МАТЕМАТИЧЕСКОЕ МОДЕЛИРОВАНИЕ, ЧИСЛЕННЫЕ МЕТОДЫ И КОМПЛЕКСЫ ПРОГРАММ / MATHEMATICAL MODELING, NUMERICAL METHODS AND PROGRAM COMPLEXES

DOI: https://doi.org/10.60797/IRJ.2024.143.115

РЕАЛИЗАЦИЯ МЕТОДА СОПРЯЖЕННЫХ ГРАДИЕНТОВ НА ГРАФИЧЕСКОМ ПРОЦЕССОРЕ С ПРИМЕНЕНИЕМ МАТРИЧНЫХ МЕТОДОВ РЕШЕНИЯ СЛАУ

Научная статья

Устюжанин Д.А.[1], Пицхелаури С.С.[2,] *, Некрасов К.А.[3]
[2] ORCID : 0000-0003-2192-1880;
[3] ORCID : 0000-0002-1863-2597;
[1, 2, 3] Уральский федеральный университет имени первого Президента России Б.Н. Ельцина, Екатеринбург, Российская Федерация

* Корреспондирующий автор (lauri2011[at]mail.ru)

Аннотация

В работе представлен алгоритм решения систем линейных алгебраических уравнений (СЛАУ) методом сопряженных градиентов на графическом процессоре. В качестве основного инструмента реализации предложен класс-контейнер для работы с матрицами на базе технологии NVIDIA CUDA. Производительность решения СЛАУ методами Крамера и сопряженных градиентов была сопоставлена на центральном и графическом процессорах. Результаты исследования показали, что распараллеленный метод сопряженных градиентов, выполненный на графическом процессоре, обладает наибольшей эффективностью при обработке СЛАУ с симметричной положительно определенной основной матрицей. Рассматриваемый подход к параллельной обработке матричных операций имеет потенциал для применения в различных областях, где требуется решение крупных систем уравнений, таких как в науке, инженерии и финансах. В целом, данная работа представляет практическую значимость в области оптимизации производительности вычислений и закладывает фундамент для решения многих математических задач на графическом процессоре.

Ключевые слова: распараллеливание алгоритмов, решение СЛАУ на графическом процессоре, матричные операции.

REALIZATION OF THE METHOD OF CONJUGATE GRADIENTS ON A GRAPHIC PROCESSOR WITH APPLICATION OF MATRIX METHODS OF SLAU SOLUTION

Research article

Ustyuzhanin D.A.[1], Pitskhelaury S.S.[2,] *, Nekrasov K.A.[3]
[2] ORCID : 0000-0003-2192-1880;
[3] ORCID : 0000-0002-1863-2597;
[1, 2, 3] Ural Federal University, Ekaterinburg, Russian Federation

* Corresponding author (lauri2011[at]mail.ru)

Abstract

The paper presents an algorithm for solving SLAEs by the method of conjugate gradients on a graphics processor. A matrix container class based on NVIDIA CUDA technology is proposed as the main implementation tool. The performance of SLAE solution by Cramer and conjugate gradient methods was compared on the central and graphic processors. The results of the study have shown that the parallelized method of conjugate gradients performed on a graphics processor has the highest efficiency when processing SLAEs with a symmetric positively defined principal matrix. The considered approach for parallel processing of matrix operations has potential for application in various areas where large systems of equations need to be solved, such as in science, engineering and finance. Overall, this work is of practical relevance in the field of computational performance optimization and lays the foundation for solving many mathematical problems on a GPU.

Keywords: parallelization of algorithms, solving SLAEs on a graphics processor, matrix operations.

Введение

Системы линейных алгебраических уравнений (СЛАУ) являются фундаментом задач для многих областей естествознания, включая механику, электротехнику, экономику и компьютерную графику. Способы решения СЛАУ имеют важное значение, особенно при обработке большого количества уравнений. Методы решения СЛАУ делятся на точные и приближенные, включая итерационные методы. Алгоритмы для вычислительных машин, основанные на точных методах, таких как метод Гаусса и метод Крамера, подходят для систем порядка не выше 20, в то время как итерационные методы могут обрабатывать системы более высокого порядка.

Одним из подходов к решению СЛАУ является параллельное программирование, включая использование графических процессоров (GPU). В отличие от центрального процессора (CPU), GPU имеет сотни или тысячи ядер, способных обрабатывать вычисления одновременно. Это делает GPU более эффективным для параллельных задач, таких как обработка графики, научные вычисления и машинное обучение. Кроме того, на GPU реализуются различные методы решения СЛАУ.

