МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра дискретной математики и алгоритмики

ЭФФЕКТИВНОЕ ИСПОЛЬЗОВАНИЕ МНОГОУРОВНЕВОЙ ПАМЯТИ ПРИ ОБРАБОТКЕ ДАННЫХ

Курсовой проект

Вежновца Дмитрия Артёмовича

студента 3 курса,

специальность «информатика»

Научный руководитель:

старший преподаватель

А.А. Толстиков

Минск, 2022

ОГЛАВЛЕНИЕ

[Введение 3](#_Toc121862810)

[Глава 1 Классическое перемножение матриц 4](#_Toc121862811)

[1.1 Определение перемножения матриц 4](#_Toc121862812)

[1.2 Программное представление матриц 4](#_Toc121862813)

[1.3 Асимптотика и программная реализация перемножения 4](#_Toc121862814)

[Глава 2 Блочное перемножение матриц 6](#_Toc121862815)

[2.1 Идея блочного перемножения 6](#_Toc121862816)

[2.2 Реализация блочного перемножения 6](#_Toc121862817)

[2.3 Класс View 7](#_Toc121862818)

[2.4 Деление и согласование блоков 7](#_Toc121862819)

[Глава 3 Алгоритм Штрассена 9](#_Toc121862820)

[3.1. Идея и асимптотика алгоритма Штрассена 9](#_Toc121862821)

[3.2. Реализация алгоритма Штрассена 10](#_Toc121862822)

[Глава 4 Исследования и выводы 11](#_Toc121862823)

[4.1. Сравнение скоростей 11](#_Toc121862824)

[4.2. Замедление класса View 12](#_Toc121862825)

[4.3. Недостаток блочного перемножения матриц 12](#_Toc121862826)

[Заключение 13](#_Toc121862827)

[Список использованных источников 14](#_Toc121862828)

[Приложение a. Код перемножения матриц представленных в виде одномерных массивов 15](#_Toc121862829)

Введение

Основной функцией компьютеров является обработка данных. Каждую секунду компьютер обрабатывает какую-либо информацию. Уже на протяжении нескольких десятилетий люди пытаются ускорить разными методам процесс обработки. Одной из важных концепций при ускорении обработки является оптимальное использование кэш-памяти. Перемножение матриц используется в различных сферах в нашей жизни. Сама обработка перемножения является достаточно тяжёлой операцией. В данном курсовом проекте рассматривается как можно ускорить перемножение матриц на основе кэш-памяти и различных методов. Рассматриваются различные методы перемножения матриц, а также различные способы представления матриц, их преимущества и недостатки при реализации алгоритма перемножения.

Глава 1 Классическое перемножение матриц

1.1 Определение перемножения матриц

Рассмотрим две матрицы и размерами и соответственно, где первое число обозначает количество строк, а второе количество столбцов. Для перемножения матриц важно, чтобы они были согласованы, то есть количество столбцов левой матрицы совпадало с количеством строк правой матрицы.

Произведением матриц и размеров и соответственно будет матрица размеров , такая что каждый элемент определяется следующим образом:

1.2 Программное представление матриц

Для программной реализации перемножения матриц для начала необходимо понять как их представить. Матрицы представляются в виде одномерного массива. Для матрицы массив будет размера . Матрицы так же можно представлять и в виде двумерных массивов, однако в данном случае мы будем использовать именно такой способ.

Существуют два способа отображения матрицы в одномерную структуру, а именно row-major и column-major. Первый способ заключается в том, чтобы хранить по очереди строки матрицы в одномерном массиве. Тогда элемент матрицы будет иметь индекс в одномерном массиве. Второй способ имеет ту же идею, но теперь будут храниться по очереди столбцы матрицы. Тогда элемент матрицы будет иметь индекс в одномерном массиве.

Для левой матрицы будем использовать row-major, а для правой column-major. Именно такой способ связан с тем, что при перемножении в левой матрице происходит итерирование по строке, а в правой по столбцу. При передаче данных оптимально использовать непрерывный блок памяти, как раз подобные представление в этом плане подходят.

1.3 Асимптотика и программная реализация перемножения

Перемножение матриц программно можно реализовать следующим образом.

Реализовать тройной цикл, в котором поочерёдно будем считать каждый элемент произведения матриц. Первые два цикла будут отвечать за переход между элементами матрицы, третий за итерирование по строке левой матрицы и столбцу правой матрицы. Код реализации для матриц, представленных в виде одномерных массивов, можно найти в Приложении A.

У нас имеются три цикла: первый от 0 до , второй от 0 до , третий от 0 до Асимптотика алгоритма будет . Для случая, когда у нас матрицы квадратные и имеют размер , асимптотика будет .

