Эффективные базовые операции линейной алгебры для решения больших разреженных линейных систем над конечными полями *

Д.А. Желтков[1]

Институт вычислительной математики РАН[1]

Во многих приложениях, таких как дискретное логарифмирование и факторизация больших чисел, необходимо решать большие разреженные линейные системы над конечными полями. В настоящее время наиболее популярны два метода: алгоритм Видемана-Копперсмита и алгоритм Ланцоша-Монтгомери.
В обоих методах наиболее затратной по времени является операция, связанная с умножением разреженной матрицы на вектор. Поэтому крайне важно выполнять эту операцию эффективно и параллельно. На параллельных машинах основным ресурсом обоих методов является размер блока.
Однако, с ростом размера блока сложность других операций (таких как умножение или обращение плотных матриц) быстро растет. Можно показать, что для больших размеров блока масштабируемость обоих методов существенно деградирует именно вследствие больших затрат на операции с плотными блоками. Для улучшения масштабируемости методов время, требуемое на выполнение базовых операций линейной алгебры над конечными полями, должно быть существенно сокращено.
В данной статье рассмотрен вопрос эффективной реализации базовых операций линейной алгебры как для конечного поля GF(2), так и для для простых конечных полей с большим число элементов. Эффективность реализации достигается за счет использования следующих подходов:
1) Алгоритмы быстрой арифметики для конечных полей.
2) Алгоритмы линейной алгебры меньшей вычислительной сложности.
3) Оптимизация промахов по кэшу.
4) Оптимизация параллельной OpenMP реализации на общей памяти.
5) Оптимизация обменов информацией между вычислительными узлами.

Ключевые слова: линейные системы над конечными полями, параллельные алгоритмы, базовые операции линейной алгебры над конечными полями

1. Введение

При решении задач дискретного логарифмирования или факторизации чисел для больших и рекордных чисел возникают сверхбольшие (порядка N от 10^6 до 10^9 и выше) разреженные (среднее число ненулевых элементов в строке p, как правило, от 100 до 1000) системы уравнений над конечными полями. В настоящее время наиболее распространенными методами для решения таких систем являются методы Ланцоша-Монтгомери [1] и Видемана-Копперсмита [2]. Оба метода выполняют $O(N)$ умножений разреженной матрицы на вектор, таким образом, их сложность – $O(pN^2)$.

Очевидно, что для сверхбольших систем время работы методов будет очень велико. Для его сокращения используются параллельные версии алгоритмов. Используется два ресурса параллельности:

- Параллельность умножения матрицы на вектор: разделение разреженной матрицы на s частей с одинаковым числом ненулевых элементов и распределение их по s узлам.

*Работа выполнена при финансовой поддержке Минобрнауки России, соглашение от 17 июня 2014 г. № 14.604.21.0034 (идентификатор соглашения RFMEFI60414X0034) в рамках ФЦП "Исследование и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014-2020 годы"

- Блочность: умножение матрицы на блок из k векторов, умножение на каждый вектор из блока происходит независимо на одном из k узлов.

Первый способ позволяет уменьшить время, затрачиваемое на арифметические операции при умножении матрицы на вектор, в s раз, кроме того, время, затрачиваемое на все остальные арифметически операции (связанные с операциями линейной алгебры над векторами) методов, либо может быть в s раз ускорено, либо крайне мало. Однако, время, требуемое для выполнения возникающих обменов данными между узлами, велико, и с ростом s падает не более, чем в $\sqrt{s}$ раз, а в большинстве практических реализаций – не уменьшается с ростом s.

Второй способ позволяет уменьшить в k раз время, затрачиваемое как на арифметические операции, так и на обмены данными (если они есть) при выполнении операции умножения разреженной матрицы на вектор. Однако, общая сложность арифметических операций, связанных с операциями с блоками векторов, растет в k раз, что приводит к тому, что, с учетом распараллеливания этих операций по k узлам, время, требуемое на их выполнение, не зависит от k.

Таким образом, оба подхода имеют недостатки, из-за которых уже на достаточно небольшом числе узлов (несколько десятков) методы перестают ускоряться при увеличении числа узлов. На практике используют комбинацию этих ресурсов параллельности. Время, затрачиваемое такими методами (если пренебречь членами, не оказывающими влияния при практически важных k и s [12]) на арифметические операции равно $O(\frac{pN^2}{ks} + \frac{N^2}{s})$, на обмены данными – $O(\frac{N^2}{k} + \frac{N^2}{ks})$.

Очевидно, что:

- При относительно малом числе узлов крайне важно эффективно выполнять операции, связанные с умножением разреженной матрицы на вектор, так как они являются доминирующими.

- С точки зрения параллельной производительности необходимо эффективно выполнять операции с блоками – если операции, связанные с блоками, в n раз быстрее, чем операция умножения матрицы на вектор, то ускорение на n узлах отличается от идеального не более, чем в два раза.

