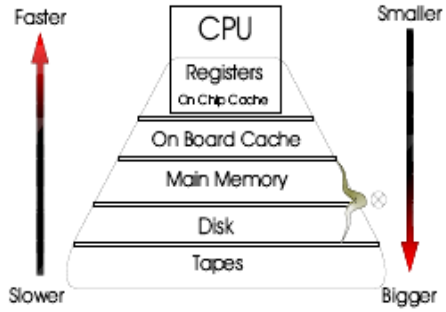
В.С. Лапонин

КЭШ-НЕЗАВИСИМЫЕ АЛГОРИТМЫ

Введение

Память современных вычислительных систем имеет очень сложную иерархию, которая представлена ниже на рисунке:



Хорошо известно, что чем дальше память располагается от процессора, тем её больше, но тем она медленнее. Кэш центрального процессора разделён на несколько уровней. Кэш-память уровня N+1, как правило, больше по размеру и медленнее по скорости обращения и передаче данных чем кэш-память уровня N.

Самой быстрой памятью является кэш первого уровня — L1-cache. По сути, она является неотъемлемой частью процессора, поскольку расположена на одном с ним кристалле и входит в состав функциональных блоков. L1 кэш работает на частоте процессора, и, в общем случае, обращение к нему может производиться каждый такт (зачастую является возможным выполнять даже несколько чтений/записей одновременно). Латентность доступа обычно равна 2 – 4 тактам ядра. Объём обычно невелик — не более 128 Кбайт.

Вторым по быстродействию является L2-cache — кэш второго уровня. Обычно он расположен либо на кристалле, как и L1, либо в непосредственной близости от ядра. Объём L2 кэша от 128 Кбайт до 1 – 12 Мбайт. В современных многоядерных процессорах кэш второго уровня, находясь на том же кристалле, является памятью раздельного пользования — при общем объёме кэша в 8 Мбайт на каждое ядро приходится по 2 Мбайта. Обычно латентность L2 кэша, расположенного на кристалле ядра, составляет от 8 до 20 тактов ядра.

Кэш третьего уровня наименее быстродействующий и обычно расположен отдельно от ядра ЦП, но он может быть очень внушительного размера : более 32 Мбайт. L3 кэш медленнее предыдущих кэшей, но всё равно значительно быстрее, чем оперативная память.

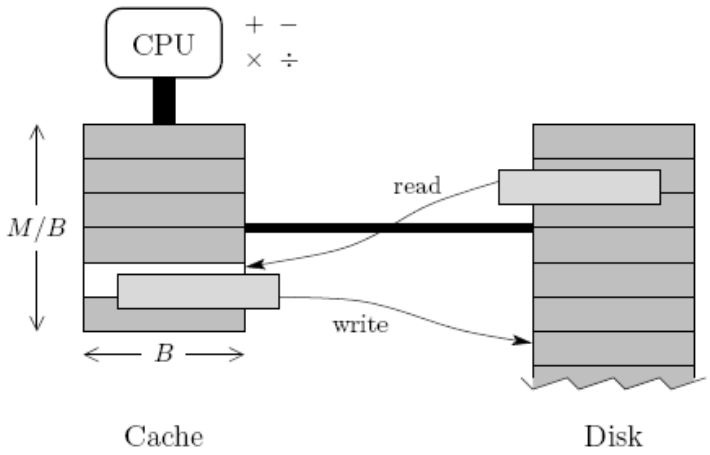
У каждой ЭВМ своя архитектура и свои размеры кэш-памяти, поэтому написать алгоритм, который будет одинаково эффективно использовать кэш-память на всех ЭВМ достаточно проблематично. В данной работе рассматривается простейшая модель с двумя уровнями памяти, для которой предлагается два кэш-независимых алгоритма.

1.Модель с двумя уровнями памяти

Для начала рассмотрим стандартную модель с двумя уровнями памяти, между которыми происходит блочный обмен. Эта модель известна как модель с внешней памятью. Стандартное описание этой модели можно найти в статье Аггарвала и Виттера[1], которая анализирует время на пересылки в памяти на задачах сортировок. Ранее Флойд[2] проанализировал цену пересылок в памяти на задачах транспонирования матриц.

Данная модель подразумевает, что у компьютера есть 2 уровня памяти:

1. Кэш, который расположен рядом с процессором, имеет быструю связь с процессором, но ограничен в объёме.
2. Диск, который расположен далеко от процессора, имеет медленную связь с процессором, но практически неограничен в объёме.