Метод сопряженных градиентов относится к итерационным методам и основывается на умножении матриц и вычислении скалярных произведений векторов. Умножение матриц строится на основе нахождения суммы

произведений соответствующих элементов исходных матриц, т. е. состоит из множества несвязных вычислений. Поэтому метод сопряженных градиентов подходит для распараллеливания на графическом процессоре.

Работа с матрицами при использовании технологии CUDA

Архитектура CUDA представляет собой специальный набор инструментов и библиотек, предназначенный для программирования графических процессоров. В качестве языков программирования CUDA поддерживает Python, C, C++ и т. д. Распараллеливание вычислений на GPU достигается за счет запуска большого количества потоков. Потоки в архитектуре CUDA представляются в виде сложной структуры. На верхнем уровне располагается сетка, которая является дискретным трехмерным пространством блоков, каждый блок в свою очередь трехмерным пространством потоков.

Матрицы заданного размера, состоящие из действительных чисел, образуют следующее линейное пространство:

$$M_{n\times m} = \left\{ \begin{pmatrix} a_{00} & \cdots & a_{0m-1} \\ \vdots & \ddots & \vdots \\ a_{n-10} & \cdots & a_{n-1m-1} \end{pmatrix} \middle| \, a_{ij} \in \mathbb{R} \right\} \quad (1)$$

В настоящей работе матрицы представляли как одномерные динамические массивы. Доступ к элементу матрицы осуществляли по следующему соотношению: $ptr + i * M + j, i = \overline{0, N-1}, j = \overline{0, M-1})$, где N – число строк в матрице, M – число столбцов в матрице, ptr – указатель на первый элемент массива. В качестве «обертки» для динамического массива использовали класс-контейнер devMatrix.

Таблица 1 - Основные методы класса devMatrix

DOI: https://doi.org/10.60797/IRJ.2024.143.115.1

Сигнатура метода	Назначение метода
`devMatrix();`	Конструктор по умолчанию.
`devMatrix(size_t N, size_t M);`	Конструктор, выделяет память, но не инициализирует данные.
`devMatrix(size_t N, size_t M, double a);`	Конструктор, инициализирует данные переданным значением.
`devMatrix(devMatrix&& A);`	Конструктор, захватывает данные rvalue объекта.
`~devMatrix();`	Деструктор, освобождает память GPU.
`devMatrix operator+(devMatrix const& A);`	Операция сложения матриц.
`devMatrix operator*(devMatrix const& A);`	Операция умножения матриц.

Класс devMatrix включает все стандартные специальные методы (см. таб. 1), в том числе перемещающий конструктор и оператор присваивания. Класс Matrix является прообразом класса devMatrix и предназначен для работы с матрицами на CPU. Для удобства devMatrix имеет конструктор от Matrix.

Сложение матриц на GPU

В качестве иллюстрации внутреннего устройства класса devMatrix и работы CUDA, рассмотрим перегрузку оператора «+». Данный оператор осуществляет сложение двух матриц. Под сложением матриц будем понимать следующую операцию:

$$A_{nm} + B_{nm} = \begin{pmatrix} a_{00} & \cdots & a_{0m-1} \\ \vdots & \ddots & \vdots \\ a_{n-10} & \cdots & a_{n-1m-1} \end{pmatrix} + \begin{pmatrix} b_{00} & \cdots & b_{0m-1} \\ \vdots & \ddots & \vdots \\ b_{n-10} & \cdots & b_{n-1m-1} \end{pmatrix} = \begin{pmatrix} a_{00} + b_{00} & \cdots & a_{0m-1} + b_{0m-1} \\ \vdots & \ddots & \vdots \\ a_{n-10} + b_{n-10} & \cdots & a_{n-1m-1} + b_{n-1m-1} \end{pmatrix}. \quad (2)$$

При использовании CUDA, сумма каждой пары элементов (a_{ij}, b_{ij}) вычисляется в отдельном потоке. В данном случае использовали двумерное представление сетки и блока, при этом число блоков в одном из измерений вычисляли как ближайшее целое, большее или равное числу $\frac{N}{32}\left(\frac{M}{32}\right)$. Блоки имели размер 32×32 потока. Каждый поток обрабатывал одну сумму (см. рис. 1).