В данной статье рассмотрен вопрос эффективной реализации базовых операций линейной алгебры как для конечного поля GF(2), так и для для простых конечных полей с большим число элементов. Эффективность реализации достигается за счет использования следующих подходов:

1) Алгоритмы быстрой арифметики для конечных полей.
2) Алгоритмы линейной алгебры меньшей вычислительной сложности.
3) Оптимизация промахов по кэшу.
4) Оптимизация параллельной OpenMP реализации на общей памяти.
5) Оптимизация обменов информацией между вычислительными узлами.

2. Арифметика над конечными полями

Прежде всего, для эффективной реализации операций линейной алгебры над конечными полями необходима эффективная реализация арифметики над конечными полями.

Для поля GF(2) фактически единственная существующая арифметическая операция – это исключающее или (**xor**), поэтому специальная реализация арифметики в этом случае не требуется. Единственное замечание – в этом случае крайне полезно использование векторных наборов процессорных инструкций (например, для архитектуры *x86* это *SSE2, AVX, AVX2, AVX512*).

Случай больших конечных полей требует более тщательного рассмотрения. В арифметических операциях над большими конечными полями сложность вызывают, главным

образом, операции умножения по модулю простого числа. Операции обращения и деления, хоть и являются существенно более вычислительно сложными, на практике используются в очень небольших количествах, поэтому эффективность их реализации фактически не влияет на скорость работы операций линейной алгебры.

Для реализации умножения по модулю простого числа широко используется метод Монтгомери [3–5]. Для его работы необходимо перевести числа в специальный формат – формат Монтгомери. Как правило, числа конвертируют в этот формат в начале работы метода решения систем, все операции внутри метода производятся в данном формате, и в конце его работы числа возвращают в исходное представление. Особенностью операции умножения по модулю простого в формате Монтгомери является то, что метод не использует делений и множества ветвлений, в отличии от умножения по модулю простого числа в обычном формате. По сложности он приблизительно равен 2.5 обычным умножениям длинных чисел, без приведения по модулю простого числа.

В разделе 2.1 приведено описание реализации арифметики над большим простым полем по словам – наиболее быстрый вариант при реализации на CPU/GPU и прочих устройствах, работающих со словами, а не с битами. Также в подразделе 2.3 описан предлагаемый метод умножения длинных чисел в формате Монтгомери или исходном формате на короткие числа по модулю большого простого числа.

Отметим, что используемые базовые алгоритмы арифметики над большим простым полем являются оптимальными по количеству операций для практически важных размеров полей, используемых при решении сверхбольший разреженных систем (до 1024 бит, то есть до 16 слов размера 64 бита).

2.1. Арифметика над большими конечными полями

Будем рассматривать методы реализации арифметики по модулю большого простого числа, практически не затрагивая вопрос архитектурных оптимизаций. Тем не менее, рассматриваются только алгоритмы, работающие со словами, а не с битами, так как CPU (а также GPU и большинство ускорителей, кроме FPGA) созданы для работы со словами, а не с битами. Для работы с одиночным битом на таких устройствах требуется, как правило, больше операций, чем для работы со словом, при этом количество операций с битами в побитовых алгоритмах значительно больше, чем количество операций со словами в аналогичных алгоритмах по словам.

Для изложения методов введем следующие обозначения:

W — число бит в используемых машинных словах.

P — простое число по модулю которого выполняются операции.

N — число бит в числе P.

n — число слов в числе P, таким образом $(n-1)W < N \leqslant nW$.

P_{norm} — нормализованное P, то есть P сдвинутое побитово влево так, чтобы старший бит старшего слова стал ненулевым.

A^i — слово с номером i числа A. Слова будем нумеровать от 0, A^0 – младшее слово числа A.

Для реализации алгоритмов необходимо несколько дополнительных базовых операций со словами:

- Сложение слов с битом переноса.
- Вычитание слов с битом переноса.

- Умножение двух слов с сохранением результата в два слова.
- Деления числа, состоящего из двух слов, на число из одного слова с получением частного, состоящего из одного слова, и остатка.

На большинстве процессорных архитектур существуют машинные инструкции, выполняющие данные операции. Тем не менее, при необходимости не составляет труда написать архитектурно независимые реализации этих операций.

Отметим, что векторных процессорных инструкций, реализующих данные операции, для архитектуры *x86* не существует, поэтому на практике выгоднее не использовать векторные инструкции, а работать со словами размера 64 бита.

Реализация таких операций, как сложение и вычитание двух чисел по модулю простого числа, а также умножения слова на большое число, элементарна и не требует подробного описания.

2.2. Умножение длинных чисел

Умножение длинных чисел можно реализовывать либо наивным способом, аналогичным умножению «столбиком», либо методом Карацубы (или другими быстрыми методами) [8].

При умножении «столбиком» мы последовательно умножаем слова одного из множителей на второй множитель, а затем складываем с соответствующим сдвигом. При этом выполняется n^2 умножений слов и примерно $2n^2 + n$ сложений с битами переноса.

Метод Карацубы разбивает число на две почти равные части и рекурсивно запускает умножение для чисел таких размеров.