Исходя из этой терминологии, кэш — это быстрая память, которой мало, а диск — это медленная память, которой много.

Основное в этой модели то, что обмен между кэшем и диском происходит блоками данных. Диск разбит на блоки, каждый из которых состоит из B элементов, и когда необходим один элемент из блока на диске, весь блок копируется в кэш. Кэш может содержать до $\frac{M}{B}$ блоков или M элементов ($B \le M$). Перед тем, как взять блок из диска, когда

кэш заполнен, алгоритм должен решить какой блок можно выкинуть из кэша.

Одно из преимуществ данной модели над остальными то, что у модели есть всего два уровня памяти. Алгоритм, работающий на такой модели, будет зависеть только от B и M . Эти характеристики исчезнут в кэш-независимой модели.

2.Элементарный блочный алгоритм

Рассмотрим задачу: перемножить две большие плотные матрицы $C = AB$. Матрицы перемножаются по правилу $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$. А теперь рассмотрим различные реализации этого умножения. Для простоты, матрицы будем хранить в одномерном массиве размера a^2 . К элементу $a_{i,j}$ будем обращаться как a[i*n + j].

Первый самый простой способ перемножения следующий: умножаем в цикле все строки первой матрицы на все столбцы второй матрицы и получаем результирующую матрицу.

Второй способ перемножения: уменьшим первые два цикла в два раза, то есть будем брать по две строки и по два столбца за раз, соответственно получать 4 значения в результирующей матрице. Постараемся как можно реже обращаться к памяти и умножать, и сохранять все в локальных переменных и указателях, что также ускорит работу.

Необходимо отметить, что когда идем по матрице в цикле, то приходится брать информацию из оперативной памяти (особенно когда считываем по столбцу). Из оперативной памяти в кэш информация передаётся блоками (т.е. когда читаем какой-то элемент, в кэш кладется несколько следующих за ним элементов в памяти), а когда идем по столбцу, то приходится обращаться в другую область памяти и, следовательно, подгружать новый блок в кэш-память, то есть происходят кэш-промахи. Чтобы такого не случалось, сделаем следующее: разобьем нашу матрицу на подматрицы размера примерно 100 на 100 (размер задаётся в аргументе функции). Назовем наши подматрицы A(1,1),…,A(n,n) соответственно. Будем перемножать подматрицы по обычному правилу. Несложно математически доказать, что ответ получится такой же, как и при обычном перемножении, но теперь, когда будем перемножать две маленькие матрицы, они обе поместятся в кэш-память и перемножатся намного быстрее. Еще для перемножения понадобятся функции «взять подматрицу», «положить подматрицу, добавив к тому, что было» и функция обнуления матрицы.

В качестве исходных матриц возьмём произвольные матрицы. Программа тестировалась на компьютере Intel(R) Core(TM)2 Duo CPU E8500 @ 3,16GHz, 3,25 GBt RAM.

Результаты

n(размерность)	500	1000	2000	4000
Обычное умножение	1.1с.	10с.	95с.	650с.
Улучшенное умножение	0.4с.	3.8с.	29с.	230с.
Блочное умножение	0.2с.	1.4с.	11с.	84с.

Из приведенной таблицы виден существенный рост производительности (ускорение примерно в 7-8 раз на самом современном процессоре). Стоит учесть, что это все считалось в одном процессе, а значит двухъядерность Core2Duo не сыграла роли. Кроме того, можно распараллелить программу и получить прирост еще в несколько раз. Также стоит отметить, что прогресс не стоит на месте, и более современные процессоры в несколько раз обгоняют модели 5-летней давности.

3. Кэш-независимая модель

Основная идея кэш-независимой модели проста[3]: необходимо построить алгоритмы для внешней памяти, не зная B и M . Но у этой простой идеи есть несколько важных моментов.

Первый – это то, что кэш-независимый алгоритм хорошо работает с двумя уровнями памяти (изначально кэшем и диском), тогда он также должен хорошо работать и между любыми двумя смежными уровнями памяти. Это следует немедля, так как он относится к каждым двум смежным уровням памяти, каждые, по-видимому, с разными значениями параметров B и M , таким образом, что блоки в памяти, которая ближе к процессору, хранят подмножества уровней памяти, которые располагаются дальше от процессора.

