным 16), что обеспечивает наиболее эффективное использование кеш памяти и является необходимым при использовании большинства SIMD-инструкций. Выровнены должны быть и обрабатываемые массивы данных. В таблице 2.3 представлено описание инструкций SSE2, используемых в данном примере. Верхние индексы указывают разрядность соответствующего элемента.

Используя эти инструкции, скалярное произведение можно реализовать следующим образом:

Таблица 2.3: Целочисленные инструкции SSE2, используемые при вычислении скалярного произведения

Инструкция	Встроенная функция	Возвращаемое значение
	_mm_setzero_si128()	0^{128}
MOVDQA	_mm_load_si128(p)	Загрузка 128-битного значения по адресу p
PMADDWD	_mm_madd_epi16(a, b)	$(a_0^{16}b_0^{16}+a_1^{16}b_1^{16}, a_2^{16}b_2^{16}+a_3^{16}b_3^{16}, a_4^{16}b_4^{16}+a_5^{16}b_5^{16}, a_6^{16}b_6^{16}+a_7^{16}b_7^{16}$
PCMPGTD	_mm_cmpgt_epi32(a,b)	$(a_0^{32}>b_0^{32}?\tilde{0}^{32}:0^{32}, a_1^{32}>b_1^{32}?\tilde{0}^{32}:0^{32}, a_2^{32}>b_2^{32}?\tilde{0}^{32}:0^{32}, a_3^{32}>b_3^{32}?\tilde{0}^{32}:0^{32})$
PUNPCKLDQ	_mm_unpacklo_epi32(a,b)	$(a_0^{32}, b_0^{32}, a_1^{32}, b_1^{32})$
PUNPCKHDQ	_mm_unpackhi_epi32(a,b)	$(a_2^{32}, b_2^{32}, a_3^{32}, b_3^{32})$
PADDQ	_mm_add_epi64(a,b)	$(a_0^{64}+b_0^{64}, a_1^{64}+b_1^{64})$
PSRLDQ	_mm_srli_si128(a,b)	Логический сдвиг a вправо на b позиций

$\tilde{0}$ =0xffffffff

```
1  __int64 DotProductSSE2(unsigned Size,const __int16* pA,const __int16* pB)
2  {
3      __m128i A,B,S,PackedProduct;
4      union
5      {
6          __m128i PackedSum;
7          __int64 Sum;
8      };
9      PackedSum=_mm_setzero_si128();//обнуление
10     for (unsigned i=0;i<Size;i+=8)
11     {
12         //загрузка данных
13         A=_mm_load_si128((__m128i*)(pA+i));
14         B=_mm_load_si128((__m128i*)(pB+i));
15         //перемножение 8 пар целых чисел и сложение пар смежных произведений
16         PackedProduct=_mm_madd_epi16(A,B);
17         //PackedProduct содержит 4 32-битных числа
18         //определяем знак каждого из них
19         S=_mm_cmpgt_epi32(_mm_setzero_si128(),PackedProduct);
20         //приписываем знаковые разряды к нулевому и первому словам
21         A=_mm_unpacklo_epi32(PackedProduct,S);
22         //приписываем знаковые разряды ко второму и третьему словам
23         B=_mm_unpackhi_epi32(PackedProduct,S);
24         //сложение по правилам 64-битной арифметики
25         A=_mm_add_epi64(A,B);
26         //накопление двух 64-битных сумм
27         PackedSum=_mm_add_epi64(PackedSum,A);
28     };
29     //сдвиг на 64 бита вправо
30     B=_mm_srli_si128(PackedSum,8);
31     //сложение 64-битных сумм
32     PackedSum=_mm_add_epi64(PackedSum,B);
33     return Sum;};
```

Здесь также предполагается, что обрабатываемые массивы имеют размер, кратный

восьми, или дополнены нулями до размера, кратного восьми. Работа функции начинается с обнуления переменной PackedSum, в которой в дальнейшем будут одновременно накапливаться суммы $\sum_j (a_{8j}b_{8j} + a_{8j+1}b_{8j+1} + a_{8j+4}b_{8j+4} + a_{8j+5}b_{8j+5})$ и $\sum_j (a_{8j+2}b_{8j+2} + a_{8j+3}b_{8j+3} + a_{8j+6}b_{8j+6} + a_{8j+7}b_{8j+7})$. На каждой итерации цикла в переменные A и B загружаются 8 16-битовых целых чисел из соответствующих массивов. Они покомпонентно перемножаются, после чего смежные пары произведений складываются, что дает четыре 32-битовых числа $a_{8j}b_{8j} + a_{8j+1}b_{8j+1}, a_{8j+2}b_{8j+2} + a_{8j+3}b_{8j+3}, a_{8j+4}b_{8j+4} + a_{8j+5}b_{8j+5}, a_{8j+6}b_{8j+6} + a_{8j+7}b_{8j+7}$. Эти числа нельзя непосредственно сложить друг с другом[3], т.к. при этом может произойти переполнение. В связи с этим их необходимо преобразовать в 64-битовое представление. Т.к. они могут быть как положительными, так и отрицательными, необходимо уделить особое внимание расширению их знакового бита. Напомним, что 32-битовое положительное число при преобразовании в 64-битовое представление должно быть дополнено нулями в старших разрядах, в то время как отрицательное число должно быть дополнено единицами. Операция _mm_cmpgt_epi32 осуществляет поэлементное сравнение четырех упакованных 32-разрядных чисел. Т.к. одним из ее аргументов является ноль, для отрицательных компонентов переменной Packed-Product она возвратит 0xffffffff, а для положительных — 0x00000000. С помощью команд _mm_unpacklo_epi32 эти значения дописываются к соответствующим компонентам PackedProduct. Заметим, что теперь для хранения его компонентов требуются две 128-битовых переменных A и B, в которых хранятся 64-битовые значения $(a_{8j}b_{8j}+a_{8j+1}b_{8j+1}, a_{8j+2}b_{8j+2}+a_{8j+3}b_{8j+3})$ и $(a_{8j+4}b_{8j+4}+a_{8j+5}b_{8j+5}, a_{8j+6}b_{8j+6}+a_{8j+7}b_{8j+7})$, соответственно. Эти переменные складываются между собой по правилам 64-битовой арифметики. В результате в переменной A образуется пара значений $(a_{8j}b_{8j}+a_{8j+1}b_{8j+1}+a_{8j+4}b_{8j+4}+a_{8j+5}b_{8j+5}, a_{8j+2}b_{8j+2}+a_{8j+3}b_{8j+3}+a_{8j+6}b_{8j+6}+a_{8j+7}b_{8j+7})$, которая добавляется к накопленной паре сумм PackedSum. После обработки всех элементов массива PackedSum сдвигается вправо на 64 позиции, вследствие чего вторая накопленная сумма оказывается в тех же разрядах, что и первая. Это дает возможность сложить их с помощью одной операции 64-битного сложения. В качестве результата функция возвращает значение переменной Sum, которая была совмещена с переменной PackedSum. Аналогичный алгоритм реализован в функции ippsDotProd_16s64s из библиотеки IPP.

Для вычисления скалярного произведения векторов чисел с плавающей точкой можно воспользоваться инструкциями MULPD (_mm_mul_pd) и ADDPD (_mm_add_pd), вычисляющих попарное произведение и попарную сумму двух чисел. Кроме того, можно воспользоваться функцией ippsDotProd_64f из библиотеки IPP.

```
1 double DotProductSSE2(unsigned Size,const double* pA,const double* pB)
2 {
3     union
4     {
5         __m128d PackedSum;
```

[3]В наборе инструкций SSE3 появились команды, которые могут упростить эти действия.

Таблица 2.4: Приведенное время вычисления скалярного произведения различными методами, нс

	__int16			double		
Размерность	C++	SSE2	IPP	C++	SSE2	IPP
16	1,95	0,98	1,95	1,95	1,95	1,95
512	1,71	0,61	0,58	2,1	1,68	0,70
4096	1,68	0,53	0,58	1,83	1,54	0,70

```
6           double C[2];
7       };
8       __m128d PackedProduct;
9       PackedSum=_mm_setzero_pd();
10      for (unsigned i=0;i<Size;i+=2)
11      {
12          //переменожение пары целых чисел
13          PackedProduct=_mm_mul_pd(*(__m128d*)(pA+i), *(__m128d*)(pB+i));
14          PackedSum=_mm_add_pd(PackedSum, PackedProduct);
15      };
16      return C[0]+C[1];
```

В таблице 2.4 представлено среднее приведенное время вычисления скалярного произведения двух векторов различными методами (операциями языка C++, явными командами SSE2 и с помощью функций IPP) на процессоре AMD Athlon 64 X2 2,2ГГц. Можно заметить, что использование SIMD-инструкций SSE2 позволяет добиться 10-кратного ускорения целочисленных вычислений. Алгоритм, реализованный в библиотеке IPP, по сути совпадает с описанным здесь. Приведенная выше реализация скалярного произведения на основе SSE не обеспечивает никакого прироста производительности, т.к. не использует суперскалярную архитектуру процессора.

Таким образом, в большинстве случаев целесообразно применять библиотеки векторных математических функций, оптимизированные под данный процессор. Но при реализации нестандартных алгоритмов может быть необходимо явное использование инструкций векторной обработки данных. В последнем случае программа должна определять тип процессора и поддержку им соответствующего набора инструкций. На Intel-совместимых процессорах это может быть реализовано с помощью инструкции CPUID.

Лабораторная работа № 1. Исследование возможностей процессора

В данной работе необходимо исследовать описанные выше возможности центрального процессора. Исследование должно проводиться каждым студентом на собственном ПК. В отчете

необходимо указать технические характеристики использованного процессора, представить результаты каждого из нижеприведенных исследований и пояснить их.

1. Сравните время выполнения арифметических операций над числами с плавающей точкой одинарной и двойной точности.

2. Постройте массив, содержащий последовательность значений переменной RNGState (см. стр. 20), и модифицируйте эту программу, заменив вычисление RNGState обращением к соответствующему элементу этого массива. Сравните время выполнения исходной и модифицированной программы для различных значений порога перехода. Объясните различия.

3. Исследуйте время обращения к элементам массива целых чисел с номерами $0, d, 2d, 3d, \ldots$ для различных значений d. Объясните полученные результаты.

4. Исследуйте время записи данных в оперативную память в линейном и псевдослучайном порядке.

5. Модифицируйте вышеприведенную программу вычисления значения многочлена с d-кратным разворачиванием цикла, исключив необходимость дополнения вектора коэффициентов нулями.

6. Модифицируйте вышеприведенную программу вычисления скалярного произведения чисел с плавающей точкой с учетом суперскалярной архитектуры процессора. Сравните производительность Вашей реализации с производительностью, достигаемой при использовании соответствующей функции библиотеки IPP.

2.3.2 Другие приемы повышения производительности

В данном разделе приведено описание некоторых приемов, использование которых может помочь компилятору сгенерировать более эффективный код [19].

1. Использование для доступа к элементам массива оператора [] вместо указателя позволяет компилятору получить более точную информацию о том, к какой области памяти будет производиться доступ. При использовании указателей компилятор не может заранее определить, указывают они на одну или на разные области памяти. В связи с этим для каждой операции разыменования указателя компилятор вынужден генерировать инструкцию выборки из памяти. При использовании оператора [], если при этом не используются арифметические преобразования указателей, некоторые компоненты массива могут быть перемещены в регистры процессора, что существенно ускорит доступ к ним.

2. При использовании сложных логических конструкций в условных операторах необходимо учитывать принцип “короткого замыкания”. Напомним, что в языке C при вычислении логического ИЛИ нескольких операндов вычисления прекращаются при обнаружении первого ненулевого значения; оставшиеся операнды игнорируются. Аналогично, при вычислении логического И вычисления

прекращаются при обнаружении первого нулевого значения. Каждый операнд в подобной конструкции транслируется в инструкцию условного перехода. Вычислительные затраты, связанные с обработкой сложных логических выражений, включают в себя как затраты на вычисление операндов, так и затраты, связанные с возможным неверным предсказанием направлений условных переходов, причем их величина может зависеть от порядка операндов в выражении. При построении сложных логических выражений необходимо упорядочивать операнды с учетом следующих факторов:

- Сложность вычисления каждого из операндов.
- Наличие побочных эффектов при вычислении операндов.
- Вероятность принятия операндом значений true и false.
- Предсказуемость перехода.
- Издержки, связанные с возможным неправильным предсказанием перехода.

В большинстве случаев приходится экспериментально определять оптимальный порядок записи операндов в логическом выражении. Желательно также избегать громоздких логических конструкций, как по соображениям читабельности кода, так и для снижения нагрузки на блок предсказания ветвлений.

3. Желательно выравнивать указатели на динамически выделяемые массивы по 8- или 16-байтовой границе. Это ускоряет работу SIMD-инструкций. Пример выравнивания по 8-байтной границе:

```
int *p;//указатель на выделенную область памяти
int *np;//указатель на реально используемую выровненную область
p = (int *)malloc(sizeof(int) * number_of_ints + 7L);
np = (int*)((((long)(p)) + 7L) & (-8L));//указатель кратен 8
...
free(p);
```

4. Желательно избегать выгрузки в память данных с последующей их загрузкой через небольшое время. Целесообразно создать локальную переменную, содержащую копию этих данных. Например, фрагмент

```
for (k = 1; k < 100; k++)
    x[k] = x[k-1] + y[k];
```

целесообразно заменить на

```
t = x[0];
for (k = 1; k < VECLEN; k++) {
    t = t + y[k];
    x[k] = t;}
```

5. При использовании конструкции switch(x) желательно, чтобы возможные значения x были расположены достаточно плотно, что позволяет реализовать данную конструкцию в виде безусловного перехода по адресу, выбираемому из массива. Если это невозможно осуществить, компилятором будет сгенерирована последовательность сравнений x со всеми значениям, указанными в операторах case. В этом случае ветви переходов целесообразно разместить в порядке убывания вероятности их срабатывания, что позволяет минимизировать среднее выполняемое количество сравнений.

6. Величины, значения которых не должны меняться, следует объявлять с модификатором const, что позволит компилятору заранее вычислить некоторые выражения, а также снизить число обращений к памяти.

7. Из циклов следует удалять все операции, результат которых не зависит от значений счетчика цикла.

8. Необходимо избегать появления длинных цепочек зависимостей. Например, в фрагменте

```
1 sum = 0.0f;
2 for (i = 0; i < 100; i++)
3       sum += a[i];
```

   на каждой итерации цикла может выполняться ровно одно сложение, независимо от того, сколько инструкций способен одновременно выполнять процессор. Это связано с тем, что согласно спецификации языка C компилятор обязан сохранять порядок операций над числами с плаваюшей точкой. Если известно, что изменение их порядка не может оказать существенного влияния на конечный результат, то целесообразно развернуть цикл. Это позволит использовать как векторизованную обработку, так и параллельное исполнение инструкций. Вышеприведенный фрагмент может быть преобразован в

```
1 sum1 = 0.0;sum2 = 0.0;sum3 = 0.0;sum4 = 0.0;
2 for (i = 0; i < 100; i + 4) {
3       sum1 += a[i];
4       sum2 += a[i+1];
5       sum3 += a[i+2];
6       sum4 += a[i+3];
7 };
8 sum = (sum4 + sum3) + (sum1 + sum2);
```

9. Необходимо выделять общие подвыражения и заранее вычислять их. При обработке чисел с плавающей точкой необходимо учитывать возможную потерю точности. Например, фрагмент

```
1 double a, b, c, d, e, f;
2 e = b * c / d;
```

```
3 f = b / d * a;
```

целесообразно заменить на

```
1 double a, b, c, d, e, f, t;
2 t = b / d;
3 e = c * t;
4 f = a * t;
```

10. Поля структур необходимо располагать в порядке убывания их размеров. Кроме того, если планируется использование массива структур, размер структуры должен быть кратен размеру ее наибольшего поля. Данное правило позволяет выровнять поля структуры по границам, кратным их размеру, что ускоряет обращение к ним. Например, фрагмент

```
1 struct {
2 char a[5]; // наименьший размер (1 байт * 5)
3 long k; // 4 байта
4 double x; // наибольший размер (8 байт)
5 } baz;
```

следует переписать как

```
1 struct {
2 double x; // наибольший размер (8 байт)
3 long k; // 4 байта
4 char a[5]; // наименьший размер (1 байт * 5)
5 char pad[7]; // обеспечивает размер структуры, кратный 8
6 } baz;
```

Аналогичным образом следует упорядочивать локальные переменные функций. Некоторые компиляторы автоматически реализуют данный прием.

11. Предпочтительным является использование 32-битных целых чисел вместо 16- и 8-битных. По умолчанию современные процессоры Intel и AMD работают с 32-битными числами. Обработка чисел меньшей разрядности требует добавления к соответствующим командам специальных префиксов, наличие которых существенно замедляет декодирование инструкций. На процессорах, работающих в 64-битном режиме, использование 32-битных инструкций также позволяет избавиться от префикса, упрощая таким образом декодирование команд и сокращая объем исполнимого кода. Вместе с тем, при обработке больших массивов данных использование 64-битных операций может существенно снизить сложность вычислений. Например, умножение длинных целых чисел с помощью 64-битных инстукций требует в 4 раза меньшего числа операций, чем при использовании 32-битных команд.

12. Многие целочисленные операции над беззнаковыми (unsigned) целыми выполняются быстрее, чем над числами со знаком. Вместе с тем, преобразование целых со знаком в числа с плавающей точкой выполняется быстрее, чем для беззнаковых целых.

13. Функции, содержащие небольшое число инструкций (до 25), целесообразно объявлять как встраиваемые (inline). Это позволяет исключить издержки, связанные с вызовом функции.

14. Деление целых чисел на заранее известную константу может быть выполнено посредством намного более быстрых операций умножения, сложения и сдвига [18]. Деление на заранее известное число с плавающей точкой x может быть заменено умножением на заранее вычисленную величину $1/x$.

Лабораторная работа № 2. Реализация вычислительного алгоритма

В данной работе предлагается реализовать один из вычислительных алгоритмов как непосредственно, так и с использованием вышеописанных приемов повышения производительности (разворачивание циклов, векторизованная обработка и др.). Входные данные должны считываться из текстового файла, имя которого передается программе в качестве параметра. Выходные данные должны включать в себя результаты вычислений с использованием непосредственной и ускоренных реализаций алгоритма, а также время выполнения вычислений каждым из способов. Выходные данные должны выводиться в текстовый файл, имя которого передается программе в качестве параметра. Для получения достоверной оценки времени вычислений может потребоваться выполнить их несколько раз. Число таких итераций также должно быть указано в качестве параметра командной строки. В отчете необходимо обосновать их использование и сравнить производительность непосредственной и ускоренной реализаций алгоритма. Использование библиотек высокопроизводительных вычислений не допускается.

1. Алгоритм Винограда умножения целочисленных $n \times n$ матриц X и Y. Компоненты $Z = XY$ вычисляются по формуле

$$z_{ik} = \sum_{j=1}^{n/2}(x_{i,2j} + y_{2j-1,k})(x_{i,2j-1} + y_{2j,k}) - a_i - b_k + (x_{in}y_{nk}\,\mathbf{1}(n \text{ нечетно}))$$

$$a_i = \sum_{j=1}^{n/2} x_{i,2j}x_{i,2j-1};\; b_k = \sum_{j=1}^{n/2} y_{2j-1,k}y_{2j,k}. \qquad (2.7)$$

2. Алгоритм Винограда умножения вещественных $n \times n$ матриц X и Y.

3. Деление многочленов над $GF(2)$ в столбик. Многочлены задаются в виде шестнадцатиричного числа произвольной длины. Например, $x^4 + x + 1$ задается как 0x13. Результат должен быть представлен в таком же формате.

4. Вычисление линейной свертки целочисленных векторов длины n, т.е. $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}, i = 0..2n-2$. Коэффициенты a_i, b_i имеют тип __int16, c_i — __int64.

5. Вычисление циклической свертки целочисленных векторов длины n, т.е. $c_i = \sum_{j=0}^{n-1} a_j b_{(i-j) \bmod n}, i = 0..n-1$. Коэффициенты a_i, b_i имеют тип __int16, c_i — __int64.

6. Вычисление линейной свертки вещественных векторов длины n, т.е. $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}, i = 0..2n-2$. Коэффициенты a_i, b_i, c_i имеют тип float.

7. Вычисление циклической свертки вещественных векторов длины n, т.е. $c_i = \sum_{j=0}^{n-1} a_j b_{(i-j) \bmod n}, i = 0..n-1$. Коэффициенты a_i, b_i, c_i имеют тип float.

8. Вычисление линейной свертки двоичных векторов длины n, т.е. $c_i = \sum_{j=0}^{n-1} a_j b_{i-j} \bmod 2, i = 0..2n-2$. Коэффициенты a_i, b_i задаются в виде шестнадцатиричного числа произвольной длины. Например,в ектор $(1,1,0,0,1)$ задается как 0x13. Результат должен быть представлен в таком же формате.

9. Вычисление циклической свертки вещественных векторов длины n, т.е. $c_i = \sum_{j=0}^{n-1} a_j b_{(i-j) \bmod n} \bmod 2, i = 0..n-1$. Коэффициенты a_i, b_i задаются в виде шестнадцатиричного числа произвольной длины. Например, вектор $(1,1,0,0,1)$ задается как 0x13. Результат должен быть представлен в таком же формате.

10. Вычисление $a^{b_i} \bmod p, i = 1..n$. Все числа являются целыми и не превосходят $2^{32}-1$. Напомним, что если $y = \sum_{j \geq 0} y_j 2^j, y_j \in \{0,1\}$, то $x^y = \prod_{j \geq 0} x^{y_j 2^j}$.

11. Вычисление $a_i^b \bmod p, i = 1..n$. Все числа являются целыми и не превосходят $2^{32}-1$.

12. Умножение длинных целых чисел “в столбик”. Числа задаются в шестнадцатиричном виде.

Глава 3

Алгоритмы компьютерной алгебры

Рассматриваемые здесь вопросы называется также получисленными алгоритмами [10].

3.1 Анализ сложности алгоритмов

Понятие сложности алгоритма может быть интерпретировано многими различными способами, например:

- число различных операций, выполняемых программой, реализующей заданный алгоритм;
- время выполнения алгоритма на заданной ЭВМ;
- сложность алгоритма для понимания и реализации.

Ясно, что приведенные критерии в значительной степени яляются субъективными и точному теоретическому оцениванию, как правило, не поддаются. В данном разделе под сложностью алгоритма будет пониматься число арифметических операций, выполняемых непосредственно над обрабатываемыми данными или промежуточными значениями, полученными из них. При этом игнорируются затраты на организацию циклов, вызов подпрограмм, управление памятью и другие вспомогательные цели. Ясно, что при реализации вычислительных алгоритмов эти затраты необходимо минимизировать.

Многие быстрые алгоритмы решения различных задач являются рекурсивными. Сложность таких алгоритмов может быть выражена в виде рекуррентных соотношений. В данном разделе будут рассмотрены методы, позволяющие получить явные выражения для сложности.

3.1.1 Метод подстановки

Идея метода подстановки состоит в том, чтобы угадать решение, а затем доказать его.

Пример 3.1. Пусть сложность некоторого алгоритма решения задачи размерности n равна

$$T(n) = 2T(\lfloor n/2 \rfloor) + n.$$

Предположим, что $T(n) \leq cn \log n$ для некоторого c. Подберем c таким образом, чтобы оценка была верна при $n = 2, 3$. Это необходимо сделать непосредственно анализируя алгоритм. Предположим, что оценка верна для $\lfloor n/2 \rfloor$, т.е. $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$. Тогда $T(n) \leq 2c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + n \leq cn \log(n/2) + n = cn \log(n) - cn \log 2 + n \leq cn \log(n)$. Последний переход законен при $c \geq 1$.

Для угадывания соотношения можно воспользоваться аналогией с другими, ранее доказанными оценками. Кроме того, можно последовательно уточнять оценки.

3.1.2 Метод итераций

Если угадать правильное соотношение не удается, можно проитерировать его, получив ряд, а затем оценить его некоторым образом.