Для упрощения изложения пусть n – четное. Обозначим L_A, L_B, H_A, H_B числа длины $\frac{n}{2}$ слов такие что $L_A^i = A^i$, $L_B^i = B^i$, $H_A^i = A^{i+\frac{n}{2}}$, $H_B^i = B^{i+\frac{n}{2}}$. Пусть $C_L = L_A L_B$, $C_H = H_A H_B$, $C_{HL} = (L_A + H_A)(L_B + H_B)$.

Легко проверить, что результат $C = C_L + 2^{\frac{nW}{2}}(C_{HL} - C_L - C_H) + 2^{nW} C_H$.

Естественным является использование метода Карацубы до тех пор, пока он эффективнее, а затем переходить на классическое умножение. Легко получить, что алгоритм Карацубы эффективнее по числу операций уже при $n = 8$, при условии равенства сложностей операций сложений с битами переноса и умножений двух слов с записью результата в число, состоящее из двух слов.

Однако, в силу большей алгоритмической сложности алгоритма Карацубы он хуже оптимизируется средствами компиляторов и на практике выигрыш без написания архитектурно оптимизированной реализации начинается только с $n \approx 20$. При решении разреженных систем над конечными полями, как уже упоминалось выше, в настоящее время используются числа длиной до 16 слов (рекордное факторизованное RSA-число имеет длину 12 слов), поэтому использование метода Карацубы (и, тем более, более быстрых асимптотически методов) не требуется.

2.2.1. Деление и вычисление остатка от деления по модулю

Деление и вычисление остатка по модулю выполняется следующим образом (аналогично реализуется, при необходимости, алгоритм деления числа из двух на слово):

1. Вначале выполняется нормализация делителя, то есть сдвиг влево на минимально возможное число бит такое, что старший бит старшего слова был равен единице. Обозначим полученное число за D.
2. Делимое сдвигается влево на то же число бит. Обозначим это число за A.
3. В полученном числе A выбираются старшие $n+1$ (где n – число слов в делителе) слов из которых формируется число соответствующей длины. Обозначим его за C.

4. Делим с остатком старшие два слова этого число на старшее слово нормализованного делителя: $(C^n C^{n-1}) = D^{n-1} * d + r$.

5. Число $R = (r, C^{n-2})$ являлось бы остатком от деления первых трех слов, если бы D^{n-2} было бы нулевым.

6. Если $d * D^{n-2} \leqslant R$, то d – результат деления нацело старших трех слов числа C на старшие два слова числа D. В ином случае, d последовательно уменьшать на единицу, при этом необходимо увеличивать число R на D^{n-1}. Отметим, что всего уменьшить d возможно не более, чем на два, так как старший бит числа D^{n-1} равен единице, и более чем однократное прибавлении его к R гарантирует переполнение, что также означает, что верное значение d достигнуто.

7. Теперь d – верный результат целочисленного деления первых трех слов числа C на первые два слова числа D. Вычтем из C число D, умноженное на d. При этом может возникнуть переполнение, т. к. мы могли ошибиться в d на единицу. В этом случае следует уменьшить d на единицу, а к C прибавить D.

8. d записываем в соответствующее слово частного.

9. Отметим, что теперь старшее слово C обязательно нулевое. Сдвинем число влево на одно слово, а в качестве последнего добавим старшее из слов, которые мы еще не использовали из числа A.

10. Продолжаем делать аналогично описанному в предыдущих пунктах до тех пор, пока не использовали все слова из A.

11. Остатком от деления будет последнее полученное C, сдвинутое вправо на столько же бит, на сколько сдвигался делитель влево для нормировки.

Отметим, на основе данного алгоритма легко реализовать быстрый метод умножения длинного числа на короткое число (состоящее из одного слова) по модулю простого числа. При этом в данной операции необходимо использовать только два старших слова нормализованного простого числа, а битовые сдвиги использовать только для трех старших чисел делимого.

Имея алгоритм деления по модулю легко реализовать нахождение обратного элемента по модулю простого числа с помощью алгоритма Евклида.

2.2.2. Умножение по модулю простого числа

Для умножения по модулю можно использовать умножение длинных чисел и деление по модулю простого числа. Однако, последний алгоритм содержит много делений и ветвлений, что оказывает существенное влияние на реальную производительность. Вместо этого следует использовать метод Монтгомери [3–5].

В методе Монтгомери все числа переводятся в формат Монтгомери: $M(A) = A * 2^{nW} \mod P$. Отметим, что так как числа в формате Монтгомери остаются в поле по модулю простого числа, то все операции, кроме умножения и обращения (и деления, то есть умножения на обратное) остаются теми же. Поэтому для программы, использующей числа по модулю большого простого числа, можно перевести все входные данные в формат Монтгомери и работать в нем, после для результата работы программы выполнив обратное преобразование.

Основным элементом в алгоритме Монтгомери является умножение в формате Монтгомери. Обозначим его за $*_M$. $A *_M B = \frac{AB}{2^{nW}} \mod P$. Таким образом, $M(A) *_M M(B) = M(AB)$.