Другой момент – это то, что если число передач памяти оптимально до постоянного коэффициента между любыми двумя смежными уровнями памяти, тогда любая комбинация этих чисел также находится в пределах оптимальной константы. Таким образом, можно проектировать и анализировать алгоритмы в модели с двумя уровнями памяти и получать результаты для произвольных многоуровневых моделей, и можно сделать алгоритмы кэш-независимыми.

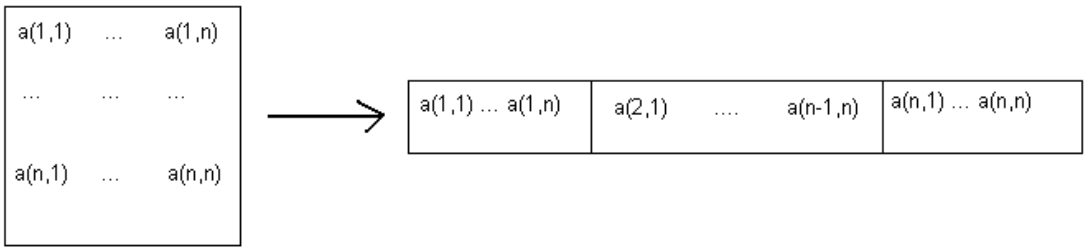
В статье [3] показано, что алгоритм, оптимально работающий на двухуровневой модели, будет эффективно работать и на многоуровневой модели.

4. Построение кэш-независимого алгоритма

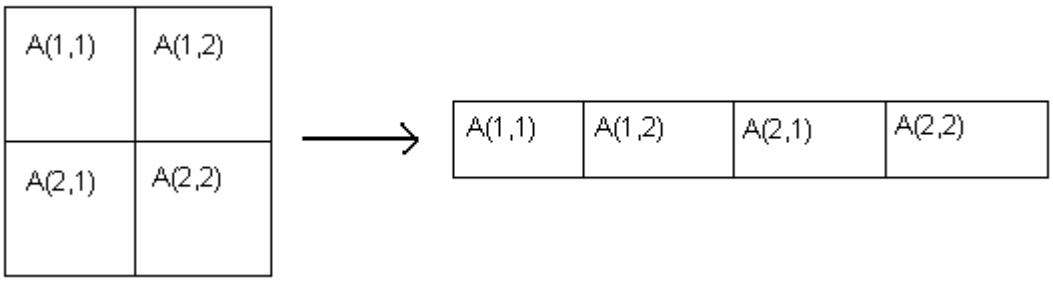
Блочный алгоритм показывает, как можно эффективно использовать кэш-память, но он имеет несколько недостатков. Первое – это то, что необходимо явно указывать размер блока, то есть необходимо его подбирать в соответствии с размером кэш-памяти, а размер кэш-памяти далеко не всегда нам известен. Вторая проблема – это то, что программа получилась плохо переносимая на другие архитектуры ЭВМ. Неизвестно, какая иерархия памяти на другой ЭВМ и насколько удачно удастся подобрать размер блока, так что необходимо как-то изменять этот алгоритм, чтобы он работал в независимости от архитектуры ЭВМ и размера кэш-памяти.

Пусть перед нами задача $C = AB$. При обычном умножении ЭВМ будет обращаться к элементам матриц, которые расположены далеко друг от друга, и в этом случае будет происходить кэш-промах, так как при подгрузке в кэш-память какого-либо «дальнего» элемента в кэш-память будет загружен не один нужный ЭВМ элемент, а целый блок. Смысл кэш-независимого алгоритма – сделать число кэш-промахов минимальным.