Пример 3.2. Рассмотрим соотношение

$$T(n) = 3T(\lfloor n/4 \rfloor) + n.$$

Подставляя его в себя, получим $T(n) = n + 3T(\lfloor n/4 \rfloor) = n + 3\lfloor n/4 \rfloor + 9T(\lfloor n/16 \rfloor) = n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor)$. После $\log_4 n$ итераций в правой части получится $T(1) = O(1)$. Таким образом, $T(n) \leq n + 3n/4 + 9n/16 + \ldots + 3^{\log_4 n} O(1) \leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + O(n^{\log_4 3}) = 4n + o(n) = O(n)$

Подобные преобразования могут привести к достаточно сложным выкладкам. Необходимо контролировать число шагов подстановки и сумму получающегося ряда. После нескольких первых шагов иногда можно угадать результат и доказать его по индукции (с меньшим количеством вычислений). Процесс подстановки можно изобразить в виде дерева рекурсии

Общий метод

Будем записывать $f(n) = \Theta(g(n))$, если $\exists c_1, c_2, n_0 : \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. Обозначение $f(n) = O(g(n))$ означает, что $\exists c, n_0 : \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$. Данное обозначение несколько отличается от принятого в матанализе. Введем также запись $f(n) = \Omega(g(n))$, означающую, что $\exists c, n_0 : \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$. Ясно, что свойство $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда имеют место $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. Кроме того, свойства $f(n) = O(g(n))$ и $g(n) = \Omega(f(n))$ равносильны.

Теорема 3.1. *Пусть $a \geq 1$ и $b > 1$ — некоторые константы, $f(n)$ — функция и для неотрицательных n функция $T(n)$ определена как*

$$T(n) = aT(n/b) + f(n).$$

Тогда:

1. *Если* $f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0$, *то* $T(n) = \Theta(n^{\log_b a})$.
2. *Если* $f(n) = \Theta(n^{\log_b a})$, *то* $T(n) = \Theta(n^{\log_b a} \log n)$.
3. *Если* $f(n) = \Omega(n^{\log_b a + \epsilon}), \epsilon > 0$ *и для достаточно больших* n $af(n/b) \leq cf(n), c < 1$, *то* $T(n) = \Theta(f(n))$.

Пример 3.3. Рассмотрим соотношение

$$T(n) = 9T(n/3) + n.$$

В данном случае $a = 9, b = 3, f(n) = n, n^{\log_b a} = \Theta(n^2)$. Таким образом, $T(n) = \Theta(n^2)$.

3.2 Операции над матрицами

3.2.1 Умножение произвольных матриц

Рассмотрим задачу вычисления $Z = XY, X \in \mathbb{K}^{m \times n}, Y \in \mathbb{K}^{n \times s}$, где $\mathbb{K}$ — некоторое коммутативное кольцо. Известно, что

$$z_{ik} = \sum_{j=1}^{n} x_{ij} y_{jk}, i = 1..m, k = 1..s$$

Непосредственное вычисление требует mns умножений и $ms(n-1)$ сложений. *Шмуэль Виноград* предложил способ замены половины умножений сложениями (предполагается коммутативность умножения):

$$z_{ik} = \sum_{j=1}^{n/2} (x_{i,2j} + y_{2j-1,k})(x_{i,2j-1} + y_{2j,k}) - a_i - b_k + (x_{in} y_{nk}\, \mathbf{1}(n \text{ нечетно}))$$

$$a_i = \sum_{j=1}^{n/2} x_{i,2j} x_{i,2j-1};\ b_k = \sum_{j=1}^{n/2} y_{2j-1,k} y_{2j,k}. \qquad (3.1)$$

Этот метод требует $\lceil n/2 \rceil ms + \lfloor n/2 \rfloor (m+s)$ умножений и $(n+2)ms + (\lfloor n/2 \rfloor - 1)(ms+m+s)$ сложений или вычитаний.

В общем случае число умножений может быть еще более уменьшено с помощью *алгоритма Штрассена*, который не требует коммутативности умножения. Пусть необходимо перемножить две 2×2 матрицы. Модификация Винограда алгоритма Штрассена:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} A & C \\ B & D \end{pmatrix} = \begin{pmatrix} aA + bB & w + v + (b - (c - a + d))D \\ w + u + d(B - (A + D - C)) & w + u + v \end{pmatrix},$$

$$u = (c-a)(C-D), v = (c+d)(C-A), w = aA + (c-a+d)(A+D-C), \qquad (3.2)$$

требует всего 7 умножений и 15 сложений. Этот подход может быть применен рекурсивно, что дает асимптотическое число умножений $O(n^{\log_2 7}) = O(n^{2.8074})$.

3.2.2 Умножение двоичных матриц

Рассмотрим задачу умножения двоичных $n \times n$ матриц над полукольцом $(\{0,1\}, \vee, \wedge)$. Алгоритм Штрассена не может быть непосредственно применен к решению этой задачи, т.к. он требует, чтобы множество с операцией сложения образовывали абелеву группу. Тем не менее, множество $\{0,1\}$ может быть *погружено в кольцо* целых чисел по модулю $n+1$ с заменой операций $\vee \to +, \wedge \to *$. После этого умножение может быть выполнено с помощью алгоритма Штрассена над кольцом. Затем результат должен быть отображен в бинарное множество путем замены всех ненулевых элементов полученной матрицы на 1.

Можно также построить специализированный алгоритм перемножения двоичных матриц, который, однако, асимптотически менее эффективен. Разобьем матрицы на $n/\log_2 n$ групп строк и столбцов: $X = (X_1|X_2|\ldots|X_{n/\log_2 n})$, $Y = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_{n/\log_2 n} \end{pmatrix}$. Тогда $XY = \bigvee_{i=1}^{n/\log_2 n} X_i Y_i$. Заметим, что каждая строка матрицы X_i содержит $\log_2 n$ элементов, равных 0 или 1. Наличие 1 в строке означает, что соответствующая строка матрицы Y_i должна быть включена в дизъюнкцию, т.е. каждая строка матрицы X_i задает некоторое подмножество строк Y_i. Эти подмножества можно перебирать таким образом, чтобы очередное подмножество отличалось от *какого-либо из предыдущих добавлением одного* элемента. Таким образом, произведение $X_i Y_i$ может быть вычислено не более чем за n^2 операций, а общая сложность составляет $O(n^3/\log_2 n)$ операций. Естественно, что этот метод может быть использован и в том случае, когда Y является вектором-столбцом. Данный алгоритм был разработан В.Л. Арлазаровым, Е.А. Диницем, М.А. Кронродом и И.А. Фараджевым и в зарубежной литературе носит название *алгоритма четырех русских*.

Если операцию $\vee$ заменить на $\oplus$ (т.е. сложение по модулю 2), то рассматриваемая алгебра станет кольцом (на самом деле полем $GF(2)$). Это дает возможность рассматривать последовательность строк матриц X_i, различающихся друг от друга *добавлением или удалением* ровно одного элемента, т.е. строки могут быть переупорядочены в соответствии с кодом Грея.

3.2.3 Алгоритмы работы с разреженными матрицами

Во многих приложениях приходится выполнять вычисления с матрицами, большинство элементов которых равны нулю. Такие матрицы носят название *разреженных*. Ясно, что операции умножения и сложения чисел, одно из которых заведомо равно нулю, можно реализовать намного более эффективно чем в случае произвольных чисел. Поэтому алгоритмы работы с разреженными матрицами, как правило, имеют сложность существенно меньшую, чем аналогичные алгоритмы для матриц произвольного вида. Как правило, сложность определяется числом N ненулевых элементов матрицы.

Для работы с разреженными матрицами необходимо надлежащим образом выбрать структуры данных. Выбор структуры данных определяется используемым алгоритмом. Наибольшее распространение получили следующие способы представления разреженной $m \times n$ матрицы A:

1. (старый Йельский формат). Ненулевые элементы матрицы записываются построчно в массиве V размерности N. Массив R размерности $m+1$ содержит номер позиции в массиве V, содержащей первый ненулевой элемент каждой строки. Таким образом, число ненулевых элементов в строке i равно $R[i+1]-R[i]$. Массив C размерности N содержит номера столбцов ненулевых элементов каждой строки.

2. (новый Йельский формат). Из матрицы выделяется ее диагональ, все элементы которой записываются в первые N элементов массива V и не учитываются далее при построении массивов R и C. Элемент $N+1$ массива V полагается равным нулю для упрощения индексации. Массивы R и C объединяются в один массив I.

Указанные форматы часто используются прикладным программным обеспечением. Если ненулевые элементы матрицы сгруппированы некоторым образом, может потребоваться комбинирование различных алгоритмов с учетом известных свойств матрицы. Например, матрица может быть представлена в виде блочной матрицы с небольшим числом плотных ненулевых блоков. В этом случае может быть использован один из вышеописанных алгоритмов работы с разреженными матрицами, в котором сложение и умножение выполняется не над скалярными величинами, а над матрицами.

Пример 3.4. Рассмотрим матрицу $A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 6 \\ 7 & 8 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 0 \\ 0 & 0 & 11 & 0 & 0 & 0 \end{pmatrix}$. В старом Йельском формате данная матрица представима как $R = (1,4,5,7,10,11,12), C = (1,4,6,4,3,6,1,2,4,5,3), V = (1,2,3,4,5,6,7,8,9,10,11)$. В новом Йельском формате она записывается как $I = (\underbrace{8,10,11,12,14,14,15}_{R}, \underbrace{4,6,4,6,1,2,3}_{C}), V = (1,0,5,9,10,0,\mathbf{0},2,3,4,6,7,8,11)$. За счет введения служебного элемента $\mathbf{0}$ в массив V номер позиции в массиве V, задающей значение некоторого ненулевого внедиагонального элемента совпадает с номером позиции в массиве I, задающей номер его столбца.

Сложение матриц, заданных в старом Йельском формате, сводится к операции слияния списков ненулевых элементов одинаковых строк различной матрицы и сложению общих элементов. Вычисление произведения $y = Ax$ состоит в суммировании элементов массива x, задаваемых элементами массива C, с весовыми коэффициентами, задаваемыми элементами массива V, т.е. $y_i = \sum_{j:A_{ij}\neq 0} A_{ij}x_j$. Произведение $y = xA$ может быть вычислено следующим образом: $y := 0; y_j = y_j + x_i A_{ij}, i = 1..m, A_{ij} \neq 0$

3.3 Операции над многочленами

3.3.1 Билинейные формы

Рассмотрим задачу вычисления

$$z_k = \sum_{i=1}^{m} \sum_{j=1}^{n} t_{ijk} x_i y_j, 1 \le k \le s,$$

где t_{ijk} — некоторые коэффициенты. Это выражение называется *билинейной формой*, задаваемой тензором (t_{ijk}). Несложно заметить, что оно линейно относительно y и x. Ограничимся нормальными вычислительными схемами, в которых все умножения производятся между линейными комбинациями x и линейными комбинациями y. Таким образом, мы строим r произведений

$$w_l = (a_{1l}x_1 + a_{2l}x_2 + \ldots + a_{ml}x_m)(b_{1l}y_1 + b_{2l}y_2 + \ldots + b_{nl}y_n), l = 1..r,$$

а результат вычисляем как

$$z_k = \sum_{l=1}^{r} c_{kl} w_l.$$

Сопоставляя эти выражения, получим, что нормальная схема вычислений корректна тогда и только тогда, когда

$$t_{ijk} = \sum_{l=1}^{r} a_{il} b_{jl} c_{kl}.$$

Заметим, что алгоритм Винограда перемножения матриц не является нормальной вычислительной схемой, т.к. там x и y в ходе вычислений смешиваются.

Нормальные вычислительные схемы удобно записывать в матричном виде как $z = C\left(Ax \cdot By\right)$, где $\cdot$ обозначает покомпонентное произведение векторов, A и B называются матрицами предварительных сложений (предсложений), а матрица C — матрица последующих сложений (постсложений).

Ненулевой тензор называют тензором ранга 1, если $\exists(a_1, \ldots, a_m), (b_1, \ldots, b_n), (c_1, \ldots, c_s) : \forall i, j, k : t_{ijk} = a_i b_j c_k$. Рангом тензора называется такое минимальное число r, что t_{ijk} выражается в виде суммы r тензоров ранга 1. Ранг тензора есть минимальное число умножений в цепочке при нормальном вычислении соответствующей билинейной формы.

Ограничимся рассмотрением задач вычисления линейной и циклической свертки. n-точечной *линейной сверткой* многочленов (или векторов их коэффициентов) $a(x) = \sum_{i=0}^{n-1} a_i x^i$ и $b(x) = \sum_{i=0}^{n-1} b_i x^i$ называется многочлен (или вектор его коэффициентов)

$$c(x) = a(x)b(x) = \sum_{i=0}^{2n-2} x^i \sum_{j=0}^{n-1} a_j b_{i-j}.$$

Здесь предполагается, что коэффициенты a_j и b_i с индексами, выходящими за допустимые границы, равны нулю. n-точечной циклической сверткой многочленов (или векторов их коэффициентов) называется многочлен (или вектор его коэффициентов)

$$c(x) = a(x)b(x) \pmod{x^n - 1} = \sum_{i=0}^{n-1} x^i \sum_{j=0}^{n-1} a_j b_{((i-j))},$$

где $((i)) \equiv i \bmod n$.

3.3.2 Алгоритмы Карацубы и Тоома-Кука вычисления свертки

Рассмотрим задачу вычисления линейной свертки $c_0 + c_1 x + c_2 x^2 = (a_0 + a_1 x)(b_0 + b_1 x)$. Результат является многочленом второй степени и однозначно определяется своими значениями в трех различных точках. В случае вещественных чисел в качестве таких точек удобно выбрать $c(-1), c(0), c(1)$. Тогда $c(-1) = a(-1)b(-1) = (a_0 - a_1)(b_0 - b_1), c(0) = a(0)b(0) = a_0 b_0, c(1) = a(1)b(1) = (a_0 + a_1)(b_0 + b_1)$. Следовательно, $c(x) = c(0)(1 - x^2) + \frac{1}{2}c(1)(x^2 + x) + \frac{1}{2}c(-1)(x^2 - x)$. Константы $\frac{1}{2}$ удобно внести в выражения для $c(1)$ и $c(-1)$. *Алгоритм быстрого умножения Тоома-Кука* может быть записан как

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \\ 1/2 & -1/2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right).$$

Видно, что данный алгоритм требует 3 умножений и 7 сложений. Во многих случаях многочлен $a(x)$ является фиксированным, что позволяет вычислить $a(0), a(1), a(-1)$ заранее. В некоторых случаях одно из этих значений оказывается нулевым, что позволяет дополнительно сократить число операций.

Рассмотрим модификацию алгоритма Тоома-Кука, демонстрирующую важный прием, используемый при построении быстрых алгоритмов. Заметим, что $c_{2n-2} = a_{n-1} b_{n-1}$. Тогда $c'(x) = a(x)b(x) - x^{2n-2} a_{n-1} b_{n-1}$ имеет степень, на единицу меньшую, чем $c(x)$. Для случая линейной 2×2 свертки это приводит к $c'(0) = a(0)b(0) = a_0 b_0$, $c'(1) = a(1)b(1) - a_1 b_1 = (a_0 + a_1)(b_0 + b_1) - a_1 b_1$. Таким образом, $c_0 = c'_0 = a_0 b_0, c_1 = c'_1 = (a_0 + a_1)(b_0 + b_1) - a_1 b_1 - a_0 b_0, c_2 = a_1 b_1$. Описанный *алгоритм Карацубы* может быть записан в следующем виде:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right).$$

Таким образом, требуется 3 умножения и 4 сложения. Этот алгоритм может быть легко адаптирован для вычисления циклической свертки:

$$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & -1 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right).$$

Основная идея алгоритма Тоома-Кука и его аналогов состоит в выборе таких интерполяционных точек, вычисление значений многочленов (и последующая интерполяция) в которых сводятся к элементарным действиям типа сложения и вычитания. В случае поля вещественных чисел эту идею можно обобщить на случай многочленов большей степени, используя точки $\pm 2, \pm 3$ и т.д., причем умножения на эти числа должны заменяться на сложения. Однако достаточно быстро этот метод становится чрезмерно громоздким. Более того, не во всяких полях оказывается возможным найти достаточное количество удобных для вычислений точек, необходимых для построения алгоритма. В связи с этим, как правило, практикуется использование т.н. гнездовых методов.

3.3.3 Алгоритм Винограда

Предположим, что необходимо вычислить

$$c(x) = a(x)b(x) \bmod m(x).$$

Алгоритмы Тоома-Кука и Карацубы основывались на вычислении значений многочленов $a(x)$ и $b(x)$ в наборе точек, перемножении этих значений и восстановлении результата с помощью интерполяции. Вычисление значений многочлена в отдельных точках можно представить как $a(x_i) \equiv a(x) \bmod (x - x_i)$. Это позволяет рассматривать интерполяцию как частный случай формулы обращения китайской теоремы об остатках.

Теорема 3.2 (Китайская теорема об остатках для многочленов)**.** *Пусть $m^{(1)}(x), \ldots, m^{(r)}(x)$ — многочлены от одной переменной x с коэффициентами из некоторого поля. Пусть $m(x) = m^{(1)}(x) \cdot \ldots \cdot m^{(r)}(x)$ и пусть также $u^{(1)}(x), \ldots, u^{(r)}(x)$ — некоторые многочлены. Тогда существует ровно один многочлен $u(x)$:*

$$(0 \leq \deg u(x) < \deg m(x)) \wedge (u(x) \equiv u^{(j)}(x) \bmod m^{(j)}(x), j = 1..r),$$

причем

$$u(x) = ((u^{(1)}(x)M_1(x) + \ldots + u^{(r)}(x)M_r(x)) \bmod m(x)), \tag{3.3}$$

где $M_j(x) = n^{(j)}(x)((n^{(j)}(x))^{-1} \bmod m^{(j)}(x)), n^{(j)}(x) = m(x)/m^{(j)}(x)$.

Пусть $m(x) = \prod_{j=1}^{r} m^{(j)}(x)$. Вычислим $a^{(j)}(x) \equiv a(x) \bmod m^{(j)}(x), b^{(j)}(x) \equiv b(x) \bmod m^{(j)}(x)$. Искомый многочлен может быть восстановлен по $c^{(j)}(x) \equiv a^{(j)}(x)b^{(j)}(x) \bmod m^{(j)}(x)$. Снижение сложности достигается в том случае, когда коэффициенты многочленов $m^{(j)}(x)$ являются "вычислительно простыми". Например, это могут быть небольшие по модулю целые или рациональные числа. Данный метод носит название *алгоритма Винограда*.

В том случае, когда необходимо вычислить циклическую свертку, многочлен $m(x)$ однозначно задан и равен $x^n - 1$. В случае необходимости вычисления линейной свертки многочлен $m(x)$ должен быть выбран таким образом, чтобы приведение по модулю не влияло на результат, т.е. он должен иметь степень не менее $\deg a(x) + \deg b(x) +$

1. Это оставляет достаточно большую свободу для выбора, причем вычислительные алгоритмы, построенные на основе различных многочленов, как правило, имеют несколько различающуюся сложность.

Матрицы билинейной формы для алгоритма Винограда формируются на основе матриц вычисления остатка от деления на многочлены $m^{(j)}(x)$, матриц вычисления произведений $c^{(j)}(x)$ и матрицы, соответствующей операции восстановления результата по формуле (3.3).

Пример 3.5. Рассмотрим построение быстрого алгоритма линейной свертки 3-точечного вектора (т.е. многочлена второй степени) с двухточечным. Пусть $a(x) = a_1x + a_0, b(x) = b_2x^2 + b_1x + b_0$. Непосредственное вычисление $c(x) = a(x)b(x)$ требует 6 умножений и 2 сложений. Пусть $m(x) = x(x-1)(x^2+1) = x^4 - x^3 + x^2 - x$. Вычеты равны

$$\begin{aligned} a^{(1)}(x) &= a_0 & b^{(1)}(x) &= b_0 \\ a^{(2)}(x) &= a_1 + a_0 & b^{(2)}(x) &= b_2 + b_1 + b_0 \\ a^{(3)}(x) &= a_1x + a_0 & b^{(3)}(x) &= b_1x + (b_0 - b_2) \end{aligned}$$

Следовательно,

$$\begin{aligned} c^{(1)}(x) &= a_0b_0 \\ c^{(2)}(x) &= (a_1 + a_0)(b_2 + b_1 + b_0) \\ c^{(3)}(x) &\equiv (a_1x + a_0)(b_1x + (b_0 - b_2)) \bmod (x^2+1) \end{aligned}$$

Вычисление первых двух вычетов требует двух операций умножения. Вычисление третьего вычета совпадает со структурой умножения комплексных чисел. Непосредственное его вычисление может быть выполнено как

$$c^{(3)}(x) = x(a_0^{(3)}b_1^{(3)} + a_1^{(3)}b_0^{(3)}) + (a_0^{(3)}b_0^{(3)} - a_1^{(3)}b_1^{(3)}).$$

Однако вычисления могут быть упрощены, если воспользоваться аналогом алгоритма Карацубы. Вычислим $a^{(3)}(x)b^{(3)}(x) - x^2a_1^{(3)}b_1^{(3)} = a_0^{(3)}b_0^{(3)} + x((a_0^{(3)} + a_1^{(3)})(b_0^{(3)} + b_1^{(3)}) - a_1^{(3)}b_1^{(3)} - a_0^{(3)}b_0^{(3)})$. Дополняя это выражение членом, образующимся при приведении по модулю x^2+1, получим $c^{(3)}(x) = a_0^{(3)}b_0^{(3)} - a_1^{(3)}b_1^{(3)} + x((a_0^{(3)} + a_1^{(3)})(b_0^{(3)} + b_1^{(3)}) - a_1^{(3)}b_1^{(3)} - a_0^{(3)}b_0^{(3)})$. Окончательный результат должен быть восстановлен в соответствии с китайской теоремой об остатках:

$$c(x) = -(x^3-x^2+x-1)c^{(1)}(x) + \frac{1}{2}(x^3+x)c^{(2)}(x) + \frac{1}{2}(x^3-2x^2+x)c^{(3)}(x) \bmod (x^4-x^3+x^2-x)$$

Это равенство может быть перезаписано как

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ 1 & 0 & -2 & 0 \\ -1 & 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} c_0^{(1)} \\ \frac{1}{2}c_0^{(2)} \\ \frac{1}{2}c_0^{(3)} \\ \frac{1}{2}c_1^{(3)} \end{pmatrix}$$

Окончательно собирая алгоритм, получим

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ 1 & 0 & -2 & 0 \\ -1 & 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & -1 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \\ 1/2 & 0 \\ 1/2 & 1/2 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \right)$$

Перемножая матрицы постсложений, получим

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 1 & -2 \\ 1 & 0 & -2 & 0 & 2 \\ -1 & 1 & 2 & -1 & 0 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \\ 1/2 & 0 \\ 1/2 & 1/2 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \right)$$

Полученный алгоритм можно улучшить, выбрав многочлен $m(x)$ несколько меньшей степени, чем это требуется, аналогично тому, как было сделано при построении алгоритма Карацубы. С помощью этого приема можно уменьшить число умножений и сложений.

Реализация алгоритма Винограда требует наличия эффективных процедур вычисления произведений многочленов по модулю неприводимых многочленов, т.е. $c^{(k)}(x) = a^{(k)}(x)b^{(k)}(x) \bmod m^{(k)}(x)$. Наиболее прямой метод состоит в вычислении линейной свертки и последующем приведении ее по модулю $m^{(k)}(x)$. Если степень исходных многочленов равна $n-1$, это требует не менее $2n-1$ умножений. Это позволяет использовать ранее построенные быстрые алгоритмы линейной свертки, модифицировав в них матрицы постсложений.

Какой-либо общей теории минимизации числа сложений не разработано, поэтому в этой части приходится полагаться на эвристические приемы. Применение алгоритма Винограда к многочленам больших степеней, как правило, приводит к большому числу сложений. Поэтому его целесообразно комбинировать с гнездовыми алгоритмами.