Для того, чтобы перевести A в формат Монтгомери достаточно умножить A на $2^{2nW} \mod P$ с помощью $*_M$, то есть $M(A) = A *_M (2^{2nW} \mod P)$.

Обратное преобразование выполняется путем умножения на 1 с помощью $*_M$, то есть $M^{-1}(A) = A *_M 1$.

Опишем реализацию по словам $*_M$. Для начала введем слово p_I следующим образом: $2^W(\frac{1}{2^W} \mod P) - Pp_I = 1$.

Теперь изложим по алгоритм $A *_M B$:

1. Умножим A на B с помощью умножения длинных чисел. Получим число C длины $2n$.
2. Умножим C^0 на p_I, получим число t из двух слов.
3. Прибавим к C число P, умноженное на t. Т. к. $2^W(\frac{1}{2^W} \mod P) - Pp_I = 1$ получим, что $C^0 + P * t = 2^W(\frac{1}{2^W} \mod P)C^0$. Таким образом, последнее слово числа C станет нулевым. Теперь можно разделить на 2^W.

   Отметим, что так как результат на необходим по модулю P, то можно в любой момент добавлять и вычитать P, умноженное на любое число. Также заметим, что результат произведения старшего слова числа t на P не влияет на обнуление младшего слова C, поэтому на практике нужно умножать не все t на P, а только его младшее слово.
4. Выполняем обнуление младшего слова P и деления на 2^W n раз.
5. В результате получаем число длины n и бит переноса. Легко понять, что $0 \leqslant C < 2P$. Поэтому если $C < P$ в качестве результата используем C, в ином случае $C - P$.

2.3. Умножение короткого числа на длинное число по модулю простого числа

Как уже было отмечено выше (подраздел 2.2.1), с помощью умножения слова на длинное число и вычисления остатка по модулю легко реализовать быстрый метод умножение короткого числа на длинное число по модулю простого числа. Это крайне важно, так как разреженные матрицы, возникающие при дискретном логарифмировании или разложении чисел на множители, состоят из коротких чисел.

Естественно, желательно обрабатывать операции умножения коротких чисел на длинные по модулю простого числа быстрее, чем умножение длинных чисел. Кроме того, использование умножения Монтгомери для такой операции является не лучшим выбором, так как после перевода короткого числа в формат Монтгомери оно перестанет быть коротким. Переводить длинное число из формата Монтгомери в обычный формат, быстро умножать в обычном формате, и переводить обратно в формат Монтгомери тоже невыгодно.

Однако, заметим, что:

$$M(A * B) = A * B * 2^{nW} \mod P = A * M(B) \mod P$$

Таким образом, если мы хотим выполнить умножение числа A, хранящегося в обычном формате, на число B, хранящееся в формате Монтгомери с получением результата $A * B$, хранящегося в формате Монтгомери, нам необходимо только лишь умножить A на $M(B)$ по модулю простого числа.

Если число A – короткое, то умножение будет выполнено быстро.

Отметим также, что при совсем маленьких A (на практике – примерно до 4 при $n = 8$) выгоднее использовать A раз сложения по модулю простого числа.

3. Базовые операции линейной алгебры над конечными полями

Базовые операции линейной алгебры, требуемые для решения разреженных систем над конечными полями, делятся на два типа – плотные (умножение блоков на матрицу, вы-

числение билинейных форм) и разреженные (умножение разреженной матрицы на вектор). Плотные операции существенно отличаются для случая GF(2) и случая большого конечного поля, в то время как разреженные практически совпадают, за исключением небольших деталей.

Плотные операции в алгоритмах Ланцоша-Монтгомери и Видемана-Копперсита устроены таким образом, что практически не требуют пересылок, поэтому в данной работе вопрос их организации не затрагивается, все плотные операции считаются локальными, происходящими в рамках одного узла. Основными идеями эффективной реализации плотных операций являются алгоритмы линейной алгебры меньшей вычислительной сложности и оптимизация числа промахов по кэшу. Отметим, что распараллеливание на общей памяти для плотных операций элементарно – легко разделить матрицы на блоки так, чтобы операции записи были независимы, и, соответственно, синхронизация потоков не требовалась.

При умножении разреженной матрицы на вектор пересылки играют существенную роль, и поэтому данная операция рассматривается как с точки зрения локальных арифметических операций, так и с точки зрения обменов информацией. Основными идеями эффективной реализации данной операции являются оптимизация промахов по кэшу, оптимизация OpenMP параллелизма, оптимизация числа пересылок.

3.1. Ускоренное вычисление скалярного произведение над большим конечным полем

Пусть $A = (A_1, \ldots, A_k)$ и $B = (B_1, \ldots, B_k)$ – вектора чисел в формате Монтгомери. Нам необходимо вычислить $C = \sum_{i=1}^{k} A_i *_M B_i \mod P$.