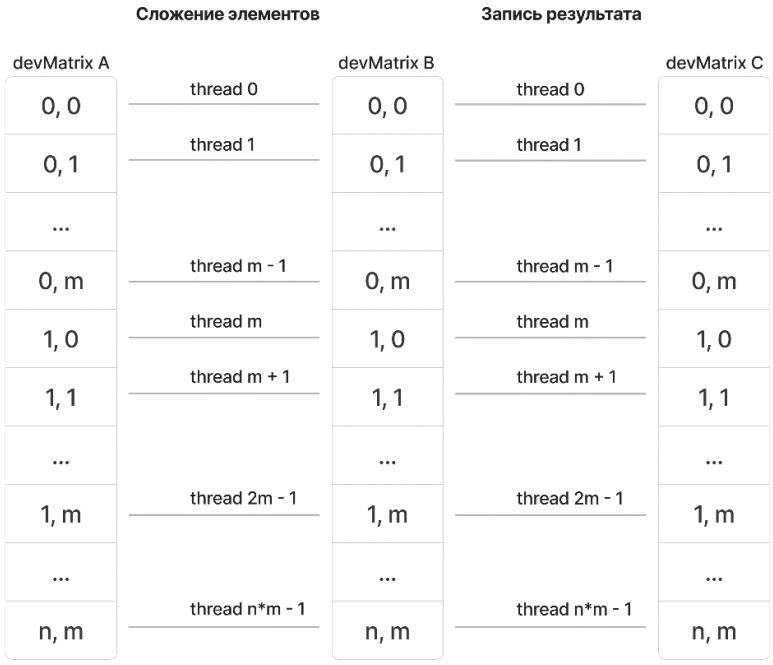

Рисунок 1 - Блок-схема суммы матриц
DOI: https://doi.org/10.60797/IRJ.2024.143.115.2

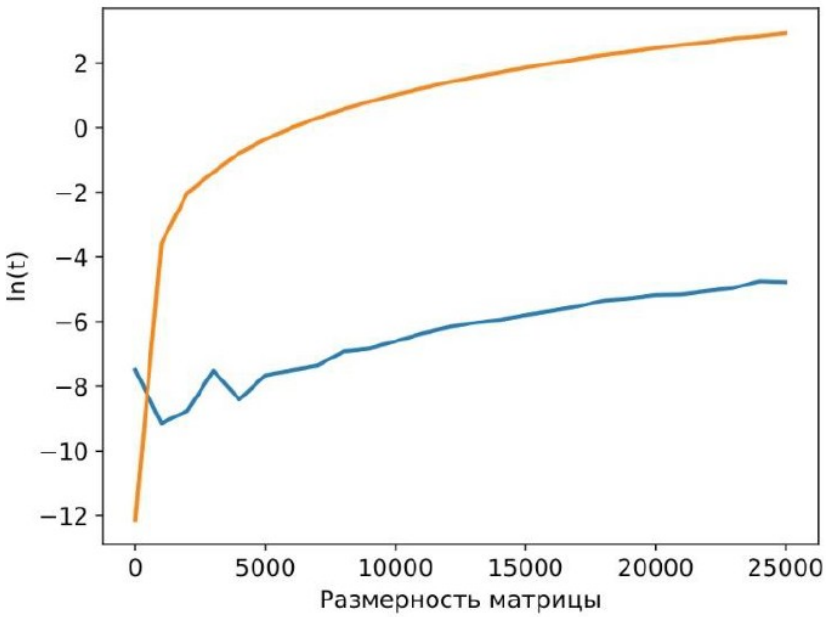

Рисунок 2 - Сравнение производительности сложения матриц на CPU и GPU
DOI: https://doi.org/10.60797/IRJ.2024.143.115.3

Примечание: оранжевая и синяя линии отображают зависимость времени от размерности матрицы для операции сложения в логарифмических координатах на CPU и GPU соответственно

Распараллеливание на GPU, при больших размерностях матриц, значительно сокращает время обработки операции сложения (см. рис. 2). При этом стоит отметить, что возрастание времени не связано на прямую с процессом суммирования элементов на GPU. Возрастание времени в данном случае определяется тем, что процесс выделения непрерывного участка памяти требует дополнительных вычислений. Также в данном случае размерность матрицы сильно ограничена объемом глобальной памяти, например квадратная матрица порядка 25000, состоящая из объектов типа double, занимает:

$$\frac{25000^2 \cdot 8}{1024^3} \approx 4.66 \text{ ГБ} \quad (3)$$

Для корректной работы, стоит учитывать, что в какой-то момент времени будет как минимум два объекта типа devMatrix, при условии, что поддерживается перемещающий конструктор. Идеология перемещающих конструктора и оператора играют важную роль в ОПП реализации матричных операций. При манипуляции большими объемами данных стоит избегать их дублирования, которое возникает при инициализации или переопределении в результате возвращения из функции rvalue объектов. В противном случае минимальное количество объектов при обработке операции возрастает до трех, что соответствует (в случае квадратной матрицы порядка 25000) 13.98 ГБ. Если отбросить ограничения, связанные с памятью, на современных видеокартах на размерность сетки накладывается ограничение в максимальное число блоков, что соответствует максимальной размерности первого измерения. Максимальная размерность блока составляет 1024 потока (подробные данные см. гайд ссылка на гайд).