Сформулируем некоторые фундаментальные свойства операций над многочленами:

1. Никакой алгоритм вычисления линейной свертки двух многочленов длин L и N не может содержать число умножений, меньшее чем $L+N-1$.

2. Если число простых делителей многочлена $p(x)$ равно t, никакой алгоритм вычисления произведения многочленов (достаточно большой степени n) по модулю $p(x)$ не может содержать число умножений, меньшее чем $2n-t$.

Хорошие алгоритмы линейных и циклических сверток табулированы [6, 5].

3.3.4 Перенос алгоритмов на поля другой природы

Всякий быстрый алгоритм представляет собой некоторое тождество, опирающееся на свойства ассоциативности, коммутативности и дистрибутивности, справедливые для того

поля, над которым он был построен. В связи с этим построенный быстрый алгоритм может использоваться и в любом расширении заданного поля. Однако алгоритм, построенный специально для расширенного поля, может оказаться более эффективным.

В некоторых случаях удается воспользоваться алгоритмами, построенными для полей иной природы. Например, рассмотрим задачу вычисления свертки (линейной или циклической), возникающую во многих задачах цифровой обработки сигналов. В большинстве сигнальных процессоров используется представление вещественных чисел с фиксированной запятой, т.е. в виде целых чисел. Если можно гарантировать, что результат выполнения операции не превысит некоторого достаточно большого значения p', то можно найти наименьшее простое число $p > p'$ и выполнять все вычисления по модулю p, т.е. в поле $GF(p)$, для которого могут существовать более эффективные вычислительные алгоритмы, чем для поля вещественных или кольца целых чисел.

Если построен алгоритм линейной или циклической свертки над полем вещественных чисел, то в некоторых случаях его удается адаптировать для конечного поля $GF(p^m)$. Пусть $\mathcal{S}(a,b)$ — билинейная форма, соответствующая определению свертки (которое не зависит от поля), а $\mathcal{F}(a,b)$ — билинейная форма, задающая некоторый быстрый алгоритм. Домножим тождество $\mathcal{F}(a,b) = \mathcal{S}(a,b)$ на минимальное число L так, чтобы избавиться от знаменателей в $\mathcal{F}(a,b)$. Полученное тождество приведем по модулю p. Если $L \neq 0 \bmod p$, полученное выражение задает быстрый алгоритм свертки для поля $GF(p^m)$.

Пример 3.6. Рассмотрим алгоритм трехточечной циклической свертки над вещественными числами, задаваемый тождествами

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & -1 \\ 1 & -1 & -1 & 2 \\ 1 & 0 & 1 & -1 \end{pmatrix} \left(\begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 1 & 0 & -1 \\ 0 & 1 & -1 \\ \frac{1}{3} & \frac{1}{3} & -\frac{2}{3} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & -2 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \right)$$

Домножая это тождество на 3 и приводя результат по модулю 2, получим быстрый алгоритм циклической свертки для поля $GF(2^m)$:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \right)$$

С другой стороны, лучший алгоритм двухточечной циклической свертки над вещественным полем имеет вид

$$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \left(\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right).$$

Т.к. он содержит четные знаменатели, перенос в поле характеристики два невозможен. Причина этого состоит в том, что в этом поле $x^2 - 1 = (x+1)^2$, что делает невозможным

использование алгоритма Винограда. В связи с этим приходится использовать алгоритм с тремя умножениями:

$$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right).$$

В зависимости от свойств многочлена $x^n - 1$ сложность алгоритма циклической свертки над конечным полем может быть как меньше, так и больше чем в случае вещественных чисел.

Для построения быстрых алгоритмов вычислений над комплексным полем могут использоваться расширенные конечные поля $GF(p^2), p$ — простое число.

Вычисления в конечном поле можно вложить в вещественное, комплексное или какое-либо иное подходящее поле. Для этого представим элементы поля $GF(p^m)$ в виде многочленов по модулю соответствующего неприводимого многочлена $\pi(z)$, т.е. $a_i = \sum_{j=0}^{m-1} a_{ij} z^j$. Тогда коэффициенты линейной свертки могут быть представлены как $c_i = \sum_{k=0}^{n-1} a_k b_{i-k} = \sum_{k=0}^{n-1} \sum_{l=0}^{m-1} \sum_{l'=0}^{m-1} a_{kl} b_{i-k,l'} z^{l+l'} \pmod{p} \pmod{\pi(z)}$. Это выражение представляет собой двумерную свертку, которую можно вычислять вышеописанными методами.

Кроме того, быстрые алгоритмы справедливы в некоторых кольцах, образованных на основе соответствующих полей.

3.3.5 Гнездовые алгоритмы свертки

Двумерная свертка представляет собой операцию, задаваемую на паре двумерных таблиц. Примером использования двумерной свертки является операция двумерной фильтрации, используемая при обработке изображений. Двумерной линейной сверткой таблицы данных $b_{k',k''}, k' = 0..N'-1, k'' = 0..N''-1$ и таблицы фильтра $a_{j',j''}, j' = 0..L'-1, j'' = 0..L''-1$ называется таблица значений $c_{i',i''} = \sum_{k'=0}^{N'-1} \sum_{k''=0}^{N''-1} a_{i'-k',i''-k''} b_{k',k''}$. Это определение может быть обобщено как на случай циклической свертки, так и на случай больших размерностей.

Заметим, что каждую из таблиц можно представить или как многочлен с векторными коэффициентами, или как многочлен от двух переменных $a(x,y) = \sum_{j'=0}^{L'-1} \sum_{j''=0}^{L''-1} a_{j'j''} x^{j'} y^{j''} = \sum_{j'=0}^{L'-1} a_{j'}(y) x^{j'}$. Это дает возможность определить двумерную линейную свертку или как как произведение многочленов от двух переменных, или как линейную свертку многочленов от одной переменной:

$$c(x,y) = a(x,y)b(x,y).$$

Непосредственное вычисление линейной свертки требует $L'L''N'N''$ умножений, что абсолютно неприемлемо для реализации.

Аналогично, двумерная (n', n'')-точечная циклическая свертка может быть определена как

$$c(x,y) = a(x,y)b(x,y) \pmod{x^{n'} - 1} \pmod{y^{n''} - 1}.$$

Быстрые алгоритмы, построенные для одномерных сверток над полями, могут быть применены и для вычисления многомерных сверток. Для этого необходимо упорядочить переменные некоторым образом и рассмотреть задачу вычисления свертки многочленов, скажем, от переменной x, коэффициентами которого являются многочлены от переменной y. Применение быстрого алгоритма вычисления свертки приведет к необходимости сложения и умножения коэффициентов этого многочлена. Т.к. они также являются многочленами, вычисление их сверток может быть снова выполнено с помощью соответствующих быстрых алгоритмов, но теперь уже умножения будут являться обычными умножениями над полем.

3.3.6 Итеративные алгоритмы

В некоторых случаях задачу вычисления одномерной свертки удается решить путем ее сведения к многомерной свертке. Это может быть полезно в тех случаях, когда применение специализированного алгоритма вычисления свертки (например, Винограда) приводит к слишком большому числу сложений. *Алгоритм Агарвала-Кули* основывается на китайской теореме об остатках для целых чисел.

Пусть надо вычислить компоненты циклической свертки

$$c_i = \sum_{k=0}^{n-1} a_{((i-k))} b_k.$$

Предположим, что $n = n'n'', (n', n'') = 1$. Заменим индексы i, k на пары (i', i'') : $i' \equiv i \bmod n', i'' \equiv i \bmod n'', (k', k'') : k' \equiv k \bmod n', k'' \equiv k \bmod n''$. В соответствии с китайской теоремой об остатках, старые индексы можно восстановить по формулам $i = N''n''i' + N'n'i'' \bmod n, k = N''n''k' + N'n'k'' \bmod n, N'n' + N''n'' = 1$. Тогда циклическую свертку можно записать как

$$c_{N''n''i'+N'n'i''} = \sum_{k=0}^{n-1} a_{N''n''(i'-k')+N'n'(i''-k'')} b_{N''n''k'+N'n'k''}.$$

Определим вспомогательные переменные

$$a_{i',i''} = a_{N''n''i'+N'n'i''}, i' = 0..n'-1, i'' = 0..n''-1$$
$$b_{k',k''} = b_{N''n''k'+N'n'k''}, k' = 0..n'-1, k'' = 0..n''-1$$
$$c_{j',j''} = c_{N''n''j'+N'n'j''}, j' = 0..n'-1, j'' = 0..n''-1$$

Тогда циклическая свертка может быть записана как

$$c_{i',i''} = \sum_{k'=0}^{n'-1} \sum_{k''=0}^{n''-1} a_{((i'-k')),((i''-k''))} b_{k'k''}$$

Здесь индексы вычисляются по модулям n' и n''. Таким образом, получена двумерная циклическая свертка. Заметим, что до настоящего времени никакого снижения

сложности не получено. Число умножений по прежнему составляет $(n'n'')^2$. Но если теперь воспользоваться каким-либо быстрым алгоритмом двумерной циклической свертки (построенном на основе более коротких алгоритмов одномерной циклической свертки), то сложность может существенно снизиться. Она будет равна

$$A(n) = n'A(n'') + M(n'')A(n')$$
$$M(n) = M(n')M(n''),$$

где $M(n)$ и $A(n)$ — число умножений и сложений, требуемое для вычисления n-точечной циклической свертки. Заметим, что

- Общее число умножений не зависит от порядка выбора чисел n', n''.
- Общее число сложений зависит от порядка их выбора.
- Использование элементарного одномерного вычислительного алгоритма с меньшим числом умножений может привести к снижению как общего числа умножэеений, так и общего числа сложений.

Пример 3.7. Рассмотрим построение алгоритма 6-точечной циклической свертки $c(x) = a(x)b(x) \bmod (x^6 - 1)$ над $GF(2^m)$. Перегруппируем коэффициенты: $a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 \longrightarrow (a_0 + a_4z + a_2z^2) + y(a_3 + a_1z + a_5z^2)$. Воспользуемся одним из быстрых алгоритмов двумерной циклической свертки $u(y) = w(y)v(y) \bmod (y^2 - 1)$:

$$\begin{pmatrix} u_0 \\ u_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} \right)$$

Подставляя перегруппированные многочлены, получим

$$\begin{pmatrix} c_0 + c_4z + c_2z^2 \\ c_3 + c_1z + c_5z^2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad \left(\begin{pmatrix} (a_0 + a_3) + (a_1 + a_4)z + (a_2 + a_5)z^2 \\ a_3 + a_1z + a_5z^2 \\ a_0 + a_4z + a_2z^2 \end{pmatrix} \cdot \begin{pmatrix} b_0 + b_4z + b_2z^2 \\ (b_0 + b_3) + (b_1 + b_4)z + (b_2 + b_5)z^2 \\ (b_0 + b_3) + (b_1 + b_4)z + (b_2 + b_5)z^2 \end{pmatrix} \right)$$

Воспользуемся теперь быстрым алгоритмом трехточечной циклической свертки $p(z) = q(z)r(z) \bmod (z^3 - 1)$

$$\begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \left(\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix} \right)$$

В результате получим

$$\begin{pmatrix} c_0 \\ c_4 \\ c_2 \\ c_3 \\ c_1 \\ c_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \left(\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} \right).$$

Рассмотрим еще один способ сведения одномерной свертки к многомерной. Пусть необходимо вычислить $c(x) = a(x)b(x), \deg a(x) = MN - 1, \deg b(x) = ML - 1$. Преобразуем исходные многочлены следующим образом:

$$a(y, z) = \sum_{i=0}^{N-1} \left(\sum_{k=0}^{M-1} a_{Ml+k} y^k \right) z^i.$$

Преобразование $b(x)$ выполняется аналогично. Заметим, что $a(x) = a(x, x^M)$. Тогда $c(y, z) = a(y, z)b(y, z)$ может быть вычислено с помощью быстрого гнездового алгоритма и $c(x) = x(x, x^M)$. Число умножений в полученном алгоритме равно произведению числа умножений двух используемых элементарных алгоритмов: линейной свертки последовательностей длины L и N и двух последовательностей длины M. В том случае, когда степени исходных многочленов не удается разложить указанным образом, они могут быть дополнены нулями до удобных значений. В этом случае необходимо проверить получившиеся матрицы пред- и постсложений на предмет наличия нулевых и повторяющихся столбцов и строк.

Пример 3.8. Рассмотрим вычисление линейной одномерной 4-точечной свертки, т.е. $c(x) = (a_0 + a_1x + a_2x^2 + a_3x^3)(b_0 + b_1x + b_2x^2 + b_3x^3)$ Это сводится к $c_0(y)z^0 + c_1(y)z^1 + c_2(y)z^2 = ((a_0 + a_1y)z^0 + (a_2 + a_3y)z^1)((b_0 + b_1y)z^0 + (b_2 + b_3y)z^1)$. Проитерируем дважды алгоритм Карацубы:

$$\begin{pmatrix} c_0(y) \\ c_1(y) \\ c_2(y) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 + a_1y \\ a_2 + a_3y \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 + b_1y \\ b_2 + b_3y \end{pmatrix} \right)$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} a_0 + a_1y \\ (a_0 + a_2) + (a_1 + a_3)y \\ a_2 + a_3y \end{pmatrix} \cdot \begin{pmatrix} b_0 + b_1y \\ (b_0 + b_2) + (b_1 + b_3)y \\ b_2 + b_3y \end{pmatrix} \right)$$

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 & 0 & -1 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \left(P \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot P \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \right), P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Данный алгоритм содержит 9 умножений и 20 сложений, в то время как оптимальный алгоритм требует 7 умножений, но большего числа сложений. Данный алгоритм можно проитерировать и получить алгоритм 16-точечной свертки, содержащий 81 умножение и т.д.

Одним из наиболее важных преимуществ итерированного алгоритма Карацубы перед другими быстрыми алгоритмами свертки является возможность его использования на любых длинах. Заметим, что число операций, необходимое для вычисления n-точечной линейной свертки (т.е. перемножения многочленов степени $n-1$) равно $T(n) = 3T(n/2) + 3n - 2$, т.е. $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.59})$. Таким образом, для достаточно больших n применение итерированного алгоритма Карацубы обеспечивает снижение числа как умножений, так и сложений.

При вычислении циклической свертки длины $n = p^\lambda$, где p — простое число, алгоритм Агарвала-Кули неприменим. В этом случае может быть полезным следующий прием. Пусть $a(x) = \sum_{i=0}^{p-1} x^i \sum_{k=0}^{n/p-1} a_{i+pk}x^{pk} = \sum_{i=0}^{p-1} x^i a_i(x^p), b(x) = \sum_{i=0}^{p-1} x^i b_i(x^p)$. Тогда

$$c(x) = \sum_{k=0}^{2p-2} x^k \sum_{i=0}^{k} a_{k-i}(x^p)b_i(x^p) \bmod (x^n - 1). \tag{3.4}$$

Необходимо отметить, что индексы многочленов здесь не приводятся по модулю. Считается, что $a_i(x) = b_i(x) = 0, i \geq p$. С учетом этого можно записать

$$\begin{aligned} c(x) &= \sum_{k=0}^{2p-1} x^k \sum_{i=0}^{k} a_{k-i}(x^p)b_i(x^p) \bmod (x^n - 1) \\ = c(z,x)|_{z=x^p} &= \sum_{k=0}^{p-1} x^k \left(\sum_{i=0}^{k} a_{k-i}(z)b_i(z) + z \sum_{i=k+1}^{p-1} a_{k-i+p}(z)b_i(z) \right) \bmod (z^{n/p} - 1) \end{aligned}$$

Таким образом, задача свелась к вычислению n/p-точечных циклических сверток многочленов от z, которые могут быть вычислены с помощью быстрого алгоритма циклической свертки меньшей размерности. Заметим, что эти тождества справедливы в любом поле. Кроме того, выражение (3.4) представляет собой *линейную* свертку последовательностей многочленов $a_i(x^p)$ и $b_j(x^p)$, для вычисления которой могли быть использованы соответствующие быстрые алгоритмы.

Пример 3.9. Построим быстрый алгоритм вычисления четырехточечной циклической свертки над полем $GF(2^m)$.

$$\begin{aligned} c(x) = & \quad (a_0 + a_1x + a_2x^2 + a_3x^3)(b_0 + b_1x + b_2x^2 + b_3x^3) \bmod (x^4 - 1) \\ = & \quad ((a_0 + a_2x^2) + x(a_1 + a_3x^2))((b_0 + b_2x^2) + x(b_1 + b_3x^2)) \bmod (x^4 - 1) \\ = & \quad ((a_0 + a_2z)(b_0 + b_2z) + z(a_1 + a_3z)(b_1 + b_3z)) \\ & + x\left((a_1 + a_3z)(b_0 + b_2z) + (a_0 + a_2z)(b_1 + b_3z)\right) \bmod (z^2 - 1). \end{aligned}$$

Здесь была произведена замена переменной $z = x^2$. Последнее произведение можно было вычислить с помощью алгоритма Карацубы:

$$\begin{aligned} c(x) = & \quad ((a_0 + a_2z)(b_0 + b_2z) + z(a_1 + a_3z)(b_1 + b_3z)) \\ & + x \quad (((a_0 + a_1) + (a_2 + a_3)z)((b_0 + b_1) + (b_2 + b_3)z) \\ & \quad -(a_0 + a_2z)(b_0 + b_2z) - (a_1 + a_3z)(b_1 + b_3z)) \bmod (z^2 - 1) \end{aligned}$$

На первом этапе вычислений необходимо воспользоваться быстрым алгоритмом двухточечной циклической свертки над $GF(2)$ для нахождения $(a_0 + a_2z)(b_0 + b_2z) \bmod (z^2 - 1)$, $(a_1 + a_3z)(b_1 + b_3z) \bmod (z^2 - 1)$, $((a_0 + a_1) + (a_2 + a_3)z)((b_0 + b_1) + (b_2 + b_3)z) \bmod (z^2 - 1)$. Далее необходимо произвести обратную замену переменных $z = x^2$, т.е. вставить нули между компонентами полученных векторов, и сложить их, учитывая смещения. Окончательно это приводит к следующей билинейной форме

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \left(P \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot P \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \right), P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Используя другую форму быстрого алгоритма двухточечной циклической свертки $c_0 = a_0(b_0 + b_1) + (a_0 + a_1)b_1 + x(a_0(b_0 + b_1) + (a_0 + a_1)b_0)$, можно было бы получить другой алгоритм с 9 умножениями, в котором матрицы предсложений были бы разными [5]. При этом одна из этих матриц содержала бы большое число единиц. Как будет показано в дальнейшем, это свойство может быть очень полезно при построении других алгоритмов.

UNIVARIATEDIVISION(f, g)
1 $q \leftarrow 0; r \leftarrow f;$
2 **while** $(r \neq 0) \wedge (\mathrm{LT}(g) | \mathrm{LT}(r))$
3 **do** $q \leftarrow q + \mathrm{LT}(r)/\mathrm{LT}(g)$
4 $r \leftarrow r - (\mathrm{LT}(r)/\mathrm{LT}(g))g$
5 **return** (q, r)

Рис. 3.1: Алгоритм деления многочленов от одной переменной

В поле характеристики 2 можно предложить еще один способ вычисления циклической свертки длины $n = 2^\lambda$, близкий к алгоритму Карацубы. Пусть $a(x) = a_0(x) + x^m a_1(x), b(x) = b_0(x) + x^m b_1(x), \deg a_i(x) < m, \deg b_i(x) < m, m = 2^{\lambda-1}$. Рассмотрим вычисление $c(x) \equiv a(x)b(x) \bmod (x^m - 1)^2$. Пусть $u_0(x) = a_0(x) + a_1(x), u_1(x) = a_1(x), w_0(x) = b_0(x) + b_1(x), w_1(x) = b_1(x)$. Ясно, что $a(x) = u_0(x) + (x^m - 1)u_1(x), b(x) = w_0(x) + (x^m - 1)w_1(x)$. Тогда

$$a(x)b(x) \equiv u_0(x)w_0(x) + (x^m - 1)\left(u_0(x)w_1(x) + u_1(x)w_0(x)\right) \bmod (x^{2m} - 1)$$

Таким образом,

$$\begin{aligned} a(x)b(x) \bmod (x^{2m} - 1) = {} & (a_0(x) + a_1(x))(b_0(x) + b_1(x)) \bmod (x^{2m} - 1) + \\ & (x^m - 1)\left((a_0(x) + a_1(x))b_1(x) + a_1(x)(b_0(x) + b_1(x)) \bmod (x^m - 1)\right) \end{aligned}$$

Заметим, что первое слагаемое представляет собой линейную свертку, а второе - сумму двух циклических сверткок, умноженную на $x^m - 1$. Исходная задача свелась к трем подзадачам вдвое меньшей размерности.

3.3.7 Деление многочленов

Предположим, что даны многочлены $a(x), b(x) \in \mathbb{F}[x]$ степени n и $m < n$, соответственно, где $\mathbb{F}$ — некоторое поле. Рассмотрим задачу построения разложения

$$a(x) = q(x)b(x) + r(x), \deg r(x) < m, \tag{3.5}$$

т.е. Евклидова деления. Классический алгоритм деления “в столбик” представлен на рисунке 3.1. Здесь $\mathrm{LT}\, a$ обозначает старший член многочлена. Этот алгоритм требует $O((n - m + 1)m)$ операций. Оказывается, что можно значительно ускорить вычисление частного и остатка, воспользовавшись аналогом метода Ньютона.

Заменим в (3.5) x на $1/x$ и домножим это тождество на x^n. В результате получим

$$x^n a\left(\frac{1}{x}\right) = \left(x^{n-m} q\left(\frac{1}{x}\right)\right)\left(x^m b\left(\frac{1}{x}\right)\right) + x^{n-m+1}\left(x^{m-1} r\left(\frac{1}{x}\right)\right).$$

Пусть $\mathrm{rev}_k\, a = x^k a(1/x)$. Заметим, что $\mathrm{rev}_n\, a = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1}x + a_n$, т.е. многочлен с коэффициентами, записанными в обратном порядке. Таким образом, имеет место

$$\mathrm{rev}_n\, a = \mathrm{rev}_{n-m}\, q\, \mathrm{rev}_m\, b + x^{n-m+1}\, \mathrm{rev}_{m-1}\, r$$

или

$$\mathrm{rev}_n\, a \equiv \mathrm{rev}_{n-m}\, q\, \mathrm{rev}_m\, b \bmod x^{n-m+1}.$$

Если $b(x)$ отличен от нуля, то его старший коэффициент, равный свободному члену $\mathrm{rev}_m\, b$, также отличен от нуля, т.е. обратим. Следовательно, $\mathrm{rev}_m\, b$ взаимно прост с x^{n-m+1}, т.е. обратим по этому модулю, откуда

$$\mathrm{rev}_{n-m}\, q \equiv \mathrm{rev}_n\, a (\mathrm{rev}_m\, b)^{-1} \bmod x^{n-m+1}.$$

Евклидовы частное и остаток можно найти как $q = \mathrm{rev}_{n-m}\, \mathrm{rev}_{n-m}\, q$ и $r = a - qb$.