Распишем подробнее:

$$\begin{aligned} C &= \sum_{i=1}^{k} A_i *_M B_i \mod P = \\ &= \sum_{i=1}^{k} (\frac{A_i B_i}{2^n W} \mod P) \mod P = \\ &= \frac{\sum_{i=1}^{k} A_i B_i}{2^n W} \mod P \end{aligned}$$

Таким образом, вместо того, чтобы делить каждое из слагаемых на 2^{nW} по модулю P можно вычислить их сумму, разделить на 2^{nW} по модулю P так же, ка это делается в алгоритме Монтгомери. После этого, в отличии от того, что происходит при умножении двух чисел в формате Монтгомери, результат будет принадлежать не полуинтервалу $[0, 2P)$, а полуинтервалу $[0, (k+1)P)$.

Пусть возникшее переполнение не превышает одного слова. Вычисление остатка по модулю от делителя, большего на одно слово делителя – быстрая операция, выполняемая за $O(n)$.

Отметим, что в результате число переполнится не более чем на $\log_2(k+1)+1$ бит. Если требовать, чтобы переполнение было всего на одно слово, необходимо, чтобы $\log_2(k+1)+1$ было меньше W. Поэтому надо разбить скалярное произведение на сумму скалярных произведений, в каждом из которых не более чем по 2^W элементов. На всех современных системах размер слова, с которым они могут работать не меньше 32 бит, в то же время, как правило, при вычислении скалярных произведений в практических задачах $k << 10^9$. Поэтому, как правило, разбивать на сумму меньших скалярных произведений не требуется.

Так как в операции $*_M$ приведение требует примерно половины операций, а умножения значительно медленнее сложений, этот прием позволяет ускорить вычисление скалярных

произведений почти в два раза уже для небольших k. Более того, очевидно, что вычисление такого скалярного произведения для больших k фактически равно по сложности вычислению скалярного произведения для просто длинных чисел, а не по модулю простого числа!

Отметим, что аналогичным образом можно поступать и для чисел не в формате Монтгомери, выполняя вычисление остатка по модулю простого числа лишь один раз, в самом конце, а не при каждом умножении. При этом для больших k сложности будут фактически одинаковы.

3.2. Прием Винограда для умножения матриц над большим простым полем

Полезным приемом при умножении матриц над большим простым полем является метод Винограда [6, 7]. Пусть матрицы A и B являются матрицами над большим простым полем (в обычном формате или формате Монтгомери). Матрица A размера $m \times k$, матрица B – $k \times n$. Пусть, для простоты изложения, k – четное.

Вычислим $A_i = \sum_{l=1}^{\frac{k}{2}} A_{i,2l-1} *_M A_{i,2l}$. Отметим, что это можно вычислять с помощью быстрых скалярных произведений. Аналогично вычислим $B_j = \sum_{l=1}^{\frac{k}{2}} B_{2l-1,j} *_M B_{2l,j}$.

Легко проверить, что:

$$C_{i,j} = \sum_{l=1}^{\frac{k}{2}} (A_{i,2l-1} + B_{2l,j}) *_M (A_{i,2l} + B_{2l-1,j}) - A_i - B_j$$

Разумеется, стоит использовать быстрые скалярные произведения. При том, что по сравнению с классическим алгоритмом используется примерно в полтора раза больше сложений, умножений используется меньше почти в два раза.

Так как сложность умножений заметно выше сложности сложений в случае большого простого поля, это позволяет ускорить матриц умножение почти в два раза, по сравнению с умножением, использующим только ускоренные скалярные произведения, и почти в 4 раза, по сравнению с наивной реализацией умножения.

3.3. Обращение матриц с помощью приема Винограда

Прием Винограда может быть применен и для обращения матриц.

Покажем на примере LU разложения для матрицы A. Пусть $L_k = [l_1 \dots l_k]$ – первые k столбцов матрицы L, $U_k = [u_1 \dots u_k]^T$ – первые k строк U. Обозначим как:

- $A_{[k+1]}$ – столбец с номером $k+1$ матрицы A.
- $A^{[k+1]}$ – строка с номером $k+1$ матрицы A.
- $[L_k]^{[k+1]}$ – строка с номером $k+1$ матрицы L_k.
- $[U_k]_{[k+1]}$ – столбец с номером $k+1$ матрицы U_k.

Тогда операции, выполняющиеся при LU разложении можно записать следующим образом:

$$l_{k+1} = A_{[k+1]} - L_k [U_k]_{[k+1]}$$

$$u_{k+1} = \frac{A^{[k+1]} - [L_k]^{[k+1]} U_k}{A_{k+1,k+1} - [L_k]^{[k+1]} [U_k]_{[k+1]}}$$

Легко понять, что необходимые для метода Винограда суммы для матриц L_{k+1} и U_{k+1} быстро пересчитываются через их значения для L_k и U_k, соответственно. При этом общее число операций, затраченных на это, равно числу операций для вычисления этих сумм для матриц L и U и зависит от размера матрицы A квадратично. В то же время, число умножений, требуемых для LU разложения, сокращено в 2 раза.