Таким образом максимальное количество строк/столбцов в квадратной матрице определяется следующим образом: $\text{Т. к. } 65536^2 > 2^{31} - 1 : N \approx 32 \cdot \sqrt[2]{2^{31} - 1},$ с учетом что в нашей реализации $\frac{N}{32}$ должно быть кратно $32: N = 1482880,$ что во много раз превышает лимит памяти.

В результате верхнюю границу размеров матрицы определяет именно объем глобальной память GPU.

Умножение матриц на GPU

Умножение матриц в алгебраической форме записывается следующим образом:

$$A_{n\times m} \cdot B_{m\times k} = C_{n\times k}, \quad c_{ij} = \sum_{d=0}^{m-1} a_{id} \cdot b_{dj} \quad (4)$$

Классический распараллеленный алгоритм произведения матрицы схож с суммированием, но для каждого элемента отдельно запускается цикл, в котором рассчитывается сумма произведений (см. рис. 3). Важно, чтобы промежуточные значения суммы хранились в быстрой регистровой памяти GPU, иначе произойдет падение производительности программы.

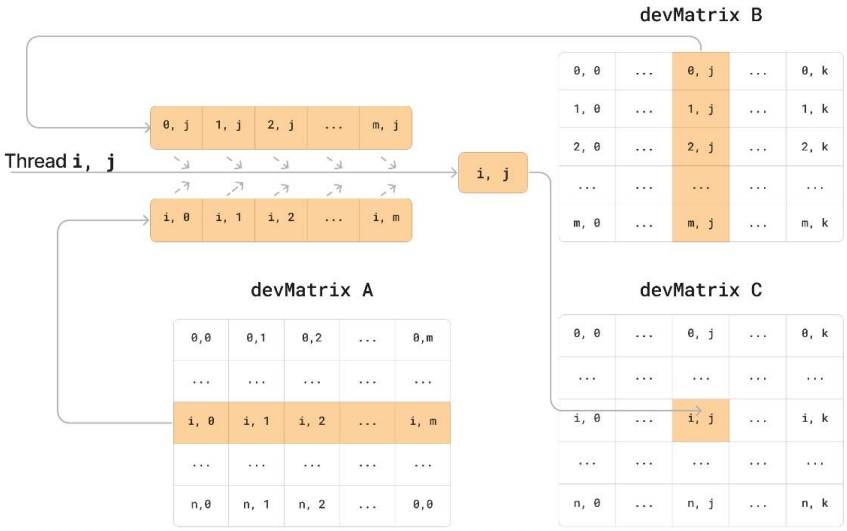

Рисунок 3 - Блок-схема произведения матриц

DOI: https://doi.org/10.60797/IRJ.2024.143.115.4

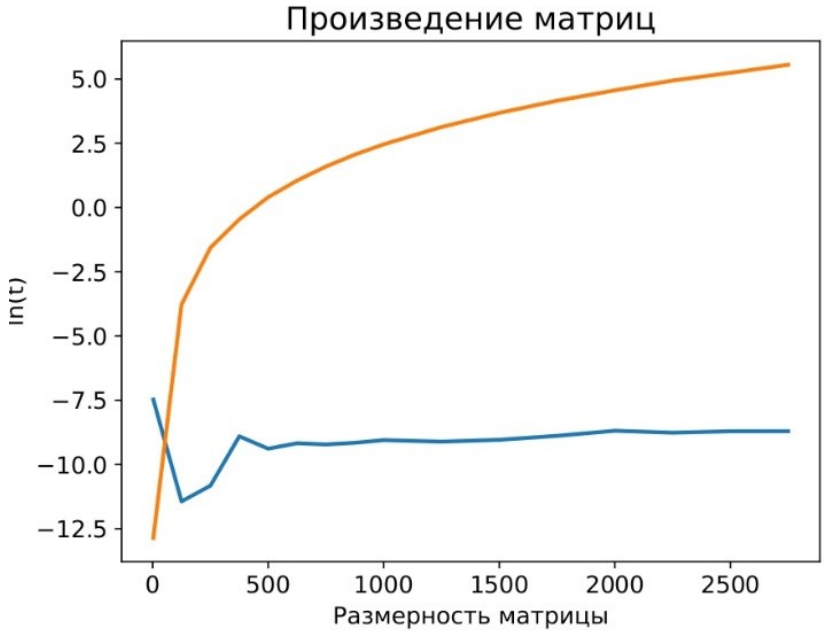

Рисунок 4 - Сравнение производительности умножения матриц на CPU и GPU
DOI: https://doi.org/10.60797/IRJ.2024.143.115.5