Пример 3.10. Пусть даны многочлены над $GF(7)$ $a(x) = 5x^5 + 4x^4 + 3x^3 + 2x^2 + x$ и $b(x) = x^2 + 2x + 3$. Тогда $\mathrm{rev}_5\, a = x^4 + 2x^3 + 3x^2 + 4x + 5$, $\mathrm{rev}_2\, b = 3x^2 + 2x + 1$. Заметим, что $(3x^2 + 2x + 1)(4x^3 + x^2 + 5x + 1) = 5x^5 + 4x^4 + 1 \equiv 1 \bmod x^4$. Отсюда $\mathrm{rev}_3\, q = (x^4+2x^3+3x^2+4x+5)(4x^3+x^2+5x+1) \equiv 6x^3+x+5 \bmod x^4$, т.е. $q = 5x^3+x^2+6$ и $r = a - qb = 3x + 3$

Таким образом, осталось решить проблему нахождения для произвольного $f(x) \in \mathbb{F}[x] : f(0) \neq 0$ и $l \in \mathbb{N}$ многочлена $g : fg \equiv 1 \bmod x^l$. Напомним, что решение уравнения $\phi(g) = 0$ методом Ньютона состоит в последовательном построении приближений $g_{i+1} = g_i - \frac{\phi(g_i)}{\phi'(g_i)}$. В данном случае уравнение имеет вид $\phi(g) = f - 1/g = 0$, откуда

$$g_{i+1} = g_i - \frac{f - 1/g_i}{1/g_i^2} = 2g_i - fg_i^2. \tag{3.6}$$

Теорема 3.3. *Пусть $f, g_0, g_1, \ldots \in \mathbb{F}[x]$, причем $f(0) \neq 0, g_0 = (f(0))^{-1}$ и $g_{i+1} = 2g_i - fg_i^2 \bmod x^{2^{i+1}}$. Тогда*

$$fg_i \equiv 1 \bmod x^{2^i}, i \geq 0.$$

Доказательство. При $i = 0$ утверждение очевидно. Предположим, что $1 - fg_i = sx^{2^i}$. Тогда $1 - fg_{i+1} = 1 - 2fg_i + f^2 g_i^2 = (1 - fg_i)^2 = s^2 x^{2^{i+1}} \equiv 0 \bmod x^{2^{i+1}}$. □

Для того, чтобы вычислить обратный элемент по модулю x^l, достаточно воспользоваться преобразованием (3.6) $\lceil \log_2 l \rceil$ раз. Заметим, что этот метод может использоваться и в полях характеристики 2.

Пример 3.11. Вычислим $(x^4 + x + 1)^{-1} \bmod x^7$ в поле $GF(2)$. Заметим, что правило (3.6) можно записать в данном случае как $g_{i+1} = fg_i^2$. Получим:

$$\begin{aligned}
g_0 &\equiv 1 && \bmod x \\
g_1 \equiv x^4 + x + 1 &\equiv x + 1 && \bmod x^2 \\
g_2 \equiv (x^4 + x + 1)(x + 1)^2 &\equiv x^3 + x^2 + x + 1 && \bmod x^4 \\
g_3 \equiv (x^4 + x + 1)(x^3 + x^2 + x + 1)^2 &\equiv x^7 + x^5 + x^3 + x^2 + x + 1 && \bmod x^8
\end{aligned}$$

Отсюда $(x^4 + x + 1)(x^5 + x^3 + x^2 + x + 1) \equiv 1 \bmod x^7$.

При вычислении $f^{-1} \bmod x^{2^r}$ на i-м шаге необходимо выполнить $M(2^{i-1})$ операций для нахождения g_{i-1}^2, $M(2^i)$ операций для вычисления fg_{i-1}^2 и еще 2^{i-1} операций вычитания для нахождения $2g_{i-1} - fg_{i-1}^2$. Таким образом, сложность одной итерации равна $M(2^i)+M(2^{i-1})+2^{i-1} \leq \frac{3}{2}M(2^i)+2^{i-1}$, где $M(n)$ — число операций, необходимых для перемножения многочленов степени $n-1$ (сложность линейной свертки). Общая сложность алгоритма равна

$$\sum_{i=1}^{r}\left(\frac{3}{2}M(2^i)+2^{i-1}\right) \leq \left(\frac{3}{2}M(2^r)+2^{r-1}\right)\sum_{i=1}^{r}2^{i-r} < 3M(2^r)+2^r = 3M(l)+l$$

Используя данный метод, алгоритм Евклидова деления (т.е. построения разложения $a = qb + r$) можно реализовать следующим образом:

1. Если $\deg a < \deg b$, возвратить $q = 0, r = a$.
2. Пусть $m = \deg a - \deg b$. Вычислить $(\mathrm{rev}_{\deg b}(b))^{-1} \bmod x^{m+1}$.
3. $q^* = \mathrm{rev}_{\deg a}\, a(\mathrm{rev}_{\deg b}(b))^{-1} \bmod x^{m+1}$
4. Возвратить $q = \mathrm{rev}_m\, q^*, r = a - bq$.

Видно, что при $\deg a = n + m, \deg b = n$ сложность данного алгоритма равна $4M(m) + M(n) + O(n)$.

3.3.8 Вычисление значений многочленов

Предположим, что задан некоторый многочлен $u(x) = u_{n-1}x^{n-1} + u_{n-2}x^{n-2} + \ldots + u_0$ и необходимо вычислить его значение в одной или нескольких точках. В том случае, когда подавляющее большинство коэффициентов равны нулю, вычисления могут выполняться непосредственно, причем могут быть использованы быстрые алгоритмы возведения в степень. Однако в общем случае намного более эффективным методом является схема Горнера:

$$u(x) = (\ldots((u_{n-1}x + u_{n-2})x + u_{n-3})x + \ldots)x + u_0 \tag{3.7}$$

Таким образом, требуется всего $n-1$ умножений и $n-1$ сложений. Заметим, что вычисление значения многочлена в точке x_0 эквивалентно нахождению остатка от деления $u(x) \bmod (x - x_0)$. В тех случаях, когда имеется возможность параллельного выполнения d инструкций (или d потоков инструкций), данное выражение может быть переписано как

$$u(x) = \sum_{j=0}^{d-1} x^j \sum_{i=0}^{\lfloor n/d \rfloor - 1} u_{di+j}x^{di} = \sum_{j=0}^{d-1} x^j \left(u_j + y(u_{j+d} + y(u_{j+2d} + \ldots + yu_{j+\lfloor n/d-1 \rfloor d})\ldots)\right), \tag{3.8}$$

что дает возможность разбить коэффициенты многочлена на d групп и обрабатывать их параллельно.

В некоторых случаях требуется вычислить значение многочлена одновременно в нескольких точках $x_0, x_1, \ldots, x_{k-1}$. Простейшим способом решения этой задачи является независимое (в т.ч. параллельное) применение схемы Горнера для каждого из x_i. Это требует $O(k(n-1))$ операций. Существует, однако, асимптотически более эффективный метод.

Для простоты изложения предположим, что $k = 2^l, l > 0$. Пусть $m_{00}(x) = \prod_{i=0}^{k-1}(x - x_i)$. Вычислим $U_{00}(x) \equiv u(x) \bmod m_{00}(x)$. Ясно, что $U_{00}(x_i) = u(x_i)$. Пусть далее $m_{10}(x) = \prod_{i=0}^{k/2-1}(x - x_i), m_{11}(x) = \prod_{i=k/2}^{k-1}(x - x_i)$. Тогда $U_{10}(x) \equiv U_{00}(x) \equiv u(x) \bmod m_{10}(x), U_{11}(x) \equiv U_{00}(x) \equiv u(x) \bmod m_{11}(x)$ и

$$u(x_i) = U_{1j}(x_i), j = \begin{cases} 0, & i = 0..k/2-1, \\ 1, & i = k/2..k-1 \end{cases}.$$

Разбиение множества точек x_i может быть рекурсивно продолжено до тех пор, пока сложность непосредственной реализации правил (3.7) или (3.8) не снизится до приемлемой величины. Для простоты изложения будем считать, что разбиение осуществляется до тех пор, пока множество $\{x_i\}$ не будет разбито на подмножества мощности 1. Тогда $u(x_i) \equiv u(x) \bmod (x - x_i), u(x_i) \equiv U$

Таким образом, алгоритм одновременного вычисления значений многочлена $u(x)$ в точках $x_0, \ldots, x_{k-1}$, или вычисления остатков от деления на $(x - x_i), i = 0..k-1$, имеет следующий вид:

1. Если $\deg u(x) = 0$, возвратить $u(x) = u + 0$.
2. Вычислить $m_{10}(x) = \prod_{i=0}^{k/2-1}(x - x_i)$, $m_{11}(x) = \prod_{i=k/2}^{k-1}(x - x_i)$.
3. Вычислить $U_{10}(x) \equiv u(x) \bmod m_{10}(x)$, $U_{11}(x) \equiv u(x) \bmod m_{11}(x)$.
4. Рекуррентно вычислить $U_{10}(x_i), i = 0..k/2-1$ и $U_{11}(x_i), i = k/2..k-1$.

Заметим, что многочлены $m_{ij}(x)$ образуют дерево, содержащее $r = 1 + \log_2 k$ ярусов. Пусть $m_{ri}(x) = (x - x_i)$. Тогда

$$m_{ij}(x) = m_{i+1,2j}(x)m_{i+1,2j+1}(x), i = 0..r-1, j = 0..2^i - 1. \tag{3.9}$$

Число операций, необходимой для нахождения многочленов $m_{ij}(x)$ для фиксированного i, равно $\sum_{j=0}^{2^i-1} M(d_{ij}) \leq M(\sum_{j=0}^{2^i-1} d_{ij}) = M(k)$, где $d_{ij} = \deg m_{ij}(x)$ и $M(n)$ обозначает число операций, необходимых для перемножения двух многочленов степени n. Заметим, что здесь могут использоваться любые быстрые алгоритмы. Отсюда получим, что сложность построения всего набора многочленов равна $M(k)\log_2 k$. Если точки x_i известны заранее, многочлены $m_{ij}(x)$ также могут быть построены заранее. Кроме того, в некоторых случаях имеется возможность сгруппировать точки таким образом, чтобы эти многочлены имели "вычислительно простой" вид, облегчающий деление на них.

Если $m_{ij}(x)$ уже построены, сложность вычисления значений многочлена в k точках $T(k)$ может быть выражена как

$$T(k) = 2T(k/2) + 2D(k/2), \tag{3.10}$$

где $D(k/2)$ — сложность вычисления остатка от деления многочлена степени k на многочлен степени $k/2$. Необходимо отметить, что шаги 2 и 3 описанного алгоритма могут выполняться независимо для $U_{10}(x)$ и $U_{11}(x)$, а потому могут быть реализованы параллельно. При этом их сложность меньше, чем при непосредственной реализации схемы Горнера для исходного многочлена.

В тех случаях, когда множество точек x_i обладает некоторой алгебраической структурой, сложность вычислений может быть снижена.

3.3.9 Интерполяция

Рассмотрим задачу построения по набору точек $(x_i, y_i), i = 0..n-1$ многочлена $f(x)$ наименьшей возможной степени, такого что $f(x_i) = y_i$. Как известно, такой многочлен единственен и имеет степень не более $n-1$. Наибольшее распространение получили интерполяционный полином (правильнее сказать формула) Лагранжа

$$f(x) = \sum_{i=0}^{n-1} y_i \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

Интерполяционный полином Лагранжа может быть найден следующим образом:

1. Пусть $m_i = (x - x_i)$. Вычислить $m_0 m_1, m_0 m_1 m_2, \ldots, m = m_0 \cdots m_{n-1}$.
2. Вычислить $n_i(x) = m(x)/m_i(x), i = 0..n-1$.
3. Вычислить $\beta_i = n_i(x_i), i = 0..n-1$
4. Вычислить $\sum_{i=0}^{n-1} \frac{y_i}{\beta_i} n_i(x)$

Первый шаг требует $\sum_{i=1}^{n-1}(2i-1) = n^2 - 2n + 1$ операций. Второй шаг требует $(2n-2)n$ операций, вычисление значений в $n_i(x_i)$ — $(2n-4)n$ операций. Вычисление y_i/β_i требует n операций. Вычисление линейной комбинации многочленов на последнем шаге требуеь $2n^2 - n$ операций. Таким образом, применение формулы Лагранжа требует $7n^2 - 8n + 1$ операций.

Несколько более эффективна с вычислительной точки зрения формула Ньютона

$$f(x) = [y_0] + [y_0; y_1](x - x_0) + [y_0; y_1; y_2](x - x_0)(x - x_1) + \ldots + [y_0; y_1; \ldots; y_{n-1}](x - x_0) \cdots (x - x_{n-2}),$$

где $[y_0; \ldots; y_k] = \frac{[y_1; \ldots; y_k] - [y_0; \ldots; y_{k-1}]}{x_k - x_0}, [y_i] = y_i$ — разделенные разности. Вычисление разделенных разностей требует $\sum_{i=1}^{n-1} 3(n-i) = \frac{3}{2}n(n-1)$ операций. Построение собственно многочлена может быть произведено по методу, аналогичному схеме Горнера

за $\sum_{i=1}^{n-1} 2i = n(n-1)$ операций. Таким образом, интерполяционный полином может быть построен за $\frac{5}{2}(n^2 - n)$ арифметических операций.

Интерполяция может быть реализована еще более эффективно путем использования стратегии "разделяй и властвуй". Заметим, что формальная производная $m'(x) = \sum_{i=0}^{n-1} \prod_{j \neq i}(x - x_j)$ и $\beta_i = m'(x_i)$. Пусть $z_i = \frac{y_i}{m'(x_i)}$ Построим дерево многочленов $\prod_i(x - x_i)$ по правилу (3.9). Заметим, что $m(x) = m_{00}(x)$ и

$$\sum_{i=0}^{n-1} z_i \frac{m(x)}{x - x_i} = \left(\sum_{i=0}^{n/2-1} z_i \frac{m_{10}(x)}{x - x_i} \right) m_{11}(x) + \left(\sum_{i=0}^{n/2-1} z_i \frac{m_{11}(x)}{x - x_i} \right) m_{10}(x).$$

Выражения в круглых скобках могут быть вычислены рекуррентно аналогичным образом. Таким образом, интерполяционный многочлен может быть построен с помощью следующего алгоритма:

1. Построить дерево многочленов $m_{ij}(x) = \prod_k (x - x_i)$ по правилу (3.9).
2. Построить $m'(x)$.
3. Вычислить $\beta_i = m'(x_i)$ с помощью алгоритма из раздела 3.3.8.
4. Вычислить $z_i = y_i/\beta_i, i = 0..n-1$.
5. $f(x) = Combine(0, ((x_i, z_i), i = 0..n-1))$.

Подпрограмма $Combine(l, ((x_i, z_i), i = 0..n-1))$ имеет следующий вид:

1. Если $n = 1$, возвратить z_0
2. $r_0(x) = Combine(l+1, ((x_i, z_i), i = 0..n/2-1))$
3. $r_1(x) = Combine(l+1, ((x_i, z_i), i = n/2..n-1))$
4. Возвратить $r_0(x)m_{l+1,1}(x) + r_1(x)m_{l+1,0}(x)$

Сложность последнего шага интерполяционного алгоритма может быть оценена как $T(n) \leq 2T(n/2) + M(n) + cn$ или $T(n) = O(M(n)\log n)$, где $M(n)$ — сложноть используемого алгоритма умножения. Сложность подготовительных шагов может быть оценена аналогичным выражением.

Лабораторная работа № 3. Реализация быстрого алгоритма свертки в виде линейной программы

В рамках данной работы необходимо построить быстрый алгоритм вычисления линейной или циклической свертки и реализовать его как непосредственно, так и с использованием SIMD-примитивов. Кроме того, необходимо реализовать программу вычисления свертки по определению. В пояснительной записке необходимо привести:

- Построение быстрого алгоритма;
- Итоговую билинейную форму;
- Последовательность арифметических операций, соответствующую полученной билинейной форме.
- Каноническую параллельную форму алгоритма;
- Время вычисления свертки с помощью трех программ.

Варианты заданий:

1. 10-точечная циклическая свертка над полем комплексных чисел с помощью алгоритма Агарвала-Кули;
2. 10-точечная циклическая свертка над полем комплексных чисел с помощью итерированного алгоритма;
3. 12-точечная циклическая свертка над полем комплексных чисел с помощью алгоритма Агарвала-Кули;
4. 12-точечная циклическая свертка над полем комплексных чисел с помощью итерированного алгоритма;
5. 5-точечная линейная свертка над полем комплексных чисел с помощью алгоритма Винограда;
6. 6-точечная линейная свертка над полем комплексных чисел с помощью итерированного алгоритма;
7. 10-точечная циклическая свертка над полем $GF(2^m)$ с помощью алгоритма Агарвала-Кули;
8. 12-точечная циклическая свертка над полем $GF(2^m)$ с помощью алгоритма Агарвала-Кули;
9. 10-точечная циклическая свертка над полем $GF(2^m)$ с помощью итерированного алгоритма;
10. 12-точечная циклическая свертка над полем $GF(2^m)$ с помощью итерированного алгоритма;
11. 5-точечная линейная свертка над полем $GF(2^m)$ с помощью алгоритма Винограда;
12. 6-точечная линейная свертка над полем $GF(2^m)$ с помощью итерированного алгоритма.

Лабораторная работа № 4.
Реализация итерированного быстрого алгоритма свертки

В рамках данной работы необходимо воспользоваться разработанной в предыдущей работе линейной подпрограммой для построения программы вычисления линейных и циклических сверток большей размерности. В пояснительной записке необходимо привести описание используемого алгоритма, его каноническую параллельную форму алгоритма, теоретическую оценку его сложности и сравнение времени вычисления свертки по определению и с использованием быстрого алгоритма для различных значений k. Варианты заданий:

1. $2^k \cdot 10$-точечная циклическая свертка над полем комплексных чисел;
2. $2^k \cdot 10$-точечная циклическая свертка над полем комплексных чисел;
3. $2^k \cdot 12$-точечная циклическая свертка над полем комплексных чисел;
4. $2^k \cdot 12$-точечная циклическая свертка над полем комплексных чисел;
5. $2^k \cdot 5$-точечная линейная свертка над полем комплексных чисел;
6. $2^k \cdot 6$-точечная линейная свертка над полем комплексных чисел;
7. $2^k \cdot 10$-точечная циклическая свертка над полем $GF(2^m)$;
8. $2^k \cdot 12$-точечная циклическая свертка над полем $GF(2^m)$;
9. $2^k \cdot 10$-точечная циклическая свертка над полем $GF(2^m)$;
10. $2^k \cdot 12$-точечная циклическая свертка над полем $GF(2^m)$;
11. $2^k \cdot 5$-точечная линейная свертка над полем $GF(2^m)$;
12. $2^k \cdot 6$-точечная линейная свертка над полем $GF(2^m)$.

Лабораторная работа № 5.
Распараллеливание быстрого алгоритма свертки

В рамках данной работы необходимо распараллелить разработанную ранее программу вычисления линейной или циклической свертки средствами OpenMP. В пояснительной записке необходимо привести

1. описание используемого алгоритма;
2. его каноническую параллельную форму;
3. теоретические оценки достижимого ускорения;
4. сравнение времени вычисления свертки с помощью последовательной и параллельных реализаций быстрого алгоритма.

3.4 Дискретное преобразование Фурье

3.4.1 Преобразование Фурье в дискретном и непрерывном случаях

Предположим, что задана функция $f(x)$, периодическая на интервале $(-\pi, \pi)$. Такая функция может быть разложена в ряд Фурье

$$f(x) = \sum_{k=-\infty}^{\infty} A_k e^{ikx},$$

где

$$A_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-ikt} \mathrm{d}t.$$

Ясно, что это определение можно легко обобщить на случай других значений L периода функции. В пределе, при $L \to \infty$ ряд Фурье переходит в *преобразование Фурье*:

$$f(x) = \int_{-\infty}^{\infty} F(t) e^{2\pi i t x} \mathrm{d}t$$

(обратное преобразование Фурье),

$$F(t) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i t x} \mathrm{d}x$$

(прямое преобразование Фурье). Иногда его также определяют как

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{i\omega t} \mathrm{d}\omega$$

и

$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{-i\omega t} \mathrm{d}t.$$

Предположим, что функция $f(t)$ дискретна и задана своими отсчетами $f_k = f(k\Delta t), k = 0..n-1$. Это приводит к следующему определению *дискретного преобразования Фурье* последовательности f_k:

$$F_j = \sum_{k=0}^{n-1} f_k e^{-2\pi i j k/n}. \tag{3.11}$$

Обратное дискретное преобразование Фурье задается выражением

$$f_k = \frac{1}{n} \sum_{j=0}^{n-1} F_n e^{2\pi i k j/n} \tag{3.12}$$

Как обычно, множитель $1/n$, а также -1 в показателе степени может перераспределяться между этими выражениями. Пусть $\alpha = e^{-2\pi i/n}$, или, в более общем случае, α — некоторый элемент порядка n. Тогда

$$\begin{aligned} F_j &= \sum_{k=0}^{n-1} f_k \alpha^{jk} \\ f_k &= \frac{1}{n} \sum_{j=0}^{n-1} F_j \alpha^{-jk} \end{aligned}$$

Величина α называется *ядром преобразования*. Чтобы убедиться в справедливости этих соотношений, заметим, что элемент порядка n должен удовлетворять равенству $\alpha^n - 1 = 0$, а также $\alpha^{rn} - 1 = 0$. Над любым полем $x^n - 1 = (x-1)(x^{n-1} + \ldots + 1) = 0$. Следовательно, для всех $r \neq 0 \bmod n$ элемент α^r является корнем последнего многочлена в этом разложении. Это означает, что

$$\sum_{j=0}^{n-1} \alpha^{rj} = 0, r \neq 0 \bmod n.$$

Если $r = 0 \bmod n$, то

$$\sum_{j=0}^{n-1} \alpha^{rj} = n.$$

Это не равно нулю, если n не кратно характеристике поля. Таким образом,

$$\sum_{j=0}^{n-1} \alpha^{-jk} \sum_{l=0}^{n-1} \alpha^{lj} f_l = \sum_{l=0}^{n-1} f_l \sum_{j=0}^{n-1} \alpha^{(l-k)j} = n f_k.$$

Вычисление ДПФ можно рассматривать как умножение вектора $(f_0, \ldots, f_{n-1})^T$ на *матрицу Вандермонда*, которая, как известно, невырождена.

В поле комплексных чисел ДПФ существует для всех $n > 0$. В конечных полях $GF(p^m)$ это не так. Действительно, ненулевые элементы поля образуют группу по умножению, а все степени α (не путать α с примитивным элементом поля!) образуют некоторую циклическую конечную подгруппу. В соответствии с теоремой Лагранжа, порядок этой подгруппы должен быть делителем порядка исходной группы, т.е. $p^m - 1$. Например, в поле $GF(2^4)$ существуют ДПФ длины $3, 5, 15$, но не существует ДПФ длины 8. Если ДПФ нужной длины не существует, можно взять некоторое расширение поля. Однако во многих случаях это крайне непрактично.

Необходимо отметить, что прямое и обратное преобразование Фурье симметричны. Действительно, $F_{n-j} = \sum_{k=0}^{n-1} f_k \alpha^{(n-j)k} = \sum_{k=0}^{n-1} f_k \alpha^{-jk}$. Кроме того, можно заметить, что F_j равны значению многочлена $f(x) = \sum_{k=0}^{n-1} f_k x^k$ в точках α^j. Следовательно, ОДПФ соответствует операции интерполяции. Это свойство используется для поиска корней многочленов над конечными полями. Действительно, если F_i (i-я спектральная компонента) оказывается равным нулю, то $f(\alpha^i) = 0$ т.е. α^i является корнем.

Теорема 3.4 (О свертке). *Пусть существует ДПФ длины n и*

$$e_i = f_i g_i, i = 0..n-1.$$

Тогда

$$E_j = \frac{1}{n}\sum_{k=0}^{n-1} F_{((j-k))} G_k.$$

Доказательство. Вычислим ДПФ вектора с компонентами $e_i = f_i g_i$:

$$E_j = \sum_{i=0}^{n-1} \alpha^{ij} f_i g_i = \sum_{i=0}^{n-1} \alpha^{ij} f_i \frac{1}{n} \sum_{k=0}^{n-1} \alpha^{-ik} G_k = \frac{1}{n} \sum_{k=0}^{n-1} G_k \left(\sum_{i=0}^{n-1} \alpha^{i(j-k)} f_i \right) = \frac{1}{n} \sum_{k=0}^{n-1} G_k F_{((j-k))}$$

□

Это тождество может быть использовано для быстрого вычисления циклической свертки.