Получение обратной матрицы из LU разложения может быть получено аналогичным образом.

Отметим, что при обращении методом Винограда возможно использование быстрых скалярных произведений, таким образом, итоговый алгоритм до 4 раз быстрее наивного.

3.4. Оптимизация промахов по кэшу при умножении матриц над большим простым полем

В случае большого простого поля оптимизация промахов по кэшу является довольно очевидной и похоже на оптимизацию промахов по кэшу для действительных чисел: матрицы иерархически разбиваются на, по возможности, почти квадратные подматрицы такого размера, все элементы 2 подматриц соответствующего уровня иерархического разбиения помещались в соответствующий кэш. Например, при наличии кэша только одного уровня матрицы просто разбиваются на блоки необходимого размера. При наличии кэша 2 уровней – сначала разбиваются на блоки, помещающиеся в кэш второго уровня, а потом каждый такой блок – на блоки, помещающиеся в кэш первого уровня.

Далее используется иерархическое блочное умножение – при умножении блоков верхнего уровня мы лишь копируем их во временный массив соответствующего им размера и выполняем блочное умножение более низкого уровня. Непосредственно вычисления происходят на нижнем уровне.

3.5. Оптимизация промахов по кэшу для алгоритма «четырех русских»

Умножение матриц над полем GF(2) в случае когда биты в слова в обеих матрицах упакованы по строкам (или по столбцам) эффективно выполняется с помощью метода «четырех русских» [9]. Так как данный алгоритм быстрый, для него оптимизация промахов по кэшу крайне критична. Однако, в данном случае она нетривиальна – в связи с особенностями алгоритма разбиение на блоки, близкие к квадратным, неэффективно: для того, чтобы данные помещались в кэш длина столбца в блоке оказывается невелика (не более 512 при размере кэша 32 Кб и использовании 64-битных слов), в то же время, для умножения столбца блок требуется вычислить или загрузить из памяти 8 таблиц из 256 элементов, и для каждого столбца эти таблицы различны.

Вместо этого разбиение производится на прямоугольные блоки размера $l \times 1$ для первого множителя и результата, второй разбивается на соответствующие блоки размеров $64 \times v$, где v – число слов в векторе для поддерживаемых процессором векторных инструкций. Размеры блоков выбираются такими, чтобы $l(1+v)+256v$ слов помещались в кэш первого уровня.

Далее умножение выполняется блочно, для каждого умножения блоков используется «алгоритм 4 русских».

3.6. Оптимизация промахов по кэшу для алгоритма Копперсмита

Алгоритм Копперсмита [10] позволяет быстро вычислять произведение матриц над полем GF(2), в которых биты упакованы в слова по разным направлениям (например, у первого множителя по столбцам, а у второго – по строкам). Это крайне полезно, например, при вычислении произведения вида X^TY. Алгоритм схож с методом «четырех русских», и оптимизация промахов по кэшу для него тоже крайне критична и нетривиальна, использо-

вание квадратных блоков приводит к росту числа вспомогательных операций, аналогично методу «четырех русских».

В данном случае также следует разбивать на прямоугольные блоки – первый множитель на блоки размера $1 \times l$, второй – на блоки размера $l \times v$, результат – на блоки $64 \times v$, где v – число слов в векторе для поддерживаемых В связи с большей вычислительной сложностью вспомогательных операций в данном методе, на большинстве современных архитектур блоки необходимо помещать в кэш второго уровня, чтобы этих операций не становилось слишком много. Таким образом, l выбирается таким, что $l(1 + v) + 256v$ слов помещаются в кэш второго уровня.

3.7. Умножение разреженной матрицы на вектор

Разреженная матрица должна быть распределена по s узлам так, чтобы на каждом было приблизительно одинаковое число элементов. Хранящиеся на каждом узле матрицы должны быть непересекающимися подматрицами исходной матрицы.

Для оптимизации промахов по кэшу можно использовать кэш-независимый формат хранения матрицы [11] на каждом узле. Кэш-независимый формат – это такой порядок хранения элементов матрицы, что хранящиеся подряд элементы находятся и в близких строках, и близких столбцах. Простейший кэш-независимый формат можно описать следующим образом: разобьем матрицу A на 4 блока $A_{11}, A_{12}, A_{21}, A_{22}$ примерно одинаковых размеров. Результирующий порядок элементов матрицы будет следующим: сначала элементы блока A_{11}, затем A_{12}, затем A_{22}, затем A_{21}. Элементы внутри блоков упорядочены аналогичным образом.

Распределение по узлам возможно следующими способами:

1. Каждый узел хранит блок, состоящий из столбцов исходной матрицы.
2. Каждый узел хранит блок, состоящий из строк исходной матрицы.
3. Каждый узел хранит блок близкий к квадратному, порядка примерно $\frac{N}{\sqrt{s}}$.

Последний вариант оптимален с точки зрения пересылок, их в этом случае за одно умножение матрицы на вектор будет $O(\frac{N}{\sqrt{s}}$. Однако, существование такого разбиения с примерно равным числом элементов в каждом блоке не всегда возможно и этот способ хранения требует нетривиального метода разделения матрицы.

