УДК 004.415.2
DOI: 10.18523/2617-3808.2021.4.10-15

Малашонок Г. І., Сухарський С. С.

АЛГОРИТМ ОБЧИСЛЕННЯ ДВОДІАГОНАЛЬНОЇ МАТРИЦІ ОРТОГОНАЛЬНИМ РОЗКЛАДАННЯМ НА ГРАФІЧНОМУ ПРОЦЕСОРІ

У роботі розглянуто та реалізовано алгоритм ортогонального розкладання матриці, який є першою частиною алгоритму SVD. Наведено реалізацію бідіагоналізації матриці та обчислення ортогональних множників методом Хаусхолдера в середовищі jCUDA на графічному процесорі, а також реалізовано алгоритм для центрального процесора для порівнянь. Проведено дослідження отриманих результатів, у яких експериментально визначалось пришвидшення обчислень за рахунок використання графічного процесора, порівняно з обчисленнями на центральному процесорі. Для матриці розміру 2048 використання відеокарти дає змогу пришвидшити обчислення у 53 рази.

Ключові слова: графічний процесор, центральний процесор, матриця, вектор, Хаусхолдер, CUDA.

Вступ

Із розвитком царини Big Data та сучасних розділів інформатики, які належать до штучного інтелекту, потреба у швидкісних та ефективних обчисленнях стала одним із найважливіших завдань сьогодення. З такою метою протягом останніх більш як десяти років активно розвивається галузь обрахунків на графічних процесорах, які містять тисячі ядер. Метою цього дослідження є реалізація алгоритму ортогонального розкладу з використанням методу Хаусхолдера та проєкту jCUDA [3] для проведення обчислень на відеокарті в мові Java та дослідження його переваг. Наприклад, у роботі Савченка [5] досліджувався алгоритм SVD із використанням прямого QR алгоритму, який показав пришвидшення щодо ітеративних версій, проте підхід із використанням методу Хаусхолдера видається перспективнішим за ефективністю обчислень. Схожий підхід із використанням серії трансформацій Хаусхолдера використовували у своїй роботі Лагабар і Нараянан [4], проте вони все ще не запропонували хорошого способу для роботи з великими матрицями, в яких можуть накопичуватись похибки зі збільшенням розміру матриць. Вони використовували тип даних *Float*.

У цій роботі ми працюємо з *Double*, а також плануємо реалізувати тип *BigDecimal* для відеокарт і працювати далі з ним. Середовище jCUDA було обрано для подальшої інтеграції з великим комплексом бібліотек для паралельного програмування Mathpartner.

Бідіагоналізація

Алгоритм. На цьому етапі ми застосовуємо метод Хауслхолдера для обрахунку ортогонального розкладу:

$$A = UDV^T,$$

де V, U – ортогональні (унітарні для поля комплексних чисел) матриці, а D – дводіагональна (тридіагональна у випадку, коли A симетрична) матриця [6]. Можна легко побудувати ортогональну матрицю Хаусхолдера, яка при множенні на заданий ненульовий вектор x перетворює його на $y = Px$, так, щоб змінилось значення не більше ніж однієї компоненти і при цьому обнулились будь-які інші компоненти цього вектора. Наприклад, нехай потрібно обнулити всі компоненти, окрім першої, тоді ми отримаємо:

$$y_1 = \left(||x||, 0, .., 0\right) \text{ або } y_2 = \left(-||x||, 0, .., 0\right).$$

Для зменшення похибки обчислень беруть такий варіант:

$$y = \left(-sign(x_1)||x||, 0, .., 0\right),$$

$$u = x - y = \left(x_1 + sign(x_1)||x||, x_2, .., x_n\right).$$

Приклад. Нехай дано матрицю:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 1 \end{pmatrix}.$$

Визначимо вектор x як перший стовпчик та обчислимо такі компоненти:

$$x = (1 \quad 1 \quad 1 \quad 1)^T;\ norm_x = (x_1^2 + .. + x_n^2)^{1/2};$$
$$u = (1 + norm_x \quad 1 \quad 1 \quad 1)^T;$$
$$P_1 = I_{4,4} - (2/norm_x^2) \cdot u \cdot u^T.$$