Примечание: оранжевая и синяя линии отображают зависимость времени от размерности матрицы для операции умножения в логарифмических координатах на CPU и GPU соответственно

На данную реализацию произведения матриц накладываются те же ограничения, что на суммирование. Разница во времени между перемножением на CPU и GPU, значительно превосходит разницу в случае суммы (см. рис. 4).

Представленный алгоритм является самой простой реализацией параллельного произведения. Существуют альтернативные алгоритмы:

1. Распараллеливание не только вычисления отдельного элемента результирующей матрицы, но и отдельных произведений соответствующей суммы, за счет третьего измерения z в сетке. У данного алгоритма есть две глобальные проблемы. Первая заключается в гонке потоков на запись произведений, появляется она из-за того, что мы вынуждены обращаться не к регистровой памяти, а к глобальной. Причем в подобной реализации в сочетание с нашим представлением класса devMatrix за суммирование произведений одного элемента не всегда отвечает одни блок, поэтому накладывается ограничение на использование разделяемой памяти. Вторая проблема связана с резким уменьшением максимальной размерности матриц из-за уменьшения числа блоков в x, y измерении из-за добавления z измерения:

$N \approx 32 \cdot \sqrt[3]{2^{31}-1}$ т. к. $\frac{N}{32}$ должно быть кратно 32: $N = 40960$.

2. Рассматривать сетку в двух измерениях соответствуя элементам результирующей матрицы. В каждом блоке будет происходить распараллеленное вычисление суммы произведений с использованием разделяемой памяти. Данный алгоритм позволит вычислять произведения матриц с самой высокой скоростью, но если будет происходить перемножение матриц $A_{n\times m} \cdot B_{m\times k}$, то максимальное значение k (число столбцов матрицы A, число строк матрицы B) равно 1024, из-за ограничения на число потоков внутри блока.

По сравнению с указанными альтернативами, классический (простой) алгоритм предпочтителен, он обеспечивает хорошую скорость около граничной области размерности матриц по памяти.

Метод сопряженных градиентов

Пусть система из n линейных алгебраических уравнений с n неизвестными представлена в матричной форме: $A \cdot X = B$,

где $A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$ – основная матрица системы,

$X = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}$ – столбец неизвестных, $B = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}$ – столбец свободных членов.

Метод сопряженных градиентов, подразумевает, что основная матрица системы A – симметрична и положительно определена, т. е. $\forall x \neq \theta \in \mathcal{R}^n : (A \cdot x, x) > 0$ ·

Для решения такой системы методом сопряженных градиентов можно использовать следующее рекуррентное соотношение:

Шаг 0:

$$r_0 = A \cdot X_0 - B, \beta_0 = \frac{(r_0, r_0)}{(A \cdot r_0, r_0)}, \Delta X_1 = \beta_0 \cdot r_0, \alpha_0 = 0 \quad (5)$$

$X_0 = \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \dots \\ \tilde{x}_n \end{pmatrix}$ – произвольный вектор.

Шаг индукции $i(i = \overline{1, n-1})$

$$X_i = X_i + \Delta X_i, \text{ где } \Delta X_i = \alpha_{i-1} \cdot \Delta X_{i-1} + \beta_{i-1} \cdot r_{i-1} (i \neq 1) \quad (6)$$

$$r_i = A \cdot X_i - B, \quad (7)$$

$$\alpha_i = \frac{(r_i, r_i) \cdot (A \cdot r_i, \Delta X_i) - (r_i, \Delta X_i) \cdot (A \cdot r_i, r_i)}{(A \cdot \Delta X_i, \Delta X_i) \cdot (A \cdot r_i, r_i) - (A \cdot r_i, \Delta X_i)^2} \quad (8)$$

$$\beta_i = \frac{(r_i, \Delta X_i) \cdot (A \cdot r_i, \Delta X_i) - (r_i, r_i) \cdot (A \cdot \Delta X_i, \Delta X_i)}{(A \cdot \Delta X_i, \Delta X_i) \cdot (A \cdot r_i, r_i) - (A \cdot r_i, \Delta X_i)^2}, \quad (9)$$