При первом варианте вычисления можно организовать или так, чтобы было $O(N \log_2 s)$ пересылок и $O(\frac{pN}{s})$ операций, или так, что число пересылок $O(N)$ и $O(\frac{pN}{s} + N)$ операций.

Во втором варианте число пересылок равно $O(N)$, а число операций – $O(\frac{pN}{s})$. Таким образом, этот вариант предпочтительней. Более того, этот вариант лучше первого с точки зрения распараллеливания на общей памяти – если разрезать матрицу сразу на такое число блоков, чтобы на каждый поток приходился один блок, записи в память, производимые каждым потоком, независимы. Таким образом, синхронизация потоков не будет требоваться, и достигается оптимальная параллельная производительность на общей памяти.

4. Экспериментальные исследования

Приведём результаты исследования ускорения, получаемого на практике при использовании описанных выше методик. Рассмотрим на примере умножения плотных матриц над большим конечным полем (результаты для обращения матриц над большим конечным полем аналогичны), метода «четырех русских» над полем GF(2) (результаты для метода Копперсмита аналогичны) и умножения разреженной матрицы на вектор над большим простым полем (результаты для поля GF(2) аналогичны).

Все эксперименты производились на процессоре Intel Core i5-4440S.

4.1. Умножение плотных матриц над большим простым полем

Рассмотрим умножение двух матриц порядка 1024 над простым полем с простым числом размера 512 бит, т. е. 8 64-битных слов.

Протестированы следующие реализации:

- Наивная реализация умножения с использованием арифметических операций в формате Монтгомери из библиотеки OpenSSL (OpenSSL naive).
- Наивная реализация с использованием арифметики, реализованной автором (Naive).
- Реализация, использующая быстрые скалярные произведения (FDP).
- Реализация, использующая приём Винограда (FDP + W).

Для последней реализации также произведено тестирование на различном числе потоков - $1, 2, 4$. Результаты приведены в таблице 1.

Таблица 1. Умножение плотных матриц порядка 1024 над полем из $\approx 2^{512}$ элементов

	OpenSSL naive	Naive	FDP	FDP+W 1	FDP+W 2	FDP+W 4
Время, с	440	275	139	70.6	35.4	18.2

4.2. Умножение плотных матриц над полем GF(2)

Рассмотрим следующие реализации:

- Метод «четырёх русских» без кэш-оптимизаций (FR).
- Метод «четырёх русских» с кэш-оптимизациями (CO FR).
- Метод «четырёх русских» с кэш-оптимизациями и AVX инструкциями (CO FR AVX).
- Метод «четырёх русских» с кэш-оптимизациями и AVX2 инструкциями (CO FR AVX2).

В качестве примера использовалось умножение двух квадратных матриц порядка $2^{15} = 32768$, тестирование всех алгоритмов производилось на 4 ядрах. Результаты приведены в таблице 2.

Таблица 2. Умножение плотных матриц порядка 2^{15} над полем GF(2)

	FR	CO FR	CO FR AVX	CO FR AVX2
Время, с	53	19	9.5	5.2

4.3. Умножение разреженной матрицы на вектор над большим простым

Эксперименты производились для матрицы порядка $2*10^6$ с 84 ненулевыми элементами в строке над полем из $\approx 2^{512}$ элементов.

Тестировались следующие алгоритмы:

- Умножение матрицы в CSR формате на вектор (CSR).

- Умножение матрицы в кэш-независимом формате на вектор (CI).
- Умножение матрицы в кэш-независимом формате на вектор, с использованием 4 потоков без разбиения матрицы на части (CI 4).
- Умножение матрицы в кэш-независимом формате на вектор, с использованием 4 потоков с разбиением матрицы на блоки столбцов (CI 4 Col).
- Умножение матрицы в кэш-независимом формате на вектор, с использованием 4 потоков с разбиением матрицы на блоки строк (CI 4 Row).

Результаты приведены в таблице 3.

Таблица 3. Умножение разреженной матрицы порядка $2 * 10^6$ на вектор над полем из $\approx 2^{512}$

	CSR	CI	CI 4	CI 4 Col	CI4 Row
Время, с	46	24	11	8.2	7.1

5. Заключение

Предложенные эффективные реализации позволяют ускорить операции линейной алгебры, использующиеся при решении систем над конечными полями в $2 - 4$ и более раз. Это позволяет существенно улучшить параллельное ускорение для методов решения таких систем.