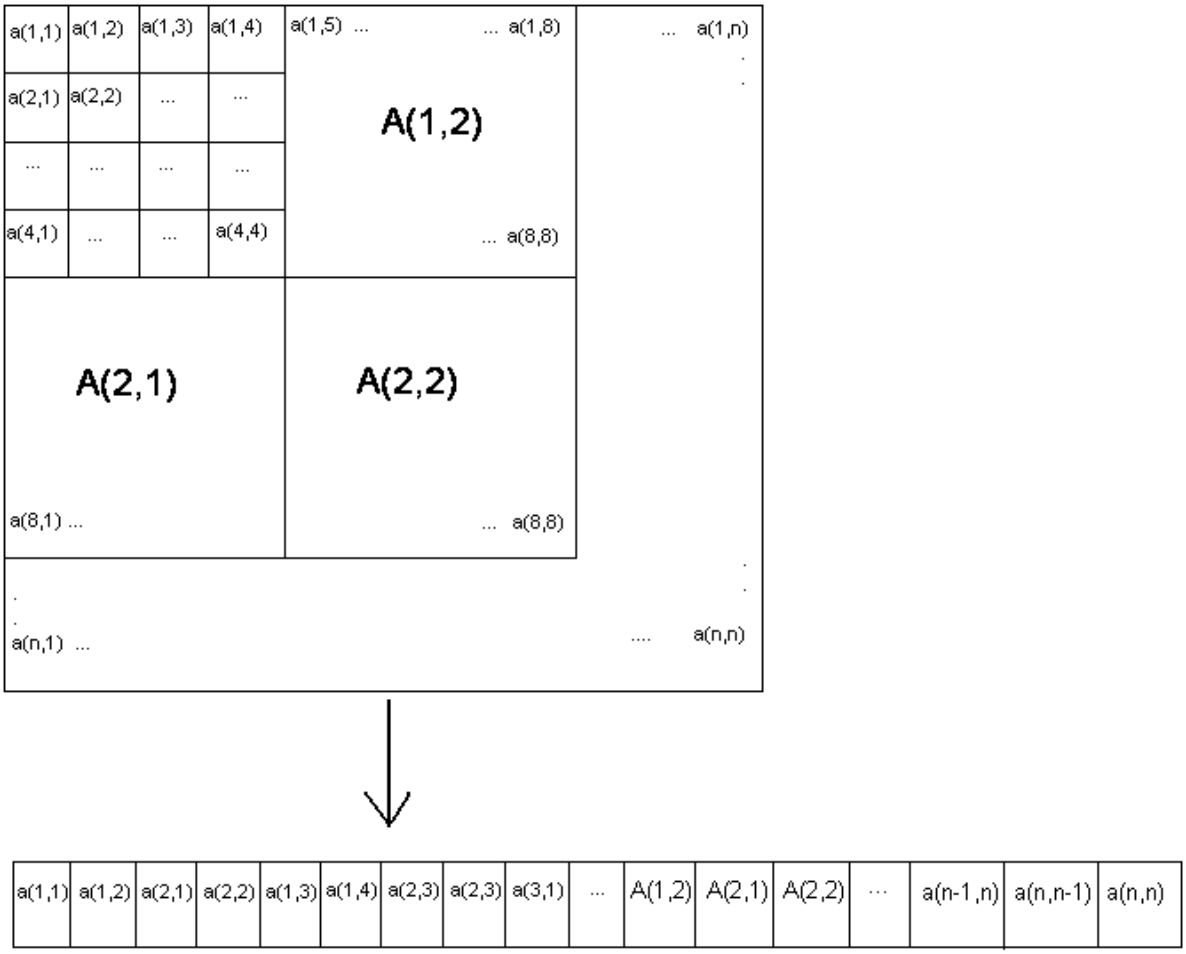
Теперь давайте рассмотрим, как хранятся матрицы в памяти ЭВМ. Если рассматривать язык программирования Си, то матрицы хранятся в строчном формате, элементы в памяти будут располагаться следующим образом:



Разобьём исходную матрицу A пополам по строкам и столбцам, получим 4 подматрицы A(1,1), A(1,2), A(2,1), A(2,2). Сделаем следующую перестановку элементов в памяти ЭВМ.



Теперь в памяти матрица лежит не по строкам, а по блокам: сначала идут элементы блока A(1,1), затем (A1,2), A(2,1) и A(2,2). Применяем эту идею рекурсивно для каждой из подматриц. В итоге получим следующую последовательность хранения элементов матрицы в памяти.



Сделав сортировку элементов матрицы A в памяти ЭВМ, сделаем перестановку элементов матрицы B по тому же алгоритму, за единственным отличием: блочное расположение матрицы B будет столбцовым. Разбив матрицу B на подматрицы, расположим их в памяти следующим образом: B(1,1), B(2,1), B(1,2), B(2,2). В итоге, переставив в памяти ЭВМ элементы обоих матриц, запускаем умножение.

Программа тестировалась на компьютере Intel(R) Core(TM)2 Duo CPU E8500 @ 3,16GHz, 3,25 GBt RAM.