ДПФ применяется практически в огромном числе вычислительных процедур, например:

- Цифровая обработка сигналов;
- Цифровая связь (ADSL, WiMAX,...)
- Сжатие и обработка изображений
- Помехоустойчивое кодирование

Непосредственное вычисление ДПФ требует n^2 умножений. Но за счет использования алгебраических свойств элемента α и поля, в котором происходят вычисления, сложность удается существенно сократить. Различные алгоритмы быстрого вычисления ДПФ получили название *быстрого преобразования Фурье* (БПФ). Некоторые из этих приемов реализованы аппаратно в цифровых сигнальных процессорах. Тем не менее, во многих случаях оказывается необходимым вычисление ДПФ на длинах, не поддерживаемых аппаратно. В этом случае возникает задача необходимо свести к набору вычислительных примитивов, поддерживаемых аппаратно.

3.4.2 Общие алгоритмы быстрого преобразования Фурье

Алгоритм Кули-Тьюки

Предположим, что необходимо вычислить ДПФ длины $n = n'n''$. Пусть $i = i' + n'i'', k = n''k' + k''$. Тогда

$$F_k = F_{n''k'+k''} = \sum_{i=0}^{n-1} f_i \alpha^{ki} = \sum_{i''=0}^{n''-1} \sum_{i'=0}^{n'-1} \alpha^{(n''k'+k'')(i'+n'i'')} f_{i'+n'i''}, k' = 0..n'-1, k'' = 0..n''-1.$$

Пусть $\gamma = \alpha^{n'}, \beta = \alpha^{n''}$. Заметим, что $\forall k', i'' : \alpha^{n'n''k'i''} = 1$. Пусть $f_{i',i''} = f_{i'+n'i''}, F_{k',k''} = F_{n''k'+k''}$. Тогда

$$F_{k',k''} = \sum_{i'=0}^{n'-1} \beta^{k'i'} \left(\alpha^{i'k''} \sum_{i''=0}^{n''-1} \gamma^{i''k''} f_{i',i''} \right)$$

Заметим, что выражение во внутренних скобках зависит только от i', k'' и представляет собой набор ДПФ с ядром γ, результат которых после нескольких дополнительных умножений передается на вход другим ДПФ с ядром β. Таким образом, ДПФ может быть вычислено следующим образом (*алгоритм Кули-Тьюки*):

1. Вычислить $\sum_{i''=0}^{n''-1} \gamma^{i''k''} f_{i',i''}, i' = 0..n'-1, k'' = 0..n''-1$. Непосредственное выполнение этого шага требует $n'n''n''$ умножений и $n'n''(n''-1)$ сложений.

2. Домножить результат предыдущего шага на $\alpha^{i'k''}, i' = 0..n'-1, k'' = 0..n''-1$. Это требует $n'n''$ умножений. Пусть $t_{i'k''}$ — результат этого шага.

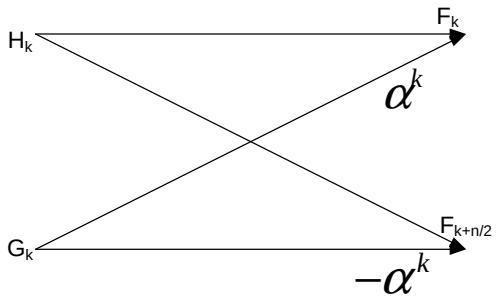
3. Вычислить $F_{k',k''} = \sum_{i'=0}^{n'-1} \beta^{k'i'} t_{i'k''}, k' = 0..n'-1$. Аналогично, этот шаг требует $n'n''n'$ умножений и $n'n''(n'-1)$ сложений.

Таким образом, общая сложность вычислений составляет $M(n) = n(n''+n'+1)$ умножений и $A(n) = n(n''+n'-2)$ сложений. В некоторых случаях величины $\beta^{k'i'}$ имеют специальный вид, облегчающий вычисления. Дальнейшее снижение сложности может быть получено за счет использования для вычисления ДПФ малой длины других быстрых алгоритмов, необязательно Кули-Тьюки. Тогда $M(n) = n'M(n'') + n''M(n') + n, A(n) = n'A(n'') + n''A(n')$.

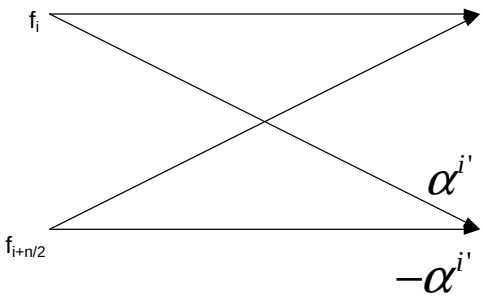
Во многих приложениях длина преобразования над полем комплексных чисел оказывается равной $n = 2^m$ (БПФ по основанию 2). В этом случае $n' = 2, n'' = 2^{m-1}$ (прореживание по времени) или $n' = 2^{m-1}, n'' = 2$ (прореживание по частоте). В первом случае $\beta = \alpha^{n/2} = -1$, т.е.

$$\begin{aligned} F_k &= \sum_{i=0}^{n/2-1} \alpha^{2ik} f_{2i} + \alpha^k \sum_{i=0}^{n/2-1} \alpha^{2ik} f_{2i+1} \\ F_{k+n/2} &= \sum_{i=0}^{n/2-1} \alpha^{2ik} f_{2i} - \alpha^k \sum_{i=0}^{n/2-1} \alpha^{2ik} f_{2i+1}, k = 0..n/2-1 \end{aligned}$$

Таким образом, входной вектор распадается на два подвектора, элементы которых записаны через одну позицию. Если процедура БПФ для этих векторов длины $n/2$ сохраняет результат во входном массиве, то и рассматриваемая процедура БПФ длины n может сохранить результат во входном векторе, воспользовавшись всего одной временной переменной. Действительно, если H_k и G_k — k-ые компоненты преобразований Фурье четных и нечетных компонент исходного вектора, соответственно, то $F_k = H_k + \alpha^k G_k, F_{k+n/2} = H_k - \alpha^k G_k$. При этом, однако, порядок элементов в выходном векторе будет отличаться от требуемого.



(a) Бабочка Кули-Тьюки



(b) Бабочка Джентльмена-Сенда

Рис. 3.2: Преобразование бабочки

При использовании прореживания по частоте имеем $\gamma = \alpha^{n'} = \alpha^{n/2} = -1$ и

$$F_{2k'} = \sum_{i'=0}^{n/2-1} \beta^{k'i'} \left(f_{i'} + f_{n/2+i'}\right)$$

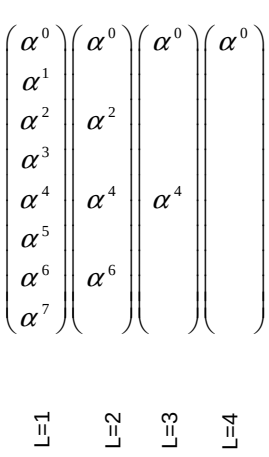
$$F_{2k'+1} = \sum_{i'=0}^{n/2-1} \beta^{k'i'} \left(\alpha^{i'}(f_{i'} - f_{n/2+i'})\right), k' = 0..n/2-1$$

Обе рассмотренные реализации алгоритма Кули-Тьюки используют различные модификации преобразования бабочки, представленного на рисунке 3.2. На рисунке 3.3 приведена одна из возможных реализаций алгоритма Кули-Тьюки с прореживанием по частоте [14]. Параметрами данной процедуры являются f, преобразуемый массив, n, его длина, а также массив w, содержащий коэффициенты $\alpha^i, i = 0..n/2-1$. На каждой итерации внешнего цикла WHILE исходная задача разбивается на две подзадачи меньшей размерности. Обработка каждой из них включает в себя применение преобразования бабочки. Т.к. результаты этого преобразования сохраняются в том же векторе, что и исходные данные, эти действия эквивалентны рекурсивному обращению к процедуре БПФ. Для вычисления преобразования бабочки используется вектор заранее вычисленных коэффициентов $w = (\alpha^i, i = 0..n/2-1)$. Необходимо отметить, что обращения к этому массиву производятся с равномерным шагом, равным текущему числу задач БПФ в разбиении (L), как показано на рисунке 3.4(a). Эта величина монотонно возрастает, что может привести к увеличению числа кеш-промахов. В связи с этим может быть более эффективным использование массива большего размера, содержащего коэффициенты, используемые на каждом шаге разбиения, как показано на рисунке 3.4(b). Пусть $k = \sum_{i=0}^{l-1} k_i 2^i, k_i \in \{0,1\}, l = \log_2(n)$. Будем обозначать элементы массивов их двоичными индексами, выписанными начиная с наименее значащего бита, т.е. $A_k = A_{k_0,k_1,\ldots,k_{l-1}}$

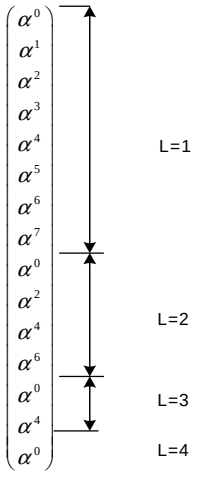
Теорема 3.5. *После завершения работы процедуры DIFCooleyTukey вектор f содержит последовательность $F_{k_{l-1},k_{l-2},\ldots,k_0}, k = 0..n-1$.*

```
DIFCooleyTukey(f, n, w)
 1  L ← 1
 2  N ← n
 3  while N > 1
 4  do N' ← N/2
 5     for K ← 0 to L − 1
 6     do J_w ← 0
 7        for J ← K · N to K · N + N' − 1
 8        do W ← w[J_w]
 9           T ← f[J]
10           f[J] ← T + f[J + N']
11           f[J + N'] ← W · (T − f[J + N'])
12           J_w ← J_w + L
13     L ← 2 · L
14     N ← N'
```

Рис. 3.3: Алгоритм БПФ вектора длины 2^m с прореживанием по частоте



(a) Непосредственная реализация



(b) Реализация с минимизацией кеш-промахов

Рис. 3.4: Элементы массива коэффициентов, используемые на различных уровнях разбиения

Доказательство. Для случая $n = 2$ утверждение теоремы проверяется непосредственно. Предположим, что оно верно для $n = 2^{m-1}, m > 1$ и рассмотрим случай $n = 2^m$. Заметим, что первая итерация внешнего цикла алгоритма состоит в вычислении вектора $(\underbrace{f_0 + f_{n/2+0}, \ldots, f_{n/2-1} + f_{n-1}}_{1}, \underbrace{\alpha^0(f_{n/2-1} - f_{n-1}), \ldots, \alpha^{n/2-1}(f_{n/2-1} - f_{n-1})}_{2})$. Далее одни и те же действия выполняются как для подвектора 1, так и для подвектора 2, что соответствует ДПФ размерности $n/2$ для четных и нечетных компонент вектора F. Согласно индукционному предположению, результат этих ДПФ записывется в последовательности $F_{k_{l-1},k_{l-2},\ldots,k_1}, k = 0..n/2 - 1$. Но результат ДПФ размерности n должен быть получен чередованием результатов ДПФ размерности $n/2$. Таким образом, после завершения работы этой процедуры в массиве f оказывается последовательность $F_{k_{l-1},k_{l-2},\ldots,k_1,k_0}, k = 0..n - 1$. □

Таким образом, возможно вычисление БПФ без использования дополнительных массивов для хранения промежуточных результатов. Однако при этом результат оказывается записан в неестественном порядке (с обращенной последовательностью битов). Существуют алгоритмы, позволяющие переупорядочить результат вычислений без использования дополнительных массивов. Аналогичные методы применимы и к БПФ с прореживанием по времени.

Сложность описанных алгоритмов удовлетворяет рекуррентным соотношениям $M(n) = 2M(n/2) + n/2 = O((n/2)\log_2 n), A(n) = 2A(n/2) + n = n\log_2 n$. Сложность алгоритма можно немного уменьшить, если заметить, что на самом внутреннем шаге алгоритма все умножения производятся на ± 1; на предпоследнем шаге умножения производятся на константы $i^k, k = 0..3$. Некоторые упрощения возможны и на последующих шагах. Популярны также алгоритмы БПФ по основанию 4.

Как правило, БПФ малых длин реализуют в виде линейных программ, в которых выполняются все возможные упрощения. Это позволяет экономить на издержках, связанных с организацией циклов и т.п. управляющими конструкциями. Кроме того, линейные программы лучше поддаются платформо-зависимой оптимизации, которая позволяет максимально эффективно использовать ресурсы процессора. Качественная реализация преобразований малых длин является необходимым условием надежной и эффективной работы преобразований больших длин.

Алгоритм Кули-Тьюки был предложен в 1805 году Гауссом и переоткрыт в 1965 году Кули (Cooley) и Тьюки (Tukey),

Алгоритм Гуда-Томаса

Предположим, что $n = n'n'', (n', n'') = 1$. Пусть $N'n' + N''n'' = 1$. Представим числа $i = 0..n - 1$ в виде пары вычетов по модулям n', n'':

$$i' \equiv i \bmod n', i'' \equiv i \bmod n'', i \equiv i'N''n'' + i''N'n' \bmod n.$$

Выходные индексы представим так:

$$k' = N''k \bmod n', k'' = N'k \bmod n'', k = n''k' + n'k'' \bmod n.$$

Действительно, $n''(N''k + Q'n') + n'(N'k + Q''n'') = k(N'n' + N''n'') \equiv k \bmod n$. Тогда

$$F_k = F_{k',k''} = \sum_{i'=0}^{n'-1} \sum_{i''=0}^{n''-1} \alpha^{(n''k'+n'k'')(i'N''n''+i''N'n')} f_{i'N''n''+i''N'n'} = \sum_{i'=0}^{n'-1} \beta^{i'k'} \left(\sum_{i''=0}^{n''-1} \gamma^{i''k''} f_{i',i''} \right),$$

где $\beta = \alpha^{N''(n'')^2}, \gamma = \alpha^{N'(n')^2}$. *Алгоритм Гуда-Томаса* требует $n'n''n'' + n''n'n' = n(n' + n'')$ умножений. Число сложений совпадает с алгоритмом Кули-Тьюки.

Требование взаимной простоты сомножителей ограничивает возможность построения рекурсивных алгоритмов. В связи с этим алгоритм Гуда-Томаса обычно комбинируют с другими алгоритмами.

Алгоритм Герцеля

В некоторых случаях требуется вычислить небольшое количество элементов вектора ДПФ. Пусть необходимо найти (возможно, среди прочих) компоненту $F_k = \sum_{i=0}^{n-1} \alpha^{ik} f_i = f(\alpha^k)$. Построим многочлен $\phi(x)$ минимальной степени с "вычислительно простыми" коэффициентами, имеющий α^k своим корнем. В случае комплексного ДПФ этот многочлен имеет вид $\phi(x) = (x - \alpha^k)(x - \alpha^{-k}) = x^2 - (2\cos(2\pi k/n))x + 1$ (вещественные коэффициенты!). В случае конечных полей воспользуемся минимальным многочленом α^k. Пусть $f(x) = Q(x)\phi(x) + r(x)$. Ясно, что $r(\alpha^k) = f(\alpha^k)$. Вычисление остатка от деления можно реализовать с помощью авторегрессионного фильтра (см. рис. 3.5). Фильтр работает следующим образом. Изначально обе его ячейки, обозначенные прямоугольниками, инициализируются нулевыми значениями. Далее на вход фильтра подаются значения $f_{n-1}, f_{n-2}, \ldots, f_0$, а содержимое ячеек сдвигается вправо с выполнением соответствующих арифметических операций. После выполнения последнего шага в ячейках останутся коэффициенты многочлена $r(x)$. Можно заметить, что преобразования, реализуемые этим фильтром, в точности повторяют действия, выполняемые при делении многочленов "в столбик" (см. рис. 3.1).

Оценим сложность для случая вычислений в комплексном поле. Если бы в качестве многочлена $\phi(x)$ был взят полином с комплексными коэффициентами, деление (т.е. вычисление по схеме Горнера) потребовало бы $2(n-2)$ вещественных умножения. За счет использования многочлена с вещественными коэффициентами сложность уменьшается до $n - 2$ умножений.

В случае конечных полей экономия может быть намного более существенной, т.к. умножения в простом поле существенно проще умножений в расширенном поле.

Преобразование Фурье как свертка

Рассмотрим *алгоритм Блюстейна*. Воспользуемся тождеством $ik = \frac{1}{2}(i^2 + k^2 - (i-k)^2)$. Тогда $F_k = \sum_{i=0}^{n-1} \alpha^{ik} f_i = \alpha^{k^2/2} \sum_{i=0}^{n-1} \alpha^{i^2/2} f_i \alpha^{-(i-k)^2/2}$. Произведем замену переменных $G_k = \alpha^{-k^2/2} F_k, g_i = \alpha^{i^2/2} f_i, h_{i-k} = \alpha^{-(i-k)^2/2}$. Тогда задача

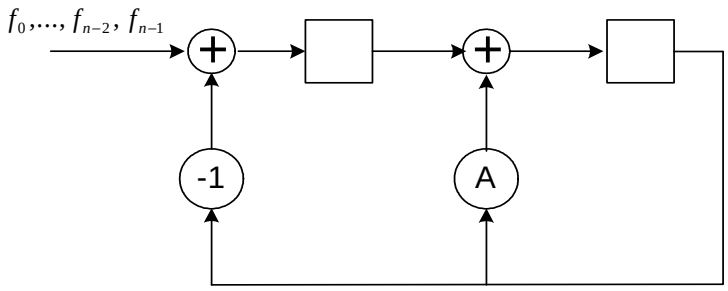


Рис. 3.5: Авторегрессионный фильтр с передаточной функцией $x^2 - Ax + 1$

вычисления ДПФ сводится к вычислению

$$\begin{pmatrix} G_0 \\ G_1 \\ \vdots \\ G_{N-1} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & \ldots & h_{N-1} \\ h_1 & h_0 & h_1 & \ldots & h_{N-2} \\ \cdots & & & & \\ h_{N-1} & h_{N-2} & h_{N-3} & \ldots & h_0 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{N-1} \end{pmatrix}$$

Полученная матрица носит название *теплицевой* (все ее диагонали содержат одинаковые элементы). Пока это привело лишь к увеличению сложности. Однако полученная $N \times N$ матрица может быть расширена до $2N - 2$ *циркулянтной* (т.е. такой матрицы, все строки которой являются циклическими сдвигами друг друга). Например, 6×6 матрица может быть преобразована следующим образом:

$$\begin{pmatrix} h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \\ h_1 & h_0 & h_1 & h_2 & h_3 & h_4 \\ h_2 & h_1 & h_0 & h_1 & h_2 & h_3 \\ h_3 & h_2 & h_1 & h_0 & h_1 & h_2 \\ h_4 & h_3 & h_2 & h_1 & h_0 & h_1 \\ h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \end{pmatrix} \longrightarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & h_4 & h_3 & h_2 & h_1 \\ h_1 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & h_4 & h_3 & h_2 \\ h_2 & h_1 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & h_4 & h_3 \\ h_3 & h_2 & h_1 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & h_4 \\ h_4 & h_3 & h_2 & h_1 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \\ h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & h_1 & h_2 & h_3 & h_4 \\ h_4 & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & h_1 & h_2 & h_3 \\ h_3 & h_4 & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & h_1 & h_2 \\ h_2 & h_3 & h_4 & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & h_1 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \end{pmatrix}$$

Умножение на такую матрицу эквиваленетно вычислению циклической свертки. Таким образом, задача вычисления ДПФ свелась к вычислению циклической свертки дополненного нулями преобразованного входного вектора g с фиксированной последовательностью h, отбрасыванию лишних компонентов полученного вектора и делению их на соответствующие константы.

БПФ простой длины

Алгоритм Рейдера также позволяет свести задачу вычисления ДПФ простой длины над некоторым полем $\mathbb{F}$ к циклической свертке. Пусть $F_k = \sum_{i=0}^{n-1} \alpha^{ik} f_i$. Тогда

$$\begin{aligned} F_0 &= \sum_{i=0}^{n-1} f_i \\ F_k &= f_0 + \sum_{i=1}^{n-1} \alpha^{ik} f_i \end{aligned}$$

Если n — простое число, то существует поле $GF(n)$ с примитивным элементом π. Введем функцию $r(i) : \pi^{r(i)} = i$ (аналог логарифма). Тогда $F_{\pi^{r(k)}} = f_0 + \sum_{i=1}^{n-1} \alpha^{\pi^{r(i)+r(k)}} f_{\pi^{r(i)}}$ Сделаем замену переменных $l = r(k), j = n - 1 - r(i)$. Тогда

$$F_{\pi^l} = f_0 + \sum_{j=1}^{n-1} \alpha^{\pi^{l-j}} f_{\pi^{n-1-j}} \tag{3.13}$$

Таким образом, задача вычисления ДПФ свелась к задаче вычисления циклической свертки переставленной входной последовательности и фиксированной последовательности. Применяя быстрый алгоритм Винограда вычисления циклической свертки, получим *алгоритм Винограда БПФ*. Существенным недостатком алгоритма Винограда является большое число нерегулярных сложений, что затрудняет векторизацию вычислений (SIMD).

С другой стороны, для вычисления циклической свертки может использоваться алгоритм БПФ длины $n-1$, которое не является простым числом, т.е. для его вычисления могут быть использованы другие алгоритмы БПФ.

3.4.3 Алгоритмы БПФ в конечных полях

В конечных полях могут быть применены все вышеописанные общие алгоритмы БПФ. Однако применение некоторых специфических свойств конечных полей позволяет существенно снизить сложность вычислений. Как правило, на практике приходится комбинировать специализированные и общие алгоритмы.

Некоторые свойства конечных полей

Определение 3.1. [11] *Линеаризованными многочленами* над $GF(p^m)$ называются многочлены вида

$$L(x) = \sum_i l_i x^{p^i}, l_i \in GF(p^m)$$

В том случае, когда $l_i \in GF(p)$, $L(x)$ называется p-полиномом.

Несложно показать, что для линеаризованных многочленов выполняется

$$L(a+b) = L(a) + L(b),\ a, b \in GF(p^m). \tag{3.14}$$

Из этого свойства вытекает следующая теорема, представленная здесь в немного модифицированном виде:

Теорема 3.6 ([4]). *Пусть* $y \in GF(p^m)$ *и элементы* $\beta_0, \beta_1, ..., \beta_{m-1}$ *образуют некоторый базис этого поля. Если*

$$y = \sum_{i=0}^{m-1} y_i \beta_i, y_i \in GF(p),$$

то

$$L(y) = \sum_{i=0}^{m-1} y_i L(\beta_i).$$

Таким образом, линеаризованный многочлен однозначно задается своими значениями в базисных элементах поля. Т.к. множество его значений входит в $GF(p^m)$, вектор этих значений может рассматриваться также как матрица $\mathcal{L}$ над $GF(p)$. Это дает возможность находить корни линеаризованных многочленов путем решения однородной системы линейных уравнений $y\mathcal{L} = 0$. Отсюда также следует то, что корни линеаризованных многочленов образуют линейное подпространство в $GF(p)^m$.

Обобщением линеаризованных многочленов являются *аффинные многочлены*, определяемые как $A(x) = a + L(x)$, где $a \in GF(p^m)$ и $L(x)$ — линеаризованный многочлен.

Определение 3.2. Следом элемента $\delta \in GF(p^m)$ называется величина

$$\mathrm{Tr}(\delta) = \sum_{i=0}^{m-1} \delta^{p^i}$$

Заметим, что $\mathrm{Tr}(\delta)^p = \sum_{i=0}^{m-1} \delta^{p^{i+1}} = \sum_{i=1}^{m} \delta^{p^i} = \sum_{i=0}^{m-1} \delta^{p^i} = \mathrm{Tr}(\delta) \Rightarrow \mathrm{Tr}(\delta) \in GF(p)$.