Литература

1. Montgomery P.L. A block Lanczos algorithm for finding dependencies over GF(2). //Advances in cryptology—EUROCRYPT'95. 1995. P. 106-120.
2. Coppersmith D. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. //Mathematics of Computation. 1994. Vol. 62. No. 205. P. 333-350.
3. Montgomery P.L. Modular multiplication without trial division. // Mathematics of computation. 1985. Vol. 44, No. 170. P. 519–521.
4. Menezes A., Van Oorschot P., Vanstone S. Handbook of applied cryptography. CRC press, 1996.
5. Koç Ç., Acar T., Kaliski B. Analyzing and comparing Montgomery multiplication algorithms. // IEEE Micro. 1996. Vol. 16, No. 3. P. 26–33.
6. Anderson N., Manley D. 2D Object Detection and Recognition: models, algorithms and networksA matrix extension of Winograd's inner product algorithm. // Theoretical computer science. 1994. Vol. 131, No. 2. P. 475–477.
7. Winograd S. A new algorithm for inner product. // IEEE Transactions on Computers. 1968. Vol. 100. No. 7. P. 693–694.
8. Карацуба А.А., Офман Ю.П. Умножение многозначных чисел на автоматах. // Доклады академии наук СССР. 1961. № 2 (145). с. 293-294.

9. Арлазаров В.Л., Диниц Е.А., Кронрод М.А., Фараджев И.А. Об экономном построении транзитивного замыкания ориентированного графа. // Доклады АН СССР. № 3(194), с. 487-488.

10. Coppersmith D. Solving linear equations over $GF(2)$: Block Lanczos algorithm. // Linear Algebra Appl. 1993. Vol. 193. P. 33–60.

11. Замарашкин Н.Л. Алгоритмы для разреженных систем линейных уравнений в GF(2). Учебное пособие. Издательство Московского Университета, 2013.

12. Замарашкин Н.Л., Желтков Д.А. Блочный метод Ланцоша-Монтгомери с малым количеством обменов // Суперкомпьютерные дни в России, 2016

Effective basic linear algebra operations for solution of large sparse linear systems over finite fields

D.A. Zheltkov[1]

INM RAS[1]

In various applications such as discrete logarithm computations and large number factorization problems the solutions of large sparse linear systems over finite fields are required. Nowadays two methods are the most popular to solve these problems: Wiedemann-Coppersmith algorithm and Lanczos-Montgomery algorithm. For both methods the part related to sparse matrix by vector multiplication is usually the most time-consuming. So it is vital to perform this operation in efficient and parallel way. On parallel machines for both methods the main resource is a block size. But as block size grows the complexity of other operations (such as dense matrix multiplication or dense matrix inversion) grows fast. Moreover, it can be shown that for large block sizes the real bottleneck of parallel scalability related with both methods is the complexity of basic linear algebra operations with blocks. To improve scalability of these methods the time needed for basic linear algebra operations over finite fields should be significantly reduced.

In this paper an efficient implementations of basic linear algebra operations are considered for the finite field GF(2) and for the prime finite fields with huge number of elements. The efficiency of the implementation is provided in several ways: 1) Fast arithmetic for large finite fields. 2) Algorithms with lower complexity. 3) Cache miss optimization. 4) Optimization of OpenMP shared memory parallel implementation. 5) Data stream optimization.

Keywords: linear systems over finite fields, parallel algorithms, basic linear algebra over finite fields

References

1. Montgomery P.L. A block Lanczos algorithm for finding dependencies over GF(2) //Advances in cryptology—EUROCRYPT'95. 1995. P. 106-120.

2. Coppersmith D. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm //Mathematics of Computation. 1994. Vol. 62. No. 205. P. 333-350.

3. Montgomery P.L. Modular multiplication without trial division. // Mathematics of computation. 1985. Vol. 44, No. 170. P. 519–521.

4. Menezes A., Van Oorschot P., Vanstone S. Handbook of applied cryptography. CRC press, 1996.

5. Koç Ç., Acar T., Kaliski B. Analyzing and comparing Montgomery multiplication algorithms. // IEEE Micro. 1996. Vol. 16, No. 3. P. 26–33.

6. Anderson N., Manley D. 2D Object Detection and Recognition: models, algorithms and networksA matrix extension of Winograd's inner product algorithm. // Theoretical computer science. 1994. Vol. 131, No. 2. P. 475–477.

7. Winograd S. A new algorithm for inner product. // IEEE Transactions on Computers. 1968. Vol. 100. No. 7. P. 693–694.

8. Karatsuba A., Ofman Y. Multiplication of multidigit numbers on automata. // Doklady Physics. 1963. № 7. с. 595-596.

9. Arlazarov V., Dinic E., Kronrod M., Faradzev I. On economical construction of transitive closure of a directed graph, Dokl. Akad. Nauk. Vol. 194 No. 11 P. 487

10. Coppersmith D. Solving linear equations over $GF(2)$: Block Lanczos algorithm. // Linear Algebra Appl. 1993. Vol. 193. P. 33–60.

11. Zamarashkin N.L. Algoritmy dlya razrezhennykh sistem lineinykh uravneniy v GF(2). Uchebnoe posobie [Algorithms for systems of linear equations over GF(2). Study guide]. Publishing of the Lomonosov Moscow State University, 2013.

12. Zamarashkin N.L., Zheltkov D.A. Block Lanczos algorithm with reduced number of data exchanges// Russian Supercomputing Days, 2016