Глава 2 Блочное перемножение матриц

2.1 Идея блочного перемножения

Пусть есть матрицы достаточно больших размеров, которые теоретически могут не влезать в память и их нужно перемножить. С этой задачей может помочь справиться блочное перемножение матриц.

Сама суть заключается в том, чтобы разделить обе матрицы на 4 блока. Таким образом получим 2 матрицы размера 2 на 2 состоящие из блоков, и перемножать именно блоки по той же формуле перемножения матриц, а после суммировать результат. Блоки так же можно делить на меньшие блоки, пока не получим матрицы допустимого размера. Результатом так же будет блочная матрица. Причём данное перемножение применимо как для квадратных, так и для прямоугольных матриц.

Данное перемножение используется так же ради повышения эффективности использования кэш-памяти. Подробнее об этом в [1, с. 11].

Рисунок 2.1 визуализирует блочное перемножение матриц.

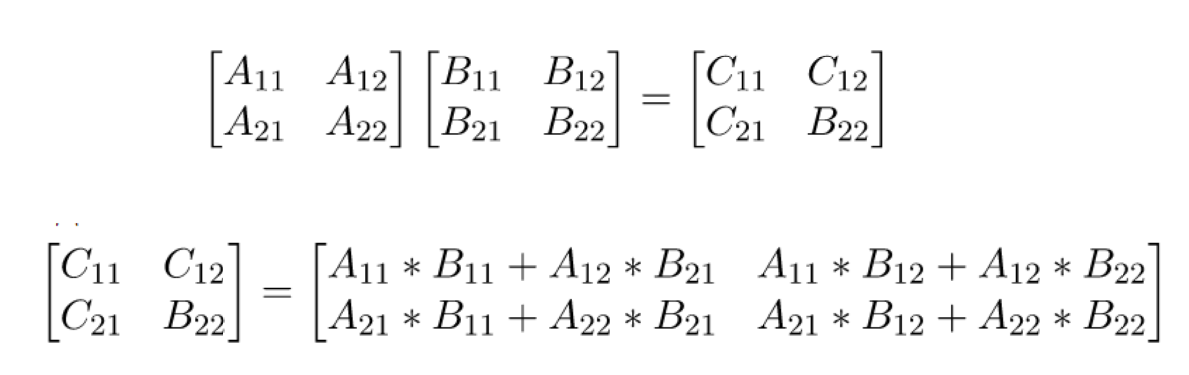


Рисунок 2.1 Блочное перемножение матриц

Блочное перемножение может так же ускорить алгоритм в следующей ситуации. Пусть мы умеем быстро перемножать матрицы размера 4 на 4 быстрее, чем за . Тогда разбив матрицы до матриц 4 на 4 можно ускорить умножение. Однако это частный случай, который можно применить только для квадратных матриц, размеры которых равны степени двойки. При этом всём любую матрицу можно дополнить до квадратной матрицы с размером равным степени двойки, приписав нулевые строки/столбцы.

2.2 Реализация блочного перемножения

Данное перемножение можно реализовать рекурсивно. Идея такова, если размеры матрицы не превосходят заданных размеров, то есть возможно эффективно использовать кэш-память, то мы производим перемножение, в противном случае делим матрицу на блоки и вызываем рекурсивно перемножение для блоков, а после собираем блоки в единую матрицу. Рекурсия вызывается до тех пор, пока блоки не станут допустимого размера.

2.3 Класс View

При реализации блочного перемножения возникает проблема. Для каждого блока выделяется новая память и переносятся данные, причём это может происходить большое количество раз, что может существенно замедлить время программы.

Так как мы работаем с динамическими одномерными массивами, то существует решение данной проблемы.

Заведём класс View, который будет состоять из указателя на одномерный массив, 6 целочисленных переменных и одной булевской. Суть в том, что можно в массиве выделить блок зная указатель, координаты начала блока, длину и ширину блока. На это уходят 4 переменные. Ещё 2 переменные уходят на размер основной матрицы, чтобы можно было корректно обращаться к элементу блока. Булевская переменная нужна чтобы определять тип отображения, а именно row-major или column-major.

Таким образом нам не нужно выделять целые новые блоки памяти и переносить их, а достаточно лишь передать указатель и несколько переменных, которые определят блок.

Для удобства работы с этим классом появилась необходимость добавить два метода — get и set. Метод get нужен чтобы получить значение с индексами в блоке, а set чтобы установить значение для заданных индексов. Пусть блок начинается с индекса , а длины строки и столбца основной матрицы соответственно. Тогда к элементу в блоке можно обратиться по следующему индексу, если тип отображения row-major, если тип отображения column-major.