Если β — примитивный элемент конечного поля $GF(p^m)$, то всякий его элемент может быть представлен как $x = \sum_{i=0}^{m-1} x_i \beta^i, x_i \in GF(p)$. Множество $\{\beta^0, \ldots, \beta^{m-1}\}$ называется *стандартным базисом* конечного поля. Т.к. поле $GF(p^m)$ может рассматриваться как векторное пространство $GF(p)^m$, существуют и другие базисы, причем для любого базиса $(\pi_0, \ldots, \pi_{m-1})$ справедливо

$$\begin{pmatrix} 1 \\ \beta \\ \vdots \\ \beta^{m-1} \end{pmatrix} = C \begin{pmatrix} \pi_0 \\ \pi_1 \\ \vdots \\ \pi_{m-1} \end{pmatrix}.$$

Часто бывает полезен *нормальный базис* $\{\gamma^{p^0}, \gamma^{p^1}, \ldots, \gamma^{p^{m-1}}\}$. Можно показать, что в любом конечном поле существует нормальный базис. Например, в поле $GF(2^2)$

нормальный базис совпадает со стандартным $\{\beta, \beta^2\}$, где β — примитивный элемент. В поле $GF(2^4)$, задаваемом многочленом $x^4 + x + 1$, нормальный базис образован элементами $\beta^3, \beta^6, \beta^{12}, \beta^9$. Ясно, что образующий элемент γ нормального базиса должен удовлетворять $\mathrm{Tr}(\delta) \neq 0$.

Пусть $(n, p) = 1$. Рассмотрим многочлен $x^n - 1$. Его корни лежат в некотором расширенном поле $GF(p^m)$. Вычислим формальную производную многочлена $x^n - 1$. Она равна nx^{n-1}. Т.к. $(n, p) = 1$, многочлены nx^{n-1} и $x^n - 1$ взаимно просты, т.е. $x^n - 1$ не имеет кратных корней. Все его корни $\alpha_0, \ldots, \alpha^{n-1}$ называются корнями n-й степени из единицы.

Из вышесказанного следует, что если необходимо вычислить ДПФ вектора $(f_0, \ldots, f_{n-1}), f_i \in GF(p^l)$ и $n \nmid p^l - 1$, то корни $x^n - 1$ лежат в некотором другом поле $GF(p^m)$, а следовательно, вычисления надо производить в некотором третьем расширенном поле $GF(p^{lcm(l,m)})$, содержащем как поле элементов вектора, так и корни из единицы. Результат ДПФ также будет принадлежать этому полю. Вычисления в больших расширениях могут быть весьма трудоемки, поэтому таких ситуаций следует по возможности избегать. Кроме того, стандартное определение ДПФ неприменимо в случае $p|n$.

Циклотомическим классом по модулю n над $GF(p)$ называется множество

$$C_s = \{s, sp, sp^2, \ldots, sp^{m_s - 1}\}, sp^{m_s} \equiv s \bmod n.$$

Ясно, что множество целых чисел $o \leq i \leq n - 1$ разбивается на набор циклотомических классов, т.е.

$$\{0, \ldots, n - 1\} = \bigcup_s C_s,$$

где s пробегает множество представителей циклотомических классов. Заметим, что циклотомические классы являются классами эквивалентности по бинарному отношению $\rho = \{(x, y) | \exists l : x \equiv yp^l \bmod n\}$.

Пример 3.12. Циклотомическими классами по модулю 9 над $GF(2)$ являются $\{0\}, \{1, 2, 4, 8, 7, 5\}, \{3, 6\}$.

Минимальным многочленом элемента α^s является многочлен с коэффициентами из $GF(p)$, равный

$$M^{(s)}(x) = \prod_{i \in C_s} (x - \alpha^i).$$

Ясно, что $x^n - 1 = \prod_s M^{(s)}(x)$, где s пробегает множество представителей ЦТК. Максимальный размер ЦТК по модулю n над $GF(p)$ (или, эквивалентно, максимальная степень неприводимого над $GF(p)$ сомножителя $x^n - 1$) указывает то расширение поля, в котором существуют корни степени n из 1. Элементы $\beta^i, i \in C_s$ (или являющиеся корнями одного минимального многочлена) называются сопряженными. Заметим, что это же определение справедливо и в поле комплексных чисел, где элементы $\pm i$ являются корнями $x^2 + 1$.

Далее, если не указано иное, будут рассматриваться поля характеристики 2.

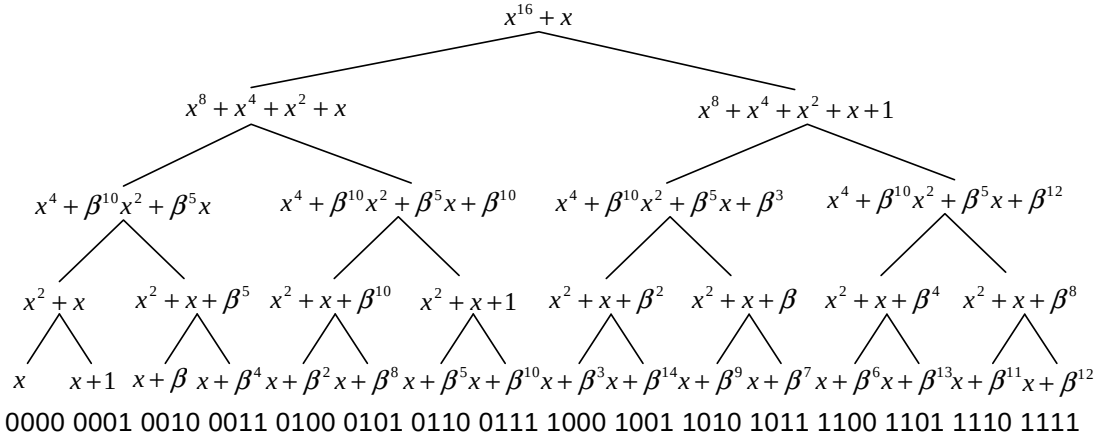


Рис. 3.6: Рекурсивное разбиение Ванга-Жу

Алгоритм Ванга-Жу

Классический алгоритм Герцеля заменяет задачу вычисления значений многочлена $f(x)$ в точках α^i вычислением значений остатков от деления $f(x)$ на минимальные многочлены элементов α^i, т.е. остатков от деления на $(x-\alpha^i)$. Это требует $O(n\log_2 n)$ умножений и $O(n^2)$ сложений. В [21] было предложено обобщение этого подхода, состоящее в том, что деление может производиться на некоторые многочлены с корнями α^i, от которых требуется лишь, чтобы число ненулевых (или вычислительно сложных) коэффициентов в них было небольшим относительно их степени. Такому критерию удовлетворяют линеаризованные и аффинные многочлены.

Пример 3.13. Рассмотрим вычисление ДПФ над $GF(2^4)$. В качестве ядра преобразования воспользуемся примитивным элементом поля, т.е. $\alpha = \beta$. Будем вычислять расширенное преобразование, т.е. дополним вектор $(F_0,\ldots,F_{n-1})$ компонентой $F_{-\infty} = f(0)$. Пример подобного разбиения представлен на рис. 3.6. На первом шаге алгоритма вычисляются остатки от деления $f(x) = \sum_{i=0}^{15} f_i x^i$ на многочлены $x^8+x^4+x^2+x$ и $x^8+x^4+x^2+x+1$. Далее осуществляется рекурсивное вычисление остатков от деления на их сомножители до тех пор, пока не получается остаток нулевой степени, т.е. компоненты ДПФ.

Как было показано выше, множество корней линеаризованного многочлена образует линейное подпространство $GF(2^m)$. В частности, множество корней $x^{2^m}-x$ совпадает с линейным пространством $GF(2^m)$. Пусть задан линеаризованный многочлен $L_j(x)$ с j-мерным пространством корней $\mathcal{L}_j$. Выберем какое-нибудь $j-1$-мерное подпространство $\mathcal{L}_{j-1}$ его корней. Ясно, что оно соответствует некоторому другому линеаризованному многочлену $\mathcal{L}_{j-1}$. Оставшиеся корни могут представлены как (смежный класс $\mathcal{L}_{j-1}$)

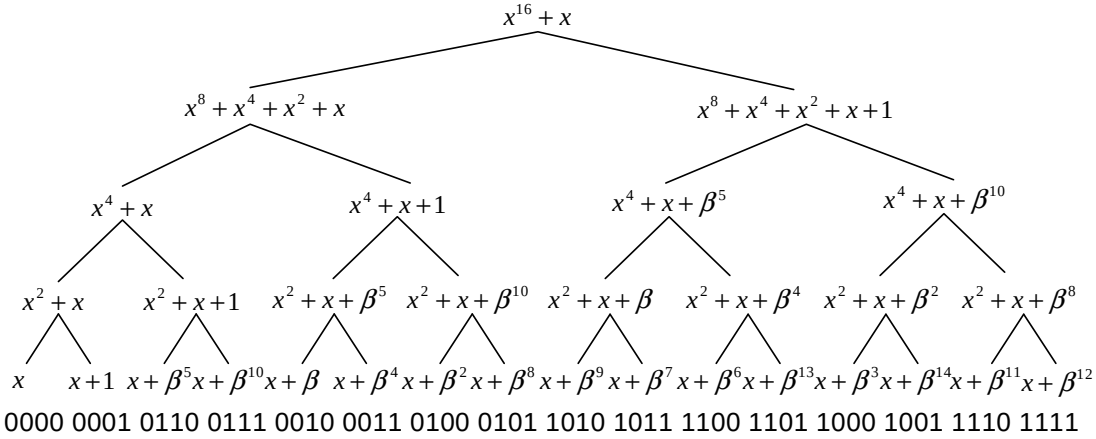


Рис. 3.7: Рекурсивное разбиение Ванга-Жу: группировка по модулям

$\alpha_i = \hat{\alpha} + \alpha_k, \alpha_k \in \mathcal{L}_{j-1}, \hat{\alpha} \notin \mathcal{L}_{j-1}$. Тогда $\prod_{\alpha_i \notin \mathcal{L}_{j-1}} (x - \alpha_i) = \prod_{\alpha_k \in \mathcal{L}_{j-1}} (x - \hat{\alpha} - \alpha_k) = L_{j-1}(x - \hat{\alpha}) = \sum_{s=0}^{j-1} l_{j-1,s}(x - \hat{\alpha})^{2^s} = \sum_{s=0}^{j-1} l_{j-1,s}x^{2^s} + \sum_{s=0}^{j-1} l_{j-1,s}\hat{\alpha}^{2^s} = L_{j-1}(x) + a$, что есть аффинный многочлен. Таким образом, произвольный линеаризованный многочлен может быть разложен на произведение линеаризованного и аффинного многочленов. Это означает, что корни n-й степени из единицы (и 0) должны быть сгруппированы по смежным классам линейных подпространств. Такое расположение элементов можно получить, упорядочив их в соответствии с двоичным значением их векторного представления. Однако такое представление не является самым эффективным. Желательно максимизировать число 2-полиномов в дереве разбиения. Корни 2-полинома обладают важным дополнительным свойством: если $L(\mu) = 0$, то $0 = L^2(\mu) = L(\mu^2)$. Подпространство с таким свойством называется модулем. Таким образом, желательно группировать элементы поля в модули. Однако до настоящего времени не ясно, всегда ли возможно такое разбиение. Пример его приведен на рис. 3.7 . Можно показать, что число сложений и умножений не превосходит $O(n \log^2 n)$.

Циклотомический алгоритм БПФ

Другой способ использования факта существования линеаризованных многочленов был предложен в [12]. Рассмотрим набор циклотомических классов по модулю n над $GF(2)$:

$$\{0\}, \{k_1, k_1 2, k_1 2^2, \ldots, k_1 2^{m_1 - 1}\}, \ldots, \{k_l, k_l 2, k_l 2^2, \ldots, k_l 2^{m_l - 1}\},$$

где $k_i \equiv k_i 2^{m_i} \bmod n$.

Многочлен $f(x) = \sum_{i=0}^{n-1} f_i x^i, f_i \in GF(2^m)$ может быть разложен как

$$f(x) = \sum_{i=0}^{l} L_i(x^{k_i}), \quad L_i(y) = \sum_{j=0}^{m_i-1} f_{k_i 2^j \bmod n} y^{2^j}. \tag{3.15}$$

Действительно, выражение (3.15) представляет собой способ группировки чисел $s \in [0, n-1]$ по циклотомическим классам: $s \equiv k_i 2^j \bmod n$. Очевидно, что такое разбиение существует всегда. Заметим, что при $k_i = 0$ свободный член f_0 мы можем записать как значение многочлена $L_0(y) = f_0 y$ при $y = x^0$.

Выражение (3.15) будем называть циклотомическим разложением многочлена $f(x)$.

Пример 3.14. Многочлен $f(x) = \sum_{i=0}^{6} f_i x^i, f_i \in GF(2^3)$ представляется как

$$\begin{aligned} f(x) = &\quad L_0(x^0) + L_1(x) + L_2(x^3); \\ L_0(y) = &\quad f_0 y, \\ L_1(y) = &\quad f_1 y + f_2 y^2 + f_4 y^4, \\ L_2(y) = &\quad f_3 y + f_6 y^2 + f_5 y^4. \end{aligned}$$

В соответствии с разложением (3.15) запишем $f(\alpha^j) = \sum_{i=0}^{l} L_i(\alpha^{j\,k_i})$. Как известно [11], элемент α^{k_i} является корнем соответствующего минимального многочлена степени m_i и, следовательно, лежит в подполе $GF(2^{m_i})$, $m_i \mid m$. Таким образом, все величины $(\alpha^{k_i})^j$ принадлежат полю $GF(2^{m_i})$ и могут быть разложены в каком-либо базисе $(\pi_{i,0}, \ldots, \pi_{i,m_i-1})$ этого поля: $\alpha^{j\,k_i} = \sum_{s=0}^{m_i-1} a_{ijs}\pi_{i,s}, a_{ijs} \in GF(2)$. Тогда значения каждого из линеаризованных многочленов могут быть вычислены в базисных точках соответствующего подполя по формуле

$$L_i(\pi_{i,s}) = \sum_{p=0}^{m_i-1} \pi_{i,s}^{2^p} f_{k_i 2^p}, \quad i \in [0, l], s \in [0, m_i - 1]. \tag{3.16}$$

Базисы $(\pi_{i,0}, \ldots, \pi_{i,m_i-1})$ для каждого из линеаризованных многочленов $L_i(y)$ могут выбираться независимо.

В соответствии с теоремой 3.6 компоненты преобразования Фурье многочлена $f(x)$ являются линейными комбинациями этих значений

$$F_j = f(\alpha^j) = \sum_{i=0}^{l} \sum_{s=0}^{m_i-1} a_{ijs} L_i(\pi_{i,s}) = \tag{3.17}$$

$$\sum_{i=0}^{l} \sum_{s=0}^{m_i-1} a_{ijs} \left(\sum_{p=0}^{m_i-1} \pi_{i,s}^{2^p} f_{k_i 2^p} \right), \quad j \in [0, n-1].$$

Последнее выражение может быть записано в матричной форме как $F = ALf$, где $F = (F_0, F_1, \ldots, F_{n-1})^T$, $f = (f_0, f_{k_1}, f_{k_1 2}, f_{k_1 2^2}, \ldots, f_{k_1 2^{m_1-1}}, \ldots, f_{k_l}, f_{k_l 2}, f_{k_l 2^2}, \ldots, f_{k_l 2^{m_l-1}})^T$

есть перестановка вектора коэффициентов исходного многочлена $f(x)$, соответствующая разложению (3.15), A — матрица, составленная из элементов $a_{ijs} \in GF(2)$, L — блочно-диагональная матрица, составленная из элементов $\pi_{i,s}^{2^p}$. Очевидно, что для линеаризованных многочленов одинаковой степени m_i, входящих в разложение (3.15), можно выбрать одинаковые базисы $(\pi_{i,s})$ в подполях $GF(2^{m_i})$, вследствие чего матрица L будет содержать большое число одинаковых блоков.

Таким образом, задача БПФ разбивается на два этапа: умножение блочно-диагональной матрицы L на исходный вектор f и умножение двоичной матрицы A на полученный вектор $S = Lf$

$$F = ALf. \tag{3.18}$$

Рассмотрим более подробно первый этап преобразования Фурье — задачу вычисления произведения $S = Lf$. Блочно-диагональная матрица

$$L = \begin{pmatrix} L_0 & 0 & \dots & 0 \\ 0 & L_1 & \dots & 0 \\ \dots & \dots & \ddots & \dots \\ 0 & 0 & \dots & L_l \end{pmatrix}$$

состоит из блоков

$$L_i = \begin{pmatrix} \pi_{i,0} & \pi_{i,0}^2 & \dots & \pi_{i,0}^{2^{m_i-1}} \\ \pi_{i,1} & \pi_{i,1}^2 & \dots & \pi_{i,1}^{2^{m_i-1}} \\ \dots & \dots & \dots & \dots \\ \pi_{i,m_i-1} & \pi_{i,m_i-1}^2 & \dots & \pi_{i,m_i-1}^{2^{m_i-1}} \end{pmatrix}.$$

Выберем в качестве $(\pi_{i,0}, \dots, \pi_{i,m_i-1})$ нормальный базис $\left(\gamma_i, \gamma_i^2, \dots, \gamma_i^{2^{m_i-1}}\right)$. Тогда матрица L состоит из блоков вида

$$L_i = \begin{pmatrix} \gamma_i^{2^0} & \gamma_i^2 & \dots & \gamma_i^{2^{m_i-1}} \\ \gamma_i^2 & \gamma_i^4 & \dots & \gamma_i^{2^0} \\ \dots & \dots & \dots & \dots \\ \gamma_i^{2^{m_i-1}} & \gamma_i^{2^0} & \dots & \gamma_i^{2^{m_i-2}} \end{pmatrix}.$$

В силу блочно-диагональной структуры матрицы L вычисление произведения $S = Lf$ может быть представлено как $S = (b_0, b_1, \dots, b_l)^T = L\,(a_0, a_1, \dots, a_l)^T$, где $b_i = (b_{i,0}, b_{i,1}, \dots, b_{i,m_i-1})$ — подвектора искомого вектора S, $a_i = (a_{i,0}, a_{i,1}, \dots, a_{i,m_i-1})$ — подвектора исходного вектора f.

Представим вычисление $b_i^T = L_i a_i^T$ как циклическую свертку

$$b_i(x) = b_{i,0} + b_{i,m_i-1}x + \ldots + b_{i,1}x^{m_i-1} = (\gamma_i + \gamma_i^{2^{m_i-1}}x + \ldots + \gamma_i^2 x^{m_i-1})(a_{i,0} + a_{i,1}x + \ldots + a_{i,m_i-1}x^{m_i-1}) \bmod (x^{m_i} - 1).$$

Для ее вычисления могут быть применены известные алгоритмы [5, 8, 2]. При этом использование свойства нормального базиса $\gamma_i + \gamma_i^2 + \ldots + \gamma_i^{2^{m_i-1}} = 1$ позволяет заметно сократить число операций при вычислении циклической свертки. Отметим, что вычисление значений линеаризованных многочленов с помощью циклической свертки было описано в монографии [8].

Описанный подход позволяет свести задачу умножения блочно-диагональной матрицы L на исходный вектор f над $GF(2^m)$ к задаче вычисления $l+1$ циклических сверток малой длины m_i. Существующие алгоритмы вычисления циклических сверток $b_i(x) = \gamma_i(x)a_i(x) \bmod (x^{m_i} - 1)$ могут быть записаны в матричном виде как

$$b_i = \begin{pmatrix} b_{i,0} \\ b_{i,1} \\ \ldots \\ b_{i,m_i-1} \end{pmatrix} = Q_i \left(\left(D_i \begin{pmatrix} \gamma_i \\ \gamma_i^{2^{m_i-1}} \\ \ldots \\ \gamma_i^2 \end{pmatrix} \right) \cdot (P_i a_i) \right), \tag{3.19}$$

где Q_i, D_i и P_i являются двоичными матрицами, а $x \cdot y$ обозначает покомпонентное произведение векторов. Очевидно, что вектор $C_i = D_i \left(\gamma_i, \gamma_i^{2^{m_i-1}}, \ldots, \gamma_i^2\right)^T$ может быть вычислен заранее. Таким образом, выражение (3.18) может быть переписано как

$$F = AQ\left(C \cdot (Pf)\right), \tag{3.20}$$

где Q — двоичная блочно-диагональная матрица объединенных последующих сложений для $l+1$ циклической свертки, C — объединенный вектор констант, P — двоичная блочно-диагональная матрица объединенных предварительных сложений.

Учитывая формулы (3.18) и (3.20), второй этап БПФ может рассматриваться как умножение двоичной матрицы AQ на вектор $C \cdot (Pf)$. Для вычисления произведения $(AQ)\left(C \cdot (Pf)\right)$ могут быть использованы модифицированный алгоритм "четырех русских" (В.Л.Арлазаров, Е.А.Диниц, М.А.Кронрод, И.А.Фараджев) для умножения булевых матриц со сложностью $O(n^2/\log n)$ сложений над элементами поля $GF(2^m)$ [3] или специализированный алгоритм, основанный на итеративном алгоритме декодирования низкоплотностных кодов.

Как было показано, задача вычисления БПФ многочлена над $GF(2^m)$ сводится к набору задач вычисления циклической свертки многочленов и умножению на двоичную матрицу. Предполагая, что сложность вычисления короткой циклической свертки в конечном поле асимптотически равна $C_{conv}(t) = O(m\log^t m), t \geq 2$, получим, что сложность всего алгоритма равна

$$\begin{aligned} C = \ & \textstyle\sum_{i=0}^{l} C_{conv}(m_i) + C_{bm} = \sum_{i=0}^{l} O(m_i \log^t m_i) + C_{bm} = \\ & O(\tfrac{n}{m} m \log^t m) + C_{bm} = O(n \log^t \log n) + C_{bm}. \end{aligned}$$

Здесь C_{bm} — сложность умножения на двоичную матрицу. Асимптотическая оценка сложности процедуры умножения на двоичную матрицу, порождаемой вышеописанным алгоритмом в настоящее время неизвестна и требует дальнейших исследований.