Позначимо через dT вектор-рядок, отриманий у результаті множення:

$$d_T = (u^T \cdot A).$$

Тоді добуток $P_1 \cdot A$ дорівнюватиме:

$$A_1 = P_1 \cdot A = \left(I_{4,4} - (2/norm_x^2) \cdot u \cdot u^T\right) \cdot A =$$
$$= A - (2/norm_x^2) \cdot u \cdot (u^T \cdot A) = A - (2/norm_x^2) \cdot u \cdot d_T;$$

$$P_1 = \begin{pmatrix} -0.5 & -0.5 & -0.5 & -0.5 \\ -0.5 & 0.83 & -0.17 & -0.17 \\ -0.5 & -0.17 & 0.83 & -0.17 \\ -0.5 & -0.17 & -0.17 & 0.83 \end{pmatrix},$$

$$A_1 = \begin{pmatrix} -2 & -2.5 & -2.5 & -2.5 \\ 0 & -0.17 & 0.83 & -0.17 \\ 0 & -0.17 & -0.17 & 0.83 \\ 0 & 0.83 & -0.17 & -0.17 \end{pmatrix}.$$

Зазначимо, що множення $dT = (u^T \cdot A)$ дуже ефективне. Рядок будемо множити на матрицю.

$$dT = (u^T \cdot A) = \sum_{i=1}^{n} u_i \cdot A_i.$$

Тобто елемент u^T з номером i множиться на всі елементи рядка i та додається до результату. Всього n^2 операцій множення і приблизно стільки ж операцій додавання.

Наша мета – звести матрицю A до дводіагонального виду, тому тепер потрібно обнулити перший рядок. При цьому не можна змінювати перший стовпчик. Отримаємо:

$$x = (0 \quad -2.5 \quad -2.5 \quad -2.5);$$
$$u = (0 \quad -2.5 - norm_x \quad -2.5 \quad -2.5);$$
$$Q_1 = I_{4,4} - (2/norm_x^2) \cdot u^T \cdot u;$$
$$d = (A_1 \cdot u^T);$$
$$A_2 = A_1 \cdot Q_1 = A_1 \cdot \left(I_{4,4} - (2/norm_x^2) \cdot u^T \cdot u\right) =$$
$$= A_1 - A_1 \cdot (2/norm_x^2) \cdot u^T \cdot u = A_1 - (2/norm_x^2) \cdot d \cdot u;$$

$$Q_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.58 & -0.58 & -0.58 \\ 0 & -0.58 & 0.79 & -0.21 \\ 0 & -0.58 & -0.21 & 0.79 \end{pmatrix};$$

$$A_1 = \begin{pmatrix} -2 & 4.33 & 0 & 0 \\ 0 & -0.29 & 0.79 & -0.21 \\ 0 & -0.29 & -0.21 & 0.79 \\ 0 & -0.29 & -0.58 & -0.58 \end{pmatrix}.$$

Такі самі дії будемо виконувати далі для наступних стовпчиків і рядків матриці, поки не дійдемо до моменту, коли матриця буде дводіагональною [2]. Ми обраховуємо норму вектора x у два етапи та перевіряємо, чи друга частина норми дорівнює нулю. Якщо це так, тоді нам не потрібно виконувати поточний крок і ми переходимо до наступного.

Нам також потрібно обраховувати лівий і правий множники. Задля ефективного використання пам'яті, ми обраховуємо їх, як тільки знайдено поточний співмножник. Обрахунок множників Хаусхолдера відбувається за $2n^2$ операцій відповідно до формул:

$$U = Pi - (2/norm_x^2) \cdot (Pi \cdot u^T) \cdot u;$$
$$V = Qi - (2/norm_x^2) \cdot u^T \cdot (u \cdot Qi).$$

Перевірку отриманого розкладу матриці виконуємо так:

$$B = U \cdot A_4 \cdot V^T; B - A;$$
$$Ch_U = (U^{-1} - U^T); Ch_v = (W^{-1} - W^T).$$

У результаті ми очікуємо, що матриці **Check**, **ChU** та **ChV** будуть нульовими (більшість елементів наближені до нуля та не більші за 10^{-15}). Саме знаходженням найбільшого абсолютного значення в матриці ми визначаємо похибку в обчисленнях.

Алгоритм. Ортогональне розкладання матриці до дводіагонального виду

1: N = A.size – 1, i = 0;
2: **while** i < N **do**
3: x = the low part of *i-th* column of matrix **A**, starting from *i-th* row
4: x1 = the low part of x: the column without one upper component (of i-th row)
5: **if** $norm_{x1} > 10^{-10}$ **then**
6: apply $norm_x$ to ***i*** element of **x**
7: **if** *U* is not initialized **then**
8: $U = I - (2/norm_x^2) * x * x^T$
9: **else**
10: $U = U - (2/norm_x^2) * (U * x^T) * x$
11: **end if**
12: $A_{i+1} = A_i - (2/norm_x^2) * x * (x^T * A_i)$
13: **end if**
14: x = ***i*** row of matrix **A** starting from *(i+1)-th* column
15: x1 = the right part of x: the row without one left component
16: **if** $norm_{x1} > 10^{-10}$ **then**
17: apply $norm_x$ to ***i+1*** element of **x**
18: **if** *V* is not initialized **then**
19: $V = I - (2/norm_x^2) * (x^T * x)$
20: **else**
21: $V = V - (2/norm_x^2) * x^T * (x * V)$
22: **end if**
23: $A_{i+1} = A_i - (2/norm_x^2) * (A_i * x^T) * x$
24: **end if**
25: increment *i*
26: **end do**
27: **return** $[U, A_i, V]$