Результаты:

n(размерность)	500	1000	2000	4000
Обычное умножение	1.1с.	10с.	95с.	650с.
Блочное умножение	0.2.с.	1.4с.	11с.	84с.
Кэш-независимое умножение	0.3с.	2.3с.	22с.	137с.

Этот алгоритм немного уступает блочному в том случае, если в блочном оптимально подобрать размер блока, а в противном случае этот алгоритм работает быстрее.

5. Умножение разреженной матрицы на блочный вектор

Как показано ранее, эффективное использование кэш-памяти даёт немалый выигрыш при умножении плотных матриц. Теперь хотелось бы ускорить операцию умножения разреженной матрицы на блочный вектор. Будем рассматривать разреженные матрицы размера не менее чем 250 000, в каждой строке примерно от 100 до 200 ненулевых элементов, в нашем случае единиц, так как рассматриваем задачу над полем GF(2). Матрица хранится в разреженном формате, то есть в двух массивах ia и ja. Пусть матрица имеет размер n, тогда массив ia будет состоять из n+1 элемента, ia[i+1] - ia[i] = количество ненулевых элементов в i-ой строке, тогда соответственно общее количество ненулевых элементов = ia[n] - ia[0]. Массив ja имеет размер равный числу ненулевых элементов и состоит из указателей на столбцы. Теперь, как и в случае плотных матриц, необходимо сделать некоторую сортировку массивов ia и ja, чтобы максимально эффективно использовать кэш-память. Ниже предлагается два способа.

1. Пусть есть матрица A размерности n на n, состоящая из nnz ненулевых элементов, хранящаяся в разреженном формате. Для удобства переведем матрицу из разреженного формата в строчно-столбцовый формат, то есть от массивов ia, ja перейдём к массивам row, col. Массив row имеет размерность nnz и хранит указатели на строки, а массив col есть массив ja (для удобства изменили название).

Будем использовать ту же идею, что и в случае плотных матриц. Разбиваем матрицу пополам по строкам и столбцам, получаем 4 подматрицы A(1,1), A(1,2), A(2,1), A(2,2), затем производим сортировку массивов row и col так, что в памяти сначала лежат указатели (на строки и столбцы), указывающие на блок A(1,1), затем на блок A(1,2), затем на блок A(2,1), а затем на блок A(2,2).

Теперь делаем аналогичную процедуру для каждой из подматриц и так далее. Эта идея была реализована в виде рекурсивной процедуры. К сожалению, даже на не очень больших матрицах(n более 50 000) процедура оказывается слишком медленной из-за большого числа перестановок, поэтому необходимо уменьшить количество перестановок.

2. Присвоим каждому элементу матрицы свой номер. Номера будем присваивать следующим образом. Точно так же, как и в предыдущем алгоритме, разобьём исходную матрицу на 4 подматрицы. Если элемент принадлежит первой подматрице, тогда в старшие 2 бита переменной номера элемента запишем 00, если элемент принадлежит второй

подматрице, тогда в старшие 2 бита переменной номера элемента запишем 01, если элемент принадлежит третей подматрице, тогда в старшие 2 бита переменной номера элемента запишем 10, а если четвёртой, то 11. Далее действуем по тому же принципу с той подматрицей, в которую попал элемент, только записывать результат будем не в старшие 2 бита, а в следующие 2 бита и так далее. Эта идея также реализована в виде рекурсивной процедуры, которая при каждом вызове записывает 2 бита в переменную номера элемента. Итак, вызываем эту процедуру для каждого элемента отдельно и получаем массив номеров элементов, в котором их нужно расположить в памяти ЭВМ. Далее нам необходимо расположить элементы массива номеров по возрастанию. Сортируя массив номеров, параллельно сортируем массивы row и col и получаем нужное расположение в памяти ЭВМ. Сделав данную перестановку, запускаем обычное умножение и получаем существенный выигрыш по скорости вычислений, что наглядно демонстрирует следующая таблица.

n(размерность)	50 000	227 000
Обычное Умножение	22с.	90с.
Кэш-независимое умножение	7с.	28с.

Литература

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. Commun. ACM, 31:1116-1127,1988.
2. R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, Complexity of Computer Calculations, Plenum, 1972 p.105-109.
3. Erik D. Demaine, Cache-Oblivious Algorithms and Data Structures, MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 021339, USA 2002 p.1-19.