Пример 3.15. Продолжим рассмотрение БПФ длины 7 над полем $GF(2^3)$. Пусть α — корень примитивного многочлена $x^3 + x + 1$. В качестве базиса поля $GF(2^3)$ выберем нормальный базис $(\gamma, \gamma^2, \gamma^4)$, где $\gamma = \alpha^3$. Разложим многочлен $f(x)$ как в примере 3.14 и представим компоненты преобразования Фурье в виде сумм

$$\begin{aligned}
f(\alpha^0) = &\quad L_0(\alpha^0) + L_1(\alpha^0) + L_2(\alpha^0) = &&L_0(1) + L_1(\gamma) + L_1(\gamma^2) + L_1(\gamma^4) + \\
&&&L_2(\gamma) + L_2(\gamma^2) + L_2(\gamma^4) \\
f(\alpha^1) = &\quad L_0(\alpha^0) + L_1(\alpha) + L_2(\alpha^3) = &&L_0(1) + L_1(\gamma^2) + L_1(\gamma^4) + L_2(\gamma) \\
f(\alpha^2) = &\quad L_0(\alpha^0) + L_1(\alpha^2) + L_2(\alpha^6) = &&L_0(1) + L_1(\gamma) + L_1(\gamma^4) + L_2(\gamma^2) \\
f(\alpha^3) = &\quad L_0(\alpha^0) + L_1(\alpha^3) + L_2(\alpha^2) = &&L_0(1) + L_1(\gamma) + L_2(\gamma) + L_2(\gamma^4) \\
f(\alpha^4) = &\quad L_0(\alpha^0) + L_1(\alpha^4) + L_2(\alpha^5) = &&L_0(1) + L_1(\gamma) + L_1(\gamma^2) + L_2(\gamma^4) \\
f(\alpha^5) = &\quad L_0(\alpha^0) + L_1(\alpha^5) + L_2(\alpha) = &&L_0(1) + L_1(\gamma^4) + L_2(\gamma^2) + L_2(\gamma^4) \\
f(\alpha^6) = &\quad L_0(\alpha^0) + L_1(\alpha^6) + L_2(\alpha^4) = &&L_0(1) + L_1(\gamma^2) + L_2(\gamma) + L_2(\gamma^2).
\end{aligned}$$

Эти тождества могут быть записаны в матричной форме как

$$F = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} L_0(1) \\ L_1(\gamma) \\ L_1(\gamma^2) \\ L_1(\gamma^4) \\ L_2(\gamma) \\ L_2(\gamma^2) \\ L_2(\gamma^4) \end{pmatrix} = AS.$$

Тогда задачу БПФ можно переписать в виде

$$F = A \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \gamma^1 & \gamma^2 & \gamma^4 & 0 & 0 & 0 \\ 0 & \gamma^2 & \gamma^4 & \gamma^1 & 0 & 0 & 0 \\ 0 & \gamma^4 & \gamma^1 & \gamma^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \gamma^1 & \gamma^2 & \gamma^4 \\ 0 & 0 & 0 & 0 & \gamma^2 & \gamma^4 & \gamma^1 \\ 0 & 0 & 0 & 0 & \gamma^4 & \gamma^1 & \gamma^2 \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_4 \\ f_3 \\ f_6 \\ f_5 \end{pmatrix}.$$

Первый этап алгоритма БПФ состоит в вычислении двух циклических сверток

$$\begin{pmatrix} b_{i,0} \\ b_{i,1} \\ b_{i,2} \end{pmatrix} = \begin{pmatrix} \gamma^1 & \gamma^2 & \gamma^4 \\ \gamma^2 & \gamma^4 & \gamma^1 \\ \gamma^4 & \gamma^1 & \gamma^2 \end{pmatrix} \begin{pmatrix} a_{i,0} \\ a_{i,1} \\ a_{i,2} \end{pmatrix}, \quad i = 1, 2,$$

где

$$S = \begin{pmatrix} L_0(1) \\ L_1(\gamma) \\ L_1(\gamma^2) \\ L_1(\gamma^4) \\ L_2(\gamma) \\ L_2(\gamma^2) \\ L_2(\gamma^4) \end{pmatrix} = \begin{pmatrix} b_{0,0} \\ b_{1,0} \\ b_{1,1} \\ b_{1,2} \\ b_{2,0} \\ b_{2,1} \\ b_{2,2} \end{pmatrix}, \qquad f = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_4 \\ f_3 \\ f_6 \\ f_5 \end{pmatrix} = \begin{pmatrix} a_{0,0} \\ a_{1,0} \\ a_{1,1} \\ a_{1,2} \\ a_{2,0} \\ a_{2,1} \\ a_{2,2} \end{pmatrix}.$$

Используя алгоритм вычисления трехточечной циклической свертки $b_i(x) = b_{i,0} + b_{i,2}x + b_{i,1}x^2 = (\gamma + \gamma^4 x + \gamma^2 x^2)(a_{i,0} + a_{i,1}x + a_{i,2}x^2) \bmod (x^3 - 1)$, представленный в [5], получим

$$b_i = \begin{pmatrix} b_{i,0} \\ b_{i,1} \\ b_{i,2} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \left(\left[\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \gamma \\ \gamma^4 \\ \gamma^2 \end{pmatrix} \right] \cdot \left[\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{i,0} \\ a_{i,1} \\ a_{i,2} \end{pmatrix} \right] \right) =$$

$$Q_i \left(C_i \cdot (P_i a_i) \right), \quad i = 1, 2.$$

С учетом $\gamma + \gamma^2 + \gamma^4 = 1$ видно, что алгоритм требует 3 умножений, 4 предварительных и 5 последующих сложений.

Теперь можно записать формулу (3.20) для рассматриваемого примера в матричной форме

$$F = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{pmatrix} = \left(\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \right) \times$$

$$\left(\begin{pmatrix} 1 \\ 1 \\ \gamma^2 + \gamma^4 \\ \gamma + \gamma^4 \\ \gamma + \gamma^2 \\ 1 \\ \gamma^2 + \gamma^4 \\ \gamma + \gamma^4 \\ \gamma + \gamma^2 \end{pmatrix} \cdot \left[\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_4 \\ f_3 \\ f_6 \\ f_5 \end{pmatrix} \right] \right) = (AQ)(C \cdot (Pf)).$$

Второй этап БПФ состоит в умножении двоичной матрицы AQ на вектор $C \cdot (Pf)$

$$F = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} (C \cdot (Pf)).$$

Этот этап может быть выполнен за 17 сложений.

Таким образом, БПФ длины 7 сводится к следующей последовательности действий:

выполнение предварительных сложений $P \times f$

$$\begin{array}{ll} V_1 = f_2 + f_4 & V_8 = f_6 + f_5 \\ V_2 = f_1 + f_2 & V_9 = f_3 + f_6 \\ V_3 = f_1 + f_4 & V_{10} = f_3 + f_5 \\ V_4 = f_1 + V_1 & V_{11} = f_3 + V_8, \end{array}$$

выполнение умножений на константы $C \cdot (Pf)$

$$\begin{array}{ll} V_5 = V_1\,\alpha & V_{12} = V_8\,\alpha \\ V_6 = V_2\,\alpha^2 & V_{13} = V_9\,\alpha^2 \\ V_7 = V_3\,\alpha^4 & V_{14} = V_{10}\,\alpha^4, \end{array}$$

умножение матрицы AQ на вектор $C \cdot (Pf)$

$$\begin{array}{ll} T_{10} = V_{12} + V_{14} & F_2 = T_8 + T_{11} \\ T_{11} = f_0 + V_{11} & F_3 = T_7 + T_{14} \\ T_{14} = f_0 + V_4 & T_{12} = F_2 + T_{10} \\ T_{15} = V_5 + V_6 & T_{13} = F_3 + T_{15} \\ T_{16} = V_6 + V_{13} & F_4 = T_7 + T_{12} \\ F_0 = V_4 + T_{11} & F_5 = T_{10} + T_{13} \\ T_9 = V_{12} + T_{16} & F_6 = F_5 + T_7 \\ T_7 = V_7 + T_9 & F_1 = F_4 + T_8. \\ T_8 = V_5 + T_9 & \end{array}$$

Общая сложность алгоритма составляет $2 \times 3 = 6$ умножений и $2 \times 4 + 17 = 25$ сложений, что на одно сложение меньше, чем для алгоритма, представленного в работе [9].

3.4.4 Применение БПФ для вычисления свертки

В соответствии с теоремой 3.4 (о свертке), если существует ДПФ длины n, то соотношения $E_i = F_i G_i, i = 0..n-1$ и $e_j = \frac{1}{n} \sum_{k=0}^{n-1} f_{((j-k))} g_k$ равносильны. Таким образом, для вычисления $e(x) = f(x)g(x) \bmod (x^n - 1)$ необходимо выполнить следующие операции:

1. Вычислить ДПФ $f(x)$ и $g(x)$.
2. Вычислить $\frac{1}{n} E_i = F_i G_i, i = 0..n-1$.
3. Вычислить обратное ДПФ вектора E_i.

Данный алгоритм требует трехкратного обращения к алгоритму БПФ и n умножений. Этот метод может использоваться и для вычисления линейной свертки. Для этого необходимо дополнить многочлены $f(x)$ и $g(x)$ степени n нулями до степени $n' \geq 2n$ и воспользоваться вышеописанным методом вычисления $(n'+1)$-точечной циклической свертки.

3.4.5 Алгоритм Шёнхаге-Штрассена

В некоторых случаях при попытке вычисления линейной свертки вышеописанным методом построение алгоритмов БПФ требуемой размерности оказывается невозможным или весьма затруднительным. Как правило, это связано с проблемой отыскания подходящего примитивного корня из единицы. В этом случае в рассматриваемую алгебру может быть искусственно введен дополнительный элемент, с

помошью которого может быть построено ДПФ (и быстрый алгоритм его вычисления) нужной размерности. Данный метод используется в *алгоритме Шёнхаге-Штрассена* вычисления линейной свертки.

Рассмотрим задачу вычисления $h(x) = f(x)g(x) : \deg h(x) < n = 2^k$. Пусть коэффициенты этих многочленов принадлежат некоторому кольцу R, такому что 2 является в нем обратимым элементом. Пусть $m = 2^{\lfloor k/2 \rfloor}$ и $t = n/m = 2^{\lceil k/2 \rceil}$. Представим перемножаемые многочлены как

$$f(x) = f'(x, x^m) = \sum_{i=0}^{t-1} f_i(x)x^{mi}, g(x) = g'(x, x^m) = \sum_{i=0}^{t-1} g_i(x)x^{mi}.$$

Таким образом, задача снова сведена к вычислению двумерной линейной свертки $h'(x, y) = f'(x, y)g'(x, y)$ и $h(x) = h'(x, x^m)$. В действительности достаточно вычислить ее по модулю $y^t + 1$, т.к.

$$f'(x, y)g'(x, y) = h'(x, y) + q'(x, y)(y^t + 1)$$

означает $h(x) = f'(x, x^m)g'(x, x^m) = h'(x, x^m) + q'(x, x^m)(x^{mt} + 1) \equiv h'(x, x^m) \bmod (x^n + 1)$. В данном случае с таким же успехом вычисления можно было бы проводить по модулю $y^t - 1$, однако такой алгоритм впоследствие будет сложно использовать рекурсивно.

Заметим, что $\deg g_i(x), f_i(x) \leq m - 1$. Это позволяет рассматривать их как элементы факторкольца $D = R[x]/ < x^{2m} + 1 >$. Оказывается, что это факторкольцо содержит элемент, на основе которого может быть построено ДПФ. Действительно, $\xi \equiv x \bmod (x^{2m} + 1)$ является примитивным корнем из единицы степени $4m$, т.к. $\xi^{4m} - 1 \equiv (x^{2m} - 1)(x^{2m} + 1) \equiv 0 \bmod (x^{2m} + 1)$ и $\xi^{2m} - 1 \equiv -2 \not\equiv 0 \bmod (x^{2m} + 1)$. Таким образом, возникла задача перемножения многочленов от переменной y, коэффициентами которых являются элементы из факторкольца, содержащего примитивный корень из единицы. Пусть $f^*(y) = f'(x, y) \bmod (x^{2m} + 1), g^*(y) = g'(x, y) \bmod (x^{2m} + 1)$. Если $t = 2m$, пусть $\eta = \xi$. В противном случае, если $t = m$, пусть $\eta = \xi^2$. Тогда η является примитивным корнем степени $2t$ из единицы и $\eta^t = -1$. С другой стороны, $\omega = \eta^2$ является примитивным корнем степени t из единицы. Таким образом, получаем

$$f^*(\eta y)g^*(\eta y) = h^*(\eta y) \bmod ((\eta y)^t + 1)$$

или

$$f^*(\eta y)g^*(\eta y) = h^*(\eta y) \bmod (y^t - 1).$$

Задача свелась к нахождению t-точечной циклической свертки. Она может быть вычислена с использованием алгоритма БПФ с ядром ω. При этом умножение на различные степени ω есть фиктивная операция, на самом деле состоящая в сдвиге вектора коэффициентов многочлена на определенное число позиций. Однако в результате умножений на степени ω приведение по модулю $x^{2m} + 1$ перестает быть фиктивной операцией, что необходимо учесть при построении алгоритма.

Таким образом, первый этап алгоритма включает в себя исключительно сложения. Далее необходимо вычислить покомпонентное произведение полученных векторов ДПФ.

Т.к. на самом деле компоненты этих векторов являются многочленами от x, их вычисление может быть произведено рекурсивно с помощью этого же метода. Последний этап алгорима вычисления циклической свертки снова требует нахождения обратного ДПФ с ядром ω, которое опять не требует умножений.

Таким образом, быстрое вычисление $h(x) = f(x)g(x) \bmod (x^n + 1)$ включает в себя следующие операции:

1. Если степень произведения $n = \deg h(x)$ меньше некоторого порога k, воспользоваться алгоритмом Карацубы или вычислить свертку по определению.
2. Пусть $n = 2^k, m = 2^{\lfloor k/2 \rfloor}, t = n/m$. Построить $f'(x,y), g'(x,y) : f(x) = f'(x, x^m), g(x) = g'(x, x^m)$.
3. Если $t = 2m$, положить $\eta = x \bmod (x^{2m} + 1)$, иначе $\eta = x^2 \bmod (x^{2m} + 1)$. Пусть $f^*(y) = f'(x,y), g^*(y) = g'(x,y)$.
4. Вычислить t-точечное ДПФ с ядром $\omega = \eta^2$ многочленов $f^*(\eta y), g^*(\eta y)$.
5. Вычислить $H_i^* = \frac{1}{t} F_i^* G_i^* \bmod (x^{2m} + 1), i = 0..t-1$.
6. Вычислить t-точечное обратное ДПФ с ядром $\omega = \eta^2$ вектора $(H_0^*, \ldots, H_{t-1}^*)$. Это приведет к многочлену $h^*(\eta y)$.
7. Восстановить $h(x) = h'(x, x^m)$.

Теорема 3.7. *Алгоритм Шёнхаге-Штрассена корректно вычисляет линейную свертку и требует* $\frac{9}{2} n \log n \log\log n + O(n \log n)$ *операций.*

Доказательство. Корректность алгоритма следует из вышеприведенных рассуждений. Пусть $T(k)$ обозначает его сложность для $n = 2^k$. Вычисление трех ДПФ требует $3t \log t$ сложений в кольце $D = R/ < x^{2m} + 1 >$, $\frac{3}{2} t \log t$ умножений на степени ω и t делений на $t \in R$. Одно сложение в D соответствует $2m$ сложениям в R, также как и деление на t, умножение на степени ω требует не более чем $2m$ операций. Перемножение элементов D по модулю $x^{2m} + 1$ может быть выполнено рекурсивно с использованием этого же алгоритма со сложностью $T(\lfloor k/2 \rfloor + 1)$. Подстановки $y \leftrightarrow \eta y$ требуют всего $3t$ операций умножения на степени η. Таким образом, имеет место $T(k) \leq 2^{\lceil k/2 \rceil} T(\lfloor k/2 \rfloor + 1) + 9mt \log t + 8mt$. Таким образом,

$$T(k) \leq 2^{\lfloor k/2 \rfloor} T(\lfloor k/2 \rfloor + 1) + 92^k (\lceil k/2 \rceil + 1)$$

□

Пример 3.16. Найдем произведение многочленов $f(x) = \sum_{i=0}^{15} f_i x^i$ и $g(x) = \sum_{i=0}^{15} g_i x^i, \in \mathbb{Q}[x]$. Произведение этих многочленов должно иметь тридцатую степень, т.е. непосредственное применение теоремы о свертке требовало бы использования 31- или 32-точечного алгоритма БПФ. Поле рациональных чисел не содержит примитивных корней из единицы третьей и более высоких степеней, поэтому построение ДПФ, не

говоря уже об алгоритмах быстрого его вычисления, невозможно[1]. Пусть $m = 4, t = 8$. Тогда $f'(x,y) = (f_0 + f_1x + f_2x^2 + f_3x^3) + y(f_4 + f_5x + f_6x^2 + f_7x^3) + y^2(f_8 + f_9x + f_{10}x^2 + f_{11}x^3) + y^3(f_{12} + f_{13}x + f_{14}x^2 + f_{15}x^3)$, $g'(x,y) = (g_0 + g_1x + g_2x^2 + g_3x^3) + y(g_4 + g_5x + g_6x^2 + g_7x^3) + y^2(g_8 + g_9x + g_{10}x^2 + g_{11}x^3) + y^3(g_{12} + g_{13}x + g_{14}x^2 + g_{15}x^3)$ и $h(x) = f(x)g(x) = f'(x,x^8)g'(x,x^8)$. Обозначая $\xi \equiv x \bmod (x^8+1)$, получим эквивалентную задачу $h^*(y) = ((f_0 + f_1\xi + f_2\xi^2 + f_3\xi^3) + y(f_4 + f_5\xi + f_6\xi^2 + f_7\xi^3) + y^2(f_8 + f_9\xi + f_{10}\xi^2 + f_{11}\xi^3) + y^3(f_{12} + f_{13}\xi + f_{14}\xi^2 + f_{15}\xi^3))((g_0 + g_1\xi + g_2\xi^2 + g_3\xi^3) + y(g_4 + g_5\xi + g_6\xi^2 + g_7\xi^3) + y^2(g_8 + g_9\xi + g_{10}\xi^2 + g_{11}\xi^3) + y^3(g_{12} + g_{13}\xi + g_{14}\xi^2 + g_{15}\xi^3) \bmod (y^8 - 1)$. Приведение по модулю $y^8 - 1$ в данном случае является фиктивной операцией. С таким же успехом произведение можно было бы привести по модулю $y^8 + 1$. Видно, что ξ является примитивным корнем шестнадцатой степени из единицы, т.к. $x^8 \equiv -1 \bmod (x^8+1)$ Обозначая $\eta = \xi$, получим

$$h^*(y) = f^*(y)g^*(y) \bmod (y^8 - 1).$$

Пусть $\omega = \eta^2 \equiv x^2 \bmod (x^8+1)$. Заметим, что $\omega^8 = 1$ и $\omega^4 = -1$, т.е. величина ω может быть использована для построения 8-точечного ДПФ. Соответствующий быстрый алгоритм (Кули-Тьюки) имеет следующий вид ($c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7$ обозначает компоненты преобразуемого вектора):

$$\begin{aligned}
t_{00} = c_0 + c_4; t_{04} = c_0 - c_4; \quad & t_{01} = c_1 + c_5; t_{05} = \omega(c_1 - c_5); \\
t_{02} = c_2 + c_6; t_{06} = \omega^2(c_2 - c_6); \quad & t_{03} = c_3 + c_7; t_{07} = \omega^3(c_3 - c_7) \\
t_{10} = t_{00} + t_{02}; t_{12} = t_{00} - t_{02}; \quad & t_{11} = t_{01} + t_{03}; t_{13} = \omega^2(t_{01} - t_{03}); \\
t_{14} = t_{04} + t_{06}; t_{16} = t_{04} - t_{06}; \quad & t_{15} = t_{05} + t_{07}; t_{17} = \omega^2(t_{05} - t_{07}) \\
C_0 = t_{10} + t_{11}; C_4 = t_{10} - t_{11}; \quad & C_2 = t_{12} + t_{13}; C_6 = t_{12} - t_{13}; \\
C_1 = t_{14} + t_{15}; C_5 = t_{14} - t_{15}; \quad & C_3 = t_{16} + t_{17}; C_7 = t_{16} - t_{17};
\end{aligned}$$

Заметим, что умножение на $\omega \equiv x^2 \bmod (x^8+1)$ соответствует циклическому сдвигу вектора коэффициентов с частичным изменением их знака. В данном случае компоненты ДПФ многочлена $f^*(y)$ задаются выражениями

$$\begin{aligned}
F_0 &= (f_{11} + f_7 + f_{15} + f_3)x^3 + (f_{14} + f_{10} + f_6 + f_2)x^2 + (f_1 + f_9 + f_5 + f_{13})x + f_{12} + f_0 + f_4 + f_8 \\
F_1 &= (f_{11} + f_{13})x^7 + (f_{10} + f_{12})x^6 + (f_7 + f_9)x^5 + (f_6 + f_8)x^4 + (f_3 + f_5)x^3 + (f_4 + f_2)x^2 + (f_1 - f_{15})x + f_0 - f_{14} \\
F_2 &= (f_7 - f_{15})x^7 + (f_6 - f_{14})x^6 + (f_5 - f_{13})x^5 + (f_4 - f_{12})x^4 + (f_3 - f_{11})x^3 + (f_2 - f_{10})x^2 + (f_1 - f_9)x + f_0 - f_8 \\
F_3 &= (f_5 - f_{11})x^7 + (f_4 - f_{10})x^6 + (f_{15} - f_9)x^5 + (f_{14} - f_8)x^4 + (f_3 + f_{13})x^3 + (f_2 + f_{12})x^2 + (f_1 - f_7)x + f_0 - f_6 \\
F_4 &= (f_3 + f_{11} - f_7 - f_{15})x^3 + (-f_{14} + f_{10} - f_6 + f_2)x^2 + (f_1 + f_9 - f_5 - f_{13})x - f_{12} + f_0 - f_4 + f_8 \\
F_5 &= (f_{11} - f_{13})x^7 + (f_{10} - f_{12})x^6 + (f_9 - f_7)x^5 + (f_8 - f_6)x^4 + (f_3 - f_5)x^3 + (f_2 - f_4)x^2 + (f_1 + f_{15})x + f_0 + f_{14} \\
F_6 &= (f_{15} - f_7)x^7 + (f_{14} - f_6)x^6 + (f_{13} - f_5)x^5 + (f_{12} - f_4)x^4 + (f_3 - f_{11})x^3 + (f_2 - f_{10})x^2 + (f_1 - f_9)x + f_0 - f_8 \\
F_7 &= -(f_{11} + f_5)x^7 - (f_{10} + f_4)x^6 - (f_9 + f_{15})x^5 - (f_8 + f_{14})x^4 + (f_3 - f_{13})x^3 + (f_2 - f_{12})x^2 + (f_1 + f_7)x + f_0 + f_6
\end{aligned}$$

Заметим, что их вычисление не требует никаких умножений, а сложения могут быть выполнены с помощью вышеописанного метода. Аналогичным образом может быть вычислено ДПФ $g^*(y)$. Компоненты ДПФ должны быть перемножены по модулю $x^8 + 1$. Для этого можно воспользоваться итерированным алгоритмом, построенным на

[1] Можно было бы погрузить поле рациональных чисел в поле комплексных чисел, в котором существуют ДПФ любой длины, но при реализации на ЭВМ это может привести к потери точности вычислений.

основе алгоритма Карацубы (9 умножений). Применяя тот же алгоритм для вычисления обратного ДПФ, получим выражение для $h^*(y) = \sum_{i=0}^{7} h_i^*(x)y^i$. Выполняя подстановку $y = x^4$, получим выражение для $8h(x) = 8f(x)g(x)$. Таким образом, данный алгоритм требует 72 умножений. Заметим, что применение алгоритма Карацубы для вычисления $f'(x,y)g'(x,y)$ потребовало бы 81 умножения.

Лабораторная работа № 6. Реализация алгоритма БПФ в виде линейной программы

В рамках данной работы необходимо построить алгоритмы БПФ в виде линейных программ и реализовать их как непосредственно, так и с использованием SIMD-примитивов. Кроме того, необходимо реализовать программу вычисления ДПФ по определению. В пояснительной записке необходимо привести:

- Построение быстрых алгоритмов;
- Последовательность арифметических операций, соответствующую полученным алгоритмам;
- Каноническую параллельную форму алгоритма;
- Время вычисления ДПФ с помощью трех программ.

Варианты заданий:

1. Дано поле $GF(2^{10})$; построить 11-точечное БПФ с помощью метода Рейдера, 31-точечное БПФ с помощью циклотомического алгоритма;3-точечное БПФ с помощью метода Рейдера.
2. Дано поле $GF(2^{10})$; построить 31-точечное БПФ с помощью метода Рейдера, 11-точечное БПФ с помощью циклотомического алгоритма;3-точечное БПФ с помощью метода Рейдера.
3. Дано поле $GF(2^{10})$; построить 11-точечное БПФ с помощью метода Рейдера, 3-точечное БПФ с помощью циклотомического алгоритма;31-точечное БПФ с помощью метода Рейдера.
4. Дано поле $GF(2^8)$; построить 17-точечное БПФ с помощью метода Рейдера, 15-точечное БПФ с помощью циклотомического алгоритма.
5. Дано поле $GF(2^8)$; построить 17-точечное БПФ с помощью метода Рейдера, 3-точечное БПФ с помощью метода Рейдера, 5-точечное БПФ с помощью метода Рейдера.
6. Дано поле $GF(2^8)$; построить 51-точечное БПФ с помощью циклотомического алгоритма, 5-точечное БПФ с помощью метода Рейдера.

7. Дано поле $GF(2^8)$; построить 51-точечное БПФ с помощью циклотомического алгоритма, 5-точечное БПФ с помощью циклотомического алгоритма.

8. Дано поле комплексных чисел; Построить 11-точечное ДПФ с помощью метода Рейдера, 9-точечное БПФ с помощью алгоритма Кули-Тьюки, 12-точечное ДПФ с помощью алгоритма Гуда-Томаса.