Подразумевается, что мы работаем конкретно с блоком, у которого индексация начинается с 0.

2.4 Деление и согласование блоков

Блоки будут делиться по следующему принципу, длину столбца и строки левой и правой матрицы делим на 2. В случае если все длины чётные всё будет работать корректно. Рассмотрим на примере получения блока . Пусть у нас заданы и все они чётные. Тогда итоговая матрица будет иметь размер на , а блок будет иметь размер на . Блок получается по следующей формуле . Блоки и будут иметь размеры на , а блоки и будут иметь размеры на . Таким образом блоки и будут иметь размеры на . С остальными блоками аналогично.

Для блоков матриц длины нужно сделать небольшую поправку. Пусть у нас матрица на , где нечётные. Тогда левый верхний блок будет иметь размер на , правый верхний на , левый нижний на , правый нижний на . В этом случае блоки всё так же останутся согласованными и перемножение пройдёт успешно.

Глава 3 Алгоритм Штрассена

3.1. Идея и асимптотика алгоритма Штрассена

Данный алгоритм является алгоритмом быстрого перемножения матриц. В отличие от классического перемножения, асимптотическое время данного алгоритма будет , что приблизительно равно .

В блочном перемножении использовалась идея разделения на блоки. Рассматривать идею будем в терминах квадратных матриц, стороны которых равны степени двойки. Пусть есть 2 матрицы размерами , и их нужно перемножить. Разделив матрицу на блоки и выполнив блочное перемножение, будет произведено 8 перемножений матриц размеров .

Идея алгоритма Штрассена заключается в том, чтобы уменьшить количество перемножений блоков до 7. Сделать это возможно произведя операции сложения и вычитания над выделенными блоками.

Дальше всё так же считается, что матрица квадратная и имеет сторону равную степени двойки. В любом случае любую матрицу можно дополнить до квадратной со стороной равной степени двойки, приписав нулевые строки и столбцы.

Из заданных 8 блоков основных матриц можно сформировать 7 новых блоков по следующим формулам:

Тогда блоки матрицы можно получить следующим образом.

Доказательство корректности алгоритма можно получить, подставив значения новых матриц через блоки начальных матриц и сделав математические преобразования. Разберём случай для блока для остальных аналогично.

Что верно из блочного перемножения матриц. Таким образом по средствам подобных преобразований сокращается количество перемножений блоков с 8 до 7.

3.2. Реализация алгоритма Штрассена

Реализация данного алгоритма подобна блочному. Вначале делится матрица на блоки. В этом моменте можно использовать описанный выше класс View. После формируются новые блоки посредствам сложения и вычитания. Для новых сформированных блоков вызывается рекурсивно эта же функция перемножения. Деление на блоки будет происходить до тех пор, пока не достигается допустимый размер матриц.

Недостаток подобной реализации заключается в том, что для новых блоков выделяется новая память и происходит пересчёт, однако уменьшение количества перемножений даёт больший выигрыш в скорости, чем замедление от формирования новых блоков.

Так же мы рассматривали данный алгоритм именно для квадратных матриц размеры которых равны степени двойки. В противном случае матрица дополняется до квадратной. На это так же тратится время, к тому же увеличивается размерность задачи. В некоторых случаях не всегда оптимально использовать данный алгоритм, например если размерность задачи сильно увеличивается. Однако для достаточно больших матриц данный алгоритм даёт значительный выигрыш по времени.

Глава 4 Исследования и выводы

4.1. Сравнение скоростей

Для начала стоит сравнить насколько ускоряется классическое перемножение при хранении матриц в виде row-major, column-major. В качестве тестовых данных рассматриваются две матрицы 2048 на 2048 состоящие только из единиц. Первый метод хранения из себя представляет вектор векторов, то есть двумерный динамический массив. Второй метод одномерный массив (row-major, column-major).

При запуске программы в режиме release первый метод выдавал среднее время около 38 секунд. Второй же метод ускорил программу в разы. Время перемножения матриц при втором способе хранения составило 3-4 секунды. При этом всём, в случае, когда и левая и правая матрица имеет одинаковый тип хранения (row-major или column-major), то классическое перемножение занимает порядка 60-80 секунд.

Дальше рассматривается только способ хранения row-major для левой матрицы, column-major для правой матрицы, а также использование класса View.

Сравнение классических перемножений с использованием класса View и без него показало, что использование данного класса замедляет его работу по причине того, что вызываются методы get. Несмотря на то, что перед методами стоит слово inline, использование данного класса всё равно затрачивает какое-то время, однако для блочного перемножения и алгоритма Штрассена оно даёт улучшение по времени из-за потери необходимости выделения дополнительной памяти в отдельных случаях.