Отметим, что метод сопряженных градиентов всегда сходится к решению исходной СЛАУ за $k \leq n$ итераций (n – порядок системы).

```
devMatrix CGP_GPU(devMatrix& A, devMatrix& B, devMatrix& X0) {
    //Расчет начальных значений
    double alpha = 0;
    devMatrix r = A * X0 - B;
    double betta = -scalar_product(r, r) / scalar_product(A * r, r);
    devMatrix delta = r*betta;
    devMatrix X = X0 + delta;

    for (size_t k = 1; k < A.N(); k++) {
        r = A * X - B;
        double2 alpha_betta = alpha_betta_calculation(r, delta, (A*r),
(A*delta), A.N());

        alpha = alpha_betta.x;
        betta = alpha_betta.y;
        delta = delta*alpha + r*betta;
        X = X + delta;
    }
    return X;
}
```

Рисунок 5 - Вид итогового алгоритма с учетом построенных инструментов для работы с матрицами

DOI: https://doi.org/10.60797/IRJ.2024.143.115.6

Расчет коэффициентов α, β происходит в специальной функции, которая распараллеливает скалярное произведение. Ядро реализовали с использованием редукции (см. рис. 6). Разбиение скалярных произведений происходит за счет добавления измерения y с индексацией $i = \overline{0, 5}$.

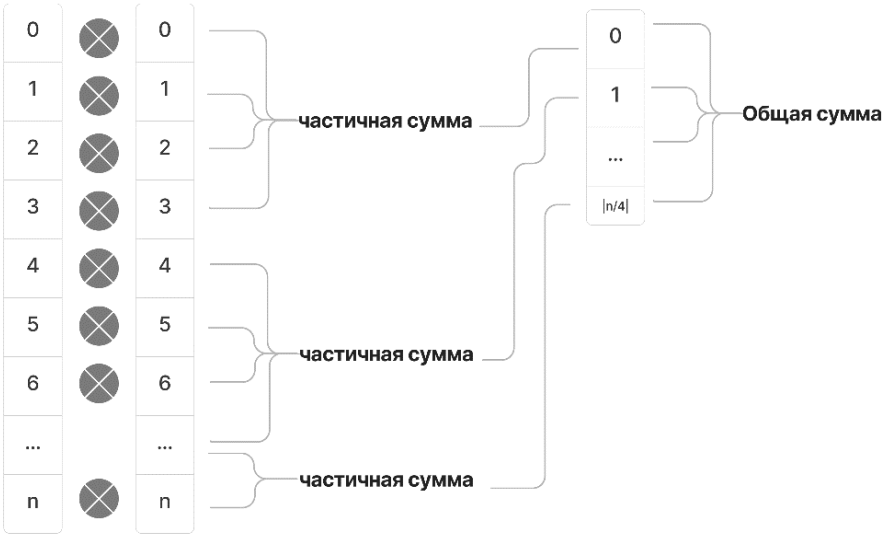

Рисунок 6 - Блок-схема реализации скалярного произведения
DOI: https://doi.org/10.60797/IRJ.2024.143.115.7

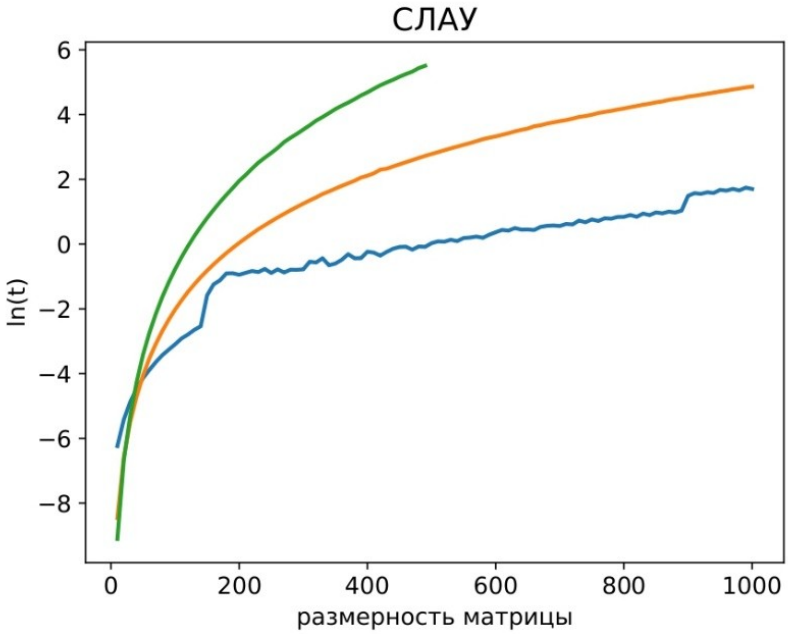

Рисунок 7 - Метод Крамера, спряжённых градиентов на CPU и GPU
DOI: https://doi.org/10.60797/IRJ.2024.143.115.8