9. Дано поле комплексных чисел; Построить 13-точечное ДПФ с помощью метода Рейдера, 8-точечное БПФ с помощью алгоритма Кули-Тьюки (прореживание по времени), 10-точечное ДПФ с помощью алгоритма Гуда-Томаса.

10. Дано поле комплексных чисел; Построить 11-точечное ДПФ с помощью метода Рейдера, 8-точечное БПФ с помощью алгоритма Кули-Тьюки (прореживание по частоте), 14-точечное ДПФ с помощью алгоритма Гуда-Томаса.

11. Дано поле комплексных чисел; Построить 7-точечное ДПФ с помощью метода Рейдера, 16-точечное БПФ с помощью алгоритма Кули-Тьюки (прореживание по частоте), 15-точечное ДПФ с помощью алгоритма Гуда-Томаса.

12. Дано поле комплексных чисел; Построить 13-точечное ДПФ с помощью метода Рейдера, 4-точечное БПФ с помощью алгоритма Кули-Тьюки, 20-точечное ДПФ с помощью алгоритма Гуда-Томаса.

Лабораторная работа № 7. Реализация алгоритма БПФ большой размерности

В рамках данной работы необходимо воспользоваться ранее построенными линейными подпрограммами n_1, n_2, n_3-точечного БПФ для построения программы $n_1 n_2 n_3$-точечного БПФ. В пояснительной записке необходимо привести:

- Описание быстрого алгоритма;
- Теоретическую оценку сложности алгоритма;
- Время вычисления ДПФ по определению и с помощью быстрого алгоритма.

Лабораторная работа № 8. Реализация параллельного алгоритма БПФ большой размерности

В рамках данной работы необходимо обеспечить параллельную реализацию средствами Open-MP алгоритма БПФ, реализованного в предыдущей работе. В пояснительной записке необходимо привести:

1. описание используемого алгоритма;

2. его каноническую параллельную форму;

3. теоретические оценки достижимого ускорения;

4. сравнение времени вычисления ДПФ с помощью последовательной и параллельных реализаций быстрого алгоритма.

3.5 Операции над целыми числами

3.5.1 Представление целых чисел в ЭВМ

Наибольшее распространение получили позиционные системы счисления. В позиционной системе по основанию b произвольное положительное целое число представимо как

$$a = \sum_{i \geq 0} a_i b^i, 0 \leq a_i < b$$

Существует несколько различных подходов к представлению отрицательных чисел, которые оказывают влияние на метод реализации вычислительных операций:

1. *Прямой код* или абсолютное значение со знаком, например -1234. Недостатком является существование "плюс нуля" и "минус нуля", т.е. двух кодов, обозначающих одно и то же число.

2. *Дополнительный код*, в котором отрицательное число $-a, a > 0$ представляется как $b^n - a$, где n — используемая разрядность. Например, в десятиразрядном десятичном представлении $-1 = (99999\,99999)_{10}$, а в двоичном 16-разрядном $-1 = (1111\,1111\,1111\,1111)_2$. Это эквивалентно вычислению по модулю b^n. Недостатком является несимметричность относительно нуля: для числа $-2^{n-1} = (100\ldots0)_2$ невозможно представить обратное.

3. *Обратный код*, в котором каждый разряд a_i модуля отрицательного числа a заменяется на $b - 1 - a_i$, например $-1 = (99999\,99998)_{10}$. Это эквивалентно вычислениям по модулю $b^n - 1$. Недостатком снова является наличие двух представлений нуля.

В большинстве современных ЭВМ используется дополнительный двоичный код. Основным преимуществом является то, что в этом случае старший бит указывает знак числа. При этом $-a = (-1 - a) + 1$. Операция $-1 - a$ может быть выполнена путем побитового инвертирования. Основным преимуществом дополнительного кода является то, что при сложении не требуется анализировать знаки операндов. Однако при умножении и делении учет знаков необходим. Это отражено в структуре системы команд большинства процессоров. В связи с этим далее ограничимся рассмотрением только операций сложения и умножения целых неотрицательных чисел. Интерес представляет

эффективная реализация операций над числами с числом разрядов, превышающим разрядность ЭВМ.

При работе с числами с многократной точностью иногда используется их представление в виде системы вычетов, основывающееся на китайской теореме об остатках.

Теорема 3.8 (Китайская теорема об остатках для целых чисел). *Пусть* $m_1, \ldots, m_r$ — *положительные целые взаимно простые числа. Пусть* $m = m_1 \cdot \ldots \cdot m_r$ *и пусть также* $a, u_1, \ldots, u_r$ — *целые числа. Тогда существует ровно одно целое число* u*:*

$$a \le u < a + m \wedge u \equiv u_j \bmod m_j, j = 1..r,$$

причем

$$u = a + ((u_1 M_1 + \ldots + u_r M_r - a) \bmod m),$$

где $M_j = n_j(n_j^{-1} \bmod m_j), n_j = m/m_j$.

3.5.2 Сложение

Рассмотрим реализацию сложения неотрицательных n-разрядных целых чисел $(u_{n-1}, \ldots, u_0)_b$ и $(v_{n-1}, \ldots, v_0)_b$ по основанию b. Следующий алгоритм формирует их сумму $(w_n, w_{n-1}, \ldots, w_0)_b$, причем $w_n \in \{0, 1\}$:

ADD(u, v, n)

```
 1  j ← 0; k ← 0;
 2  while j < n
 3      do w_j ← u_j + v_j + k
 4          if w_j ≥ b
 5            then w_j ← w_j − b
 6                 k = 1
 7            else  k = 0
 8          j ← j + 1
 9  w_n = k
10  return (w_n, ..., w_0)
```

Заметим, что при работе этого алгоритма всегда выполняются соотношения $u_j + v_j + k \le (b-1) + (b-1) + 1 < 2b$. Т.к. данный алгоритм выполняет ровно n итераций, длина его входа равна n, а длина выхода $n+1$, какие-либо существенные ускорения процедуры сложения предложить сложно.

С целью максимального использования аппаратных возможностей ЭВМ целесообразно выбирать в качестве основания системы счисления 2^m, где m — разрядность ЭВМ. При этом приведение по модулю и вычисление переноса осуществляются автоматически.

Заметим, что если число представлено в виде системы вычетов, то сложение может производиться путем покомпонентного сложения вектора вычетов (модулярная арифметика), т.е. имеет место изоморфизм. Это может быть использовано для распараллеливания вычислений (Single Instruction Multiple Data).

3.5.3 Умножение

Умножение “в столбик”

Рассмотрим реализацию умножения неотрицательных целых чисел $(u_{m-1}, \ldots, u_0)_b$ и $(v_{n-1}, \ldots, v_0)_b$ по основанию b. Следующий алгоритм формирует их произведение $(w_{m+n-1}, \ldots, w_0)_b$:

MULTIPLY(u, v, m, n)
1 $w_j \leftarrow 0, j = 0..m-1;$
2 $j \leftarrow 0;$
3 **while** $j < n$
4 **do if** $v_j > 0$
5 **then** $i \leftarrow 0; k \leftarrow 0;$
6 **while** $i < m$
7 **do** $t \leftarrow u_i * v_j + w_{i+j} + k;$
8 $w_{i+j} \leftarrow t \bmod b;$
9 $k \leftarrow \lfloor t/b \rfloor;$
10 $i \leftarrow i + 1;$
11 $w_{j+m} = k;$
12 **else** $w_{j+m} \leftarrow 0;$
13 $j \leftarrow j + 1;;$
14 **return** $(w_{m+n-1}, \ldots, w_0)$

На каждом шаге алгоритма умножения выполняются неравенства

$$0 \le t < b^2, 0 \le k < b.$$

На основе этих свойств можно оценить размер регистра, требуемый для хранения промежуточных значений. Если не производить обнуление выходного массива, то можно получить более общий алгоритм умножения с накоплением $w = u * v + w'$.

Быстрое умножение

Всякое целое число можно представить как значение многочлена $u(x) = \sum_{i=0}^{m-1} u_i x^i$ в точке b, где $u = (u_{m-1}, \ldots, u_1, u_0)_b$ и b — основание системы счисления. Поэтому задача умножения целых чисел может быть решена с использованием быстрых алгоритмов линейной свертки, в частности:

1. алгоритма Карацубы:
2. алгоритма Тоома-Кука;
3. на основе быстрого преобразования Фурье и теоремы о свертке;
4. алгоритма Шёнхаге-Штрассена.

Например, пусть $u = b^n U_1 + U_0, v = b^n V_1 + V_0$. Тогда (*алгоритм Карацубы*)

$$\begin{aligned} uv = & (b^{2n} + b^n)U_1V_1 + b^n(U_1 - U_0)(V_0 - V_1) + (b^n + 1)U_0V_0 = \\ = & U_0V_0 + b^{2n}U_1V_1 + b^n((U_0 + V_0)(U_1 + V_1) - U_0V_0 - U_1V_1). \end{aligned}$$

Эта процедура может быть применена рекурсивно. В этом случае число элементарных умножений асимптотически сходится к $O(n^{\log_2 3})$.

Пример 3.17.

$351 = 13 * 27 = 3 * 7 + 100 * 1 * 2 + 10 * ((1 + 3) * (2 + 7) - 3 * 7 - 1 * 2) = 21 + 100 * 2 + 10 * (36 - 21 - 2) = 21 + 200 + 130 = 351$

3.5.4 Деление

Основная сложность при реализации классического метода деления "в столбик" состоит в необходимости угадывать разряды частного. Этот процесс должен быть формализован. Прежде всего заметим, что деление m-разрядного числа на n-разрядное, $m > n$, сводится к последовательности делений $n + 1$-разрядных чисел u на n-разрядное число v, причем $0 \leq u/v < b$, где b — основание системы счисления. Таким образом, необходимо построить алгоритм для нахождения $q = \lfloor u/v \rfloor, u = (u_n, u_{n-1}, \ldots, u_0)_b, v = (v_{n-1}, \ldots, v_0)_b$.

Условие $u/v < b$ может быть переформулировано как $u/b < v$, т.е. $(u_n, u_{n-1}, \ldots, u_1)_b < (v_{n-1}, \ldots, v_0)_b$. q должно быть единственным целым числом, таким, что $0 \leq u - qv < v$. Попытаемся угадать q как

$$\hat{q} = \min\left(\left\lfloor \frac{u_n b + u_{n-1}}{v_{n-1}} \right\rfloor, b - 1\right). \tag{3.21}$$

Попытаемся оценить, насколько хорошо такое приближение.

Лемма 3.1. $\hat{q} \geq q$.

Доказательство. Т.к. $q \leq b - 1$, утверждение верно при $\hat{q} = b - 1$. В противном случае $\hat{q} \leq \frac{u_n b + u_{n-1}}{v_{n-1}} < \hat{q} + 1 \Rightarrow \hat{q}v_{n-1} \geq u_n b + u_{n-1} - v_{n-1} + 1$. Следовательно, $u - \hat{q}v \leq u - \hat{q}v_{n-1}b^{n-1} \leq u_n b^n + \ldots + u_0 - (u_n b^n + u_{n-1}b^{n-1} - v_{n-1}b^{n-1} + b^{n-1}) = u_{n-2}b^{n-2} + \ldots + u_0 - b^{n-1} + v_{n-1}b^{n-1} < v_{n-1}b^{n-1} \leq v$. Тогда $\hat{q}v > u - v \geq qv - v = (q-1)v$, т.е. $\hat{q} > q - 1$. □

Докажем теперь, что если $\hat{q}$ превышает q, то превышение незначительно.

Теорема 3.9. *Если* $v_{n-1} \geq \lfloor b/2 \rfloor$, *то* $\hat{q} - 2 \leq q \leq \hat{q}$

Доказательство. Предположим, что это не так, т.е. $\hat{q} \geq q + 3$. Тогда

$$\hat{q} \leq \frac{u_n b + u_{n-1}}{v_{n-1}} = \frac{u_n b^n + u_{n-1}b^{n-1}}{v_{n-1}b^{n-1}} \leq \frac{u}{v_{n-1}b^{n-1}} < \frac{u}{v - b^{n-1}}.$$

Случай $v = b^{n-1}$ невозможен, т.к. тогда $q = \hat{q}$. Т.к. $q + 1 > u/v$,

$$3 \le \hat{q} - q < \frac{u}{v - b^{n-1}} - \frac{u}{v} + 1 = \frac{u}{v}\left(\frac{b^{n-1}}{v - b^{n-1}}\right) + 1.$$

Тогда $\frac{u}{v} > 2\frac{v-b^{n-1}}{b^{n-1}} \ge 2(v_{n-1} - 1)$. Т.к. $b - 4 \ge \hat{q} - 3 \ge q = \lfloor u/v \rfloor \ge 2(v_{n-1} - 1)$, $v_{n-1} < \lfloor b/2 \rfloor$. □

Условие этой теоремы носит название условия нормализации. Его можно обеспечить, домножив делимое и делитель на $\lfloor b/(v_{n-1} + 1) \rfloor$.

Кроме того, можно показать, что если $\hat{q}v_{n-2} > b\hat{r} + u_{n-2}$, то $q < \hat{q}$, где $\hat{r} = u_n b + u_{n-1} - \hat{q}v_{n-1}$. В противном случае $q \in \{\hat{q}, \hat{q} - 1\}$.

Рассмотрим вычисление частного и остатка от деления $(u_{m+n-1}, \ldots, u_0)_b$ на $(v_{n-1}, \ldots, v_0)$.

DIVIDE(u, v, m, n)

```
 1  d ← ⌊b/(v_{n-1} + 1)⌋;
 2  (u_{m+n}, u_{m+n-1} ..., u_0)_b ← d * (u_{m+n-1} ..., u_0)_b
 3  (v_{n-1} ..., v_0)_b ← d * (v_{n-1} ..., v_0)_b
 4  j ← m
 5  while j ≥ 0
 6      do q̂ ← ⌊(u_{j+n} b + u_{j+n-1})/v_{n-1}⌋
 7         r̂ ← (u_{j+n} b + u_{j+n-1}) mod v_{n-1}
 8         if (q̂ = b) ∨ (q̂ v_{n-2} > b r̂ + u_{j+n-2})
 9           then q̂ ← q̂ − 1
10                r̂ ← r̂ + v_{n-1}
11         if (r̂ < b) ∨ (q̂ v_{n-2} > b r̂ + u_{j+n-2})
12           then q̂ ← q̂ − 1
13                r̂ ← r̂ + v_{n-1}
14         (u_{j+n} ..., u_j)_b ← (u_{j+n} ... u_j)_b − q̂ (v_{n-1} ... v_1 v_0)_b
15         if (u_{j+n} ... u_j)_b < 0
16           then NegFlag ← true
17                (u_{j+n} ... u_j)_b ← (u_{j+n} ... u_j)_b + b^{n+1}
18           else NegFlag ← false
19         q_j = q̂
20         if NegFlag = true
21           then q_j ← q_j − 1
22                (u_{j+n} ... u_j)_b ← (u_{j+n} ... u_j)_b + (0 v_{n-1} ... v_1 v_0)_b
23         j ← j − 1
24  return ((q_m ... q_1 q_0)_b, (u_{n-1} ... u_1 u_0)_b / d)
```

Некоторые фрагменты этого алгоритма выполняются очень редко, что затрудняет отладку.

3.5.5 Возведение в степень

Рассмотрим задачу эффективного вычисления $y = x^n, n \in \mathbb{N}$. Пусть $n = \sum_{i\geq 0} n_i 2^i, n_i \in \{0,1\}$. Тогда

$$y = x^{\sum_{i\geq 0} n_i 2^i} = \prod_{i\geq 0} x^{n_i 2^i} = \prod_{i\geq 0} \left(x^{2^i}\right)^{n_i}. \tag{3.22}$$

Заметим, что $x^{2^i} = x^{2^{i-1}} x^{2^{i-1}}$, а значение n_i указывает, должен ли сомножитель x^{2^i} учитываться при вычислении y. Таким образом, возведение в степень может быть выполнено за

$$\log_2(n) + \sum_{i\geq 0} n_i$$

операций умножения (*бинарный алгоритм возведения в степень*).

Двоичный метод возведения в степень не является оптимальным. Например, вычисление x^{15} требует 6 умножений, в то время как $z = x^3$ может быть вычислено с помощью 2 умножений, а $y = z^5$ — с помощью еще трех, т.е. всего необходимо 5 умножений. Таким образом, если $n = pq$, задача возведения в степень может быть разделена на две подзадачи, каждая из которых может быть решена двоичным методом. Как правило, этот метод несколько лучше двоичного, но не всегда (например, при $n = 33$).

3.6 Основные результаты

1. Алгоритм Штрассена перемножения матриц.
2. Алгоритмы Тоома-Кука и Карацубы вычисления свертки.
3. Алгоритм Винограда и границы сложности вычисления свертки.
4. Алгоритм Агарвала-Кули и итеративный метод вычисления свертки.
5. Алгоритмы Кули-Тьюки (Cooley-Tukey) и Гуда-Томаса БПФ.
6. Алгоритмы Блюстейна и Рейдера вычисления БПФ простой длины.
7. Применение свойств конечных полей для ускорения БПФ.
8. Алгоритм Шёнхаге-Штрассена линейной свертки.
9. Представление целых чисел в ЭВМ
10. Применение алгоритмов свертки для умножения целых чисел
11. Возведение целых чисел в степень.

Упражнения

1. Построить алгоритм шеститочечной циклической свертки двумя способами над вещественным и конечным полями.
2. Написать программу, генерирующую билинейные формы алгоритма Винограда перемножения многочленов по модулю заданного многочлена $\pi(x)$.
3. Доказать, что если $\pi_0, \ldots, \pi_{m-1}$ — базис поля $GF(2^m)$, то матрица $\begin{pmatrix} \pi_0 & \pi_0^2 & \ldots & \pi_0^{2^{m-1}} \\ \pi_1 & \pi_1^2 & \ldots & \pi_1^{2^{m-1}} \\ \ldots & & & \\ \pi_{m-1} & \pi_{m-1}^2 & \ldots & \pi_{m-1}^{2^{m-1}} \end{pmatrix}$ обратима и ее определитель равен 1.
4. Построить алгоритм Кули-Тьюки БПФ длины 8 над комплексным полем.
5. Построить алгоритм Гуда-Томаса БПФ длины 6 над комплексным полем.
6. Построить алгоритм Рейдера БПФ длины 7 над комплексным полем и полем $GF(2^3)$.
7. Построить алгоритм Ванга-Жу БПФ длины 7 над полем $GF(2^3)$.
8. Построить циклотомический алгоритм БПФ длины 5 и 15 над полем $GF(2^4)$.
9. Вычислить 156616*879521 с помощью алгоритма Карацубы.

3.7 Задания для курсовых работ

В рамках курсовой работы необходимо реализовать быстрый алгоритм решения некоторой задачи компьютерной алгебры. При реализации алгоритма необходимо использовать параллельную и векторизованную обработку данных. Допускается применение библиотек вычислительных примитивов, таких как Intel Performance Primitives Library и Intel Math Kernel Library. Пояснительная записка должна содержать:

1. Описание алгоритма;
2. Описание используемых вычислительных примитивов (НОД, умножение и т.п.) с указанием алгоритмов, использованных для их реализации;
3. Описание возможностей параллелизации алгоритма и ожидаемого ускорения;
4. Зависимость времени вычислений от размерности решаемой задачи.

Далее приведены варианты заданий. Подробное описание алгоритмов, предлагаемых к реализации, приведено в дополнительной литературе, работа с которой является обязательным элементом курсовой работы. Во всех случаях необходимо обеспечить работоспособность программы на данных любой размерности, в т.ч. превышающей разрядность процессора.

1. Разложение на множители целых чисел с помощью метода Диксона [20].
2. Разложение на множители целых чисел с помощью ρ-метода Полларда-Брента [20, 13].
3. Разложение на множители целых чисел с помощью $p-1$-метода Полларда [20, 13].
4. Реализация алгоритма нахождения дискретного логарифма в конечном поле $GF(p)$ ρ-методом Полларда, где p — простое число [17].
5. Нахождение дискретного логарифма в конечном поле $GF(2^m)$ ρ-методом Полларда [17].
6. Нахождение дискретного логарифма в конечном поле $GF(p)$ ρ-методом Сильвера-Полига-Хеллмана, где p — простое число [17].
7. Нахождение дискретного логарифма в конечном поле $GF(2^m)$ ρ—методом Сильвера-Полига-Хеллмана [17].
8. Нахождение дискретного логарифма в поле $GF(2^m)$ методом Эйдльмана-Меркля-Полларда [17].
9. Нахождение дискретного логарифма в поле $GF(2^m)$ методом Копперсмита [17].
10. Реализация алгоритма Видеманна решения разреженных систем линейных алгебраических уравнений над конечным полем [20].
11. Факторизация многочленов над полем $GF(p)$ с помощью алгоритма Берлекэмпа [20].
12. Факторизация многочленов над полем $GF(2)$ с помощью алгоритма Кантора-Зассенхауза [20].
13. Факторизация многочленов над кольцом целых чисел [20].

Предметный указатель

Литература

[1] Архитектуры и топологии многопроцессорных вычислительных систем / А. Богданов, В. Корхов, В. Мареев, Е. Станкова. — ИНТУИТ, 2004.

[2] *Афанасьев В. Б., Грушко И. И.* Алгоритмы БПФ для полей $GF(2^m)$ // Помехоустойчивое кодирование и надежность ЭВМ. — М.: Наука, 1987. — С. 33–55.

[3] *Ахо А., Хопкрофт Д., Ульман Д.* Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979. — 536 с.

[4] *Берлекэмп Э. Р.* Алгебраическая теория кодирования. — М.: Мир, 1971. — 477 с.

[5] *Блейхут Р.* Теория и практика кодов, контролирующих ошибки. — М.: Мир, 1986. — 576 с.

[6] *Блейхут Р.* Быстрые алгоритмы цифровой обработки сигналов. — М.: Мир, 1989. — 448 с.

[7] *Воеводин В., Воеводин В.* Параллельные вычисления. — 2002.

[8] *Габидулин Э. М., Афанасьев В. Б.* Кодирование в радиоэлектронике. — М.: Радио и связь, 1986. — 176 с.

[9] *Захарова Т. Г.* Вычисление преобразования Фурье в полях характеристики 2 // *Проблемы передачи информации*. — 1992. — Т. 28, № 2. — С. 62–76.

[10] *Кнут Д.* Искусство программирования. — М.: Вильямс, 2000. — Т. 2.

[11] *Мак-Вильямс Ф. Д., Слоэн Н. Д. А.* Теория кодов, исправляющих ошибки. — М.: Связь, 1979. — 744 с.

[12] *Трифонов П. В., Федоренко С. В.* Метод быстрого вычисления преобразования Фурье над конечным полем // *Проблемы передачи информации*. — 2003. — Т. 39, № 3. — С. 3–10.

[13] *Cohen H.* A course in computational algebraic number theory. — Springer, 1996.

[14] *E.Chu, George A.* Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms. — CRC Press, 2000.

[15] Intel 64 and IA-32 Architectures Optimization Reference Manual. — Intel, 2006. — November.

[16] *Knuth D. E.* The Art of Computer Programming. — Addison-Wesley, 1973. — Vol. 2.

[17] *Odlyzko A. M.* Discrete logarithms in finite fields and their cryptographic significance // Theory and Application of Cryptographic Techniques. — 1984. — Pp. 224–314. `citeseer.ist.psu.edu/odlyzko84discrete.html`.

[18] *Robison A.* n-bit unsigned division via n-bit multiply-add // Proceedings of 17th IEEE Symposium on Computer Arithmetic. — 2005. — Pp. 131– 139.

[19] Software Optimization Guide for AMD64 Processors. — AMD, 2005. — September.

[20] *von zur Gathen J., Gerhard J.* Modern Computer Algebra. — Cambridge University Press, 1999.

[21] *Wang Y., Zhu X.* A fast algorithm for the Fourier transform over finite fields and its VLSI implementation // *IEEE Journal on Selected Areas in Communications*. — 1988. — Vol. 6, no. 3. — Pp. 572–577.