При желании можно убрать использование класса и методов get/set и работать только с переменными, которые будут заменять использование этого класса. В рамках этого проекта для точного сравнения алгоритмов будут рассматриваться все перемножения с использованием класса View, в противном случае классическое перемножение будет иметь преимущество.

При сравнении алгоритмов блочного и классического перемножения матриц оба алгоритма показали приблизительно одинаковое время работы. В блочном перемножении блоки делились ровно один раз. В рамках этого проекта для блоков меньшего размера используется так же классическое перемножение матриц, а не ускоренное, по средствам этого получается приблизительно одинаковый результат, так как количество операций перемножения не изменяется, и кэш-промахи остаются приблизительно одинаковыми. Итоговое время работы алгоритмов в среднем составляет 5,4 секунды. Однако стоит заметить, что при делении блоков дважды среднее время блочного перемножения матриц уменьшилось до 5,1 секунд. При последующем увеличении количества делений время не улучшалось.

Для алгоритма Штрассена понадобилось рассмотреть несколько случаев, а именно когда размеры исходной матрицы равны степени двойки, и когда размеры исходной матрицы на единицу больше, чем степень двойки. Это связано с тем, что использование данного алгоритма требует достраивания матрицы до квадратной со степенью двойки, поэтому рассматривается лучший и худший случай.

В лучшем случае алгоритм Штрассена превзошёл и классическое, и блочное перемножение матриц. Время работы алгоритма для матрицы 2048 на 2048, когда деление происходило дважды в среднем составило 4,3 секунды, что на 0,8 секунд лучше, чем блочное перемножение и на 1,1 секунду лучше, чем классическое.

Для худшего случая рассматривается две матрицы размерами 1025 на 1025. Время работы алгоритмы Штрассена по-прежнему составило 4.3 секунды, так как матрица дополнялась до размера 2048 на 2048, но при этом классическое и блочное перемножение справлялись приблизительно за секунду. Из-за повышения размера матрицы время работы алгоритма может очень сильно замедляться.

4.2. Замедление класса View

По итогу самое лучшее время работы показал алгоритм классического перемножения без использования класса View. В среднем данный алгоритм перемножал матрицы размерами 2048 на 2048 за 3 секунды. Таким образом данный класс замедляет работу. Если очень важна скорость работы, то стоит использовать замену данному классу, и работать только с переменными.

4.3. Недостаток блочного перемножения матриц

С квадратными матрицами данный алгоритм работает неплохо, однако существует случай, когда понижается эффективность работы кэш-памяти. В случае, когда одна из сторон матриц значительно меньше второй. Для левой матрицы это случай, когда длина строки слишком мала, а для правой длина столбца слишком мала. В таком случае блоки памяти в кэш передаются слишком маленькими, значительно меньше, чем размер кэша. За счёт этого, понижается эффективность его работы.

Существует следующее решение этой проблемы. Вместо того, чтобы передавать в качестве одного блока одну строку или столбец, можно передавать сразу несколько подряд идущих строк/столбцов. Данное решение помогает заместить потери эффективности.

Ссылка на GitHub со всем кодом проекта:

https://github.com/devooox/CoureProject3rdGrage

Заключение

Различные методы показали разное время работы. В некоторых случаях лучше всего работал алгоритм Штрассена, в некоторых случаях лучше проявлял себя блочный метод, а в некоторых и вовсе классическое перемножение давало лучший результат. Обобщив все эти решения и разделив на частные случаи, можно организовать единый метод перемножения, который будет давать лучшее время работы.

В дальнейшем возможны следующие шаги исследования в подобной теме:

1. Реализация алгоритмов с хранением на внешних носителях (HDD, SSD);
2. Многопоточная реализация алгоритмов с учетом инклюзивных и эксклюзивных уровней кэша многоядерного процессора.

Список использованных источников

1. Demaine, Erik D. Cache-Oblivious Algorithms and Data Structures. [Article] – MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA.

Приложение a.  
Код перемножения матриц представленных в виде одномерных массивов

**int\* matrixMultiplication(const int\* A, const int\* B, int n, int m, int l) {**

**int\* C = new int[n\*l];**

**for (int i = 0; i < n\*l; ++i) {**

**C[i] = 0;**

**}**

**for (int i = 0; i < n; ++i) {**

**for (int j = 0; j < l; ++j) {**

**for (int k = 0; k < m; ++k) {**

**C[i\*l+j] += A[i\*m+k]\*B[k+j\*m];**

**}**

**}**

**}**

**return C;**

**}**