Примечание: зеленая, оранжевая и синяя линии отображают зависимость времени от размерности основной матрицы системы при решении СЛАУ в логарифмических координатах на CPU методом Крамера и МСГ и МСГ на GPU соответственно

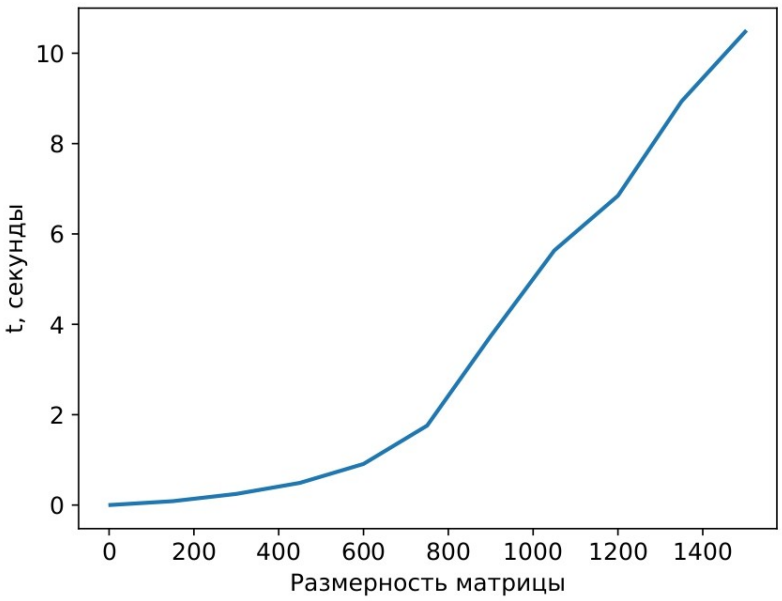

Рисунок 8 - Метод спряжённых градиентов на GPU
DOI: https://doi.org/10.60797/IRJ.2024.143.115.9

Примечание: зависимость времени от размерности основной матрицы системы при решении СЛАУ методом МСГ на GPU

Итоговое сравнение производительности различных методов решения СЛАУ, показывает, что распараллеливание на GPU сильно уменьшает время (см. рис. 7, см рис. 8).

Все тесты проводились на следующей конфигурации:

Процессор: AMD Ryzen 5 5600 6-Core Processor 3.50 GHz;

Оперативная память: 16,0 ГБ;

Графический процессор: GeForce RTX 3060 12,0 ГБ, 3584 ядер CUDA.

Заключение

В настоящей работе предложены основные инструменты для обработки матричных операций с использованием технологии CUDA. На примере сложения и умножения матриц показано, что алгоритмы, построенные на распараллеливании вычислений, работают на несколько порядков эффективнее аналогичных последовательных программ на CPU. Определены границы размерностей матриц, которые можно обработать на GPU.

На основе библиотеки devMatrix, построен алгоритм решения СЛАУ методом сопряженных градиентов, который превзошел аналогичный алгоритм на CPU и метод Крамера по скорости решения. Полученный результат демонстрирует актуальность технологии CUDA в контексте научных расчетов, а также предоставляет возможность написания простых, но в то же время эффективных матричных алгоритмов на базе библиотеки devMatrix.

Финансирование

Работа выполнена на основе гранта Министерства образования РФ № FEUZ-2023-0013.

Конфликт интересов

Не указан.

Рецензия

Все статьи проходят рецензирование. Но рецензент или автор статьи предпочли не публиковать рецензию к этой статье в открытом доступе. Рецензия может быть предоставлена компетентным органам по запросу.

Funding

The work was carried out on the basis of a grant from the Ministry of Education of the Russian Federation No. FEUZ-2023-0013.

Conflict of Interest

None declared.

Review

All articles are peer-reviewed. But the reviewer or the author of the article chose not to publish a review of this article in the public domain. The review can be provided to the competent authorities upon request.

Список литературы / References

1. Крылов В. И. Вычислительные методы в 2 т. / В. И. Крылов, В. В. Бобков, П. И. Монастырный. — Москва : Наука, 1976.

2. Некрасов К. А. Параллельные вычисления общего назначения на графических процессорах / К. А. Некрасов, С. И. Поташников, А. С. Боярченков [и др.]. — Изд-во Уральского ун-та, 2016. — 103 с. — ISBN 978-5-7996-1722-6.

3. Тумаков Д. Н. Технология программирования CUDA / Д. Н. Тумаков, Д. Е. Чикрин, А. А. Егорчев [и др.]. — Казань : Казанский федеральный ун-т. — ISBN 978-5-00019-913-8.

4. Локтионов И. К. Численные методы / И. К. Локтионов. — Москва : Инфа-Инженерия, 2022. — 308 с. — ISBN 978-5-9729-0786-1.

5. Огородникова О. М. Вычислительные методы в компьютерном инжиниринге / О. М. Огородникова. — Изд-во Уральского ун-та, 2013. — 130 с. — ISBN 978-5-7996-0816-3.

6. Амосов А. А. Вычислительные методы для инженеров / А. А. Амосов, Ю. А. Дубинский, Н. В. Копченова. — Москва : Высш. шк, 1994. — 543 с. — ISBN 5-06-000625-5.

7. Пирумов У. Г. Численные методы / У. Г. Пирумов, В. Ю. Гидаспов, И. Э. Иванов [и др.]. — Москва : Юрайт, 2023. — 421 с. — ISBN 978-5-534-03141-6.

8. Снытиков А. В. Математическое моделирование и программная модель CUDA / А. В. Снытников, А. С. Колганов, Н. Н. Попова. — Москва : МАКС Пресс, 2018. — 171 с. — ISBN 978-5-317-05911-8.

9. Тоуманен Б. Программирование GPU при помощи Python и CUDA / Б. Тоуманен. — Москва : ДМК Пресс, 2020. — 235 с. — ISBN 978-5-97060-821-0.

10. Programming Guide // CUDA Toolkit Documentation. — URL: https://docs.nvidia.com/cuda/archive/11.4.0/cuda-c-programming-guide/index.html (accessed: 24.11.2023).

Список литературы на английском языке / References in English

1. Krylov V. I. Vychislitel'nye metody v 2 t. [Computational methods in 2 volumes] / V. I. Krylov, V. V. Bobkov, P. I. Monastyrny. — Moscow : Nauka, 1976. [in Russian]

2. Nekrasov K. A. Parallel'nye vychislenija obshhego naznachenija na graficheskih processorah [General-purpose parallel computing on graphics processors] / K. A. Nekrasov, S. I. Potashnikov, A. S. Boyarchenkov [et al.]. — Publishing House of the Ural University, 2016. — 103 p. — ISBN 978-5-7996-1722-6. [in Russian]

3. Tumakov D. N. Tehnologija programmirovanija CUDA [CUDA programming technology] / D. N. Tumakov, D. E. Chirkin, A. A. Egorchev [et al.]. — Kazan : Kazan Federal University. — ISBN 978-5-00019-913-8. [in Russian]

4. Loktionov I. K. Chislennye metody [Numerical methods] / I. K. Loktionov. — Moscow : Infa-Engineering, 2022. — 308 p. — ISBN 978-5-9729-0786-1. [in Russian]

5. Ogorodnikova O. M. Vychislitel'nye metody v komp'juternom inzhiniringe [Computational methods in computer engineering] / O. M. Ogorodnikova. — Publishing house of the Ural University, 2013. — 130 p. — ISBN 978-5-7996-0816-3. [in Russian]

6. Amosov A. A. Vychislitel'nye metody dlja inzhenerov [Computational methods for engineers] / A. A. Amosov, Yu. A. Dubinsky, N. V. Kopchenova. — Moscow : Higher School of Economics, 1994. — 543 p. — ISBN 5-06-000625-5. [in Russian]

7. Pirumov U. G. Chislennye metody [Numerical methods] / U. G. Pirumov, V. Y. Gidaspov, I. E. Ivanov [et al.]. — Moscow : Yurait, 2023. — 421 p. — ISBN 978-5-534-03141-6. [in Russian]

8. Snytnikov A. V. Matematicheskoe modelirovanie i programmnaja model' CUDA [Mathematical modeling and the CUDA software model] / A. V. Snytnikov, A. S. Kolganov, N. N. Popova. — Moscow : MAKS Press, 2018. — 171 p. — ISBN 978-5-317-05911-8. [in Russian]

9. Tumanen B. Programmirovanie GPU pri pomoshhi Python i CUDA [GPU programming using Python and CUDA] / B. Tumanen. — Moscow : DMK Press, 2020. — 235 p. — ISBN 978-5-97060-821-0. [in Russian]

10. Programming Guide // CUDA Toolkit Documentation. — URL: https://docs.nvidia.com/cuda/archive/11.4.0/cuda-c-programming-guide/index.html (accessed: 24.11.2023).