Програмна реалізація. Для подальших порівнянь та досліджень описаний вище алгоритм було реалізовано мовою Java у двох варіантах:

– послідовний алгоритм на центральному процесорі (CPU);
– за допомогою jCUDA на графічному процесорі (GPU).

У бібліотеці mathpar було додано новий клас SVD, в якому реалізовано метод *bidiagonalize (MatrixD A, Ring ring),* що приймає на вхід матрицю *A* та змінну типу Ring, яка визначає алгебраїчний простір поточних змінних. Клас *MatrixD* має готові функції для роботи з матрицями (транспонування, обернення тощо). На виході функція повертає масив із трьома об'єктами MatrixD: U – лівий множник, D – бідіагональна матриця, V – правий множник. Матриця A, що передається у функцію, заповнюється згенерованими числами типу Double в діапазоні від 0 до 10. Ring виставляється R64 – для роботи з дійсними числами. Для роботи з векторами використано клас *VectorS*. Цей клас надає можливість обраховувати норму вектора, множити його на скаляр тощо. Також у програмі використовується клас Element, який використовується і в MatrixD, і в VectorS, що дає можливість без додаткового програмування виконувати додавання, множення та інші операції для цих класів.

Програму під відеокарту було розроблено за схожим алгоритмом, проте усі функції підлаштовано та оптимізовано під виконання на графічному процесорі. Перш за все, програма виконується так, що контроль виконання (виклик функцій та розгалуження логіки) відбувається з центрального процесора, який оптимізований під виконання потоку операцій, а усі інтенсивні обчислення виконуються на відеокарті.

Програма складається з класу SVD, у якому реалізовано метод *biDiagonalize(double[] m),* який приймає на вхід представлену в вигляді стрічки (вектора) матрицю з числами типу Double. На виході метод повертає масив об'єктів типу Pointer – вказівники на об'єкти, розташовані на відеокарті. Більшість методів використано з бібліотек JCublas та JCuda, які надають готові алгоритми та функції для базової лінійної алгебри та для роботи з пам'яттю, за прикладом аналогічних бібліотек з основного пакета NVIDIA Cuda [1]. Крім цього, для деяких частин алгоритму було розроблено власні функції, які виконуються на графічному процесорі.

Експериментальні дослідження

Експерименти проводили на двох різних графічних процесорах і на центральному процесорі, а саме: NVIDIA GeForce MX150, NVIDIA GeForce GTX 1660 та Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz 2.59 GHz. Основну увагу приділено дослідженню швидкості виконання на різних пристроях, за різних розмірів матриці, для визначення часової складності та прискорення щодо інших алгоритмів.

Передусім було перевірено залежність часу виконання від розміру матриці для послідовного алгоритму та алгоритму на відеокарті. Розміри матриці брали від 4 до 2048. Обчислення проводили на операційній системі Windows 10. Нижче можна побачити таблицю та графік із порівнянням часу виконання (дані в секундах).

Ці результати підтверджують наші очікування щодо того, що обчислення на відеокарті є значно швидшими. Як бачимо, на невеликих розмірах матриць (4, 8, 16, 32) різниці фактично немає. Проте вона потроху наростає, і вже на розмірі 128 відбувається пришвидшення у 8 разів. На більших розмірах перевага в часі виконання ще значущіша – різниця в 32 рази за розміру матриці 1024 та в 53 рази на вдвічі більшій розмірності матриці. На графік результати виконання для матриці розміром 2048 вже не виводили, для кращого розуміння тенденції двох кривих.

Таблиця 1

Порівняння часу виконання послідовного та GPU алгоритмів

Розмір матриці	Intel(R) Core(TM) i7-9850H CPU	GeForce MX150	Пришвидшення (у *n* разів)
4	0.002	0.002	1
8	0.005	0.003	1.6
16	0.01	0.006	1.6
32	0.028	0.012	2.3
64	0.104	0.024	4.3
128	0.408	0.051	8
256	2.59	0.172	15.1
512	19.033	0.859	22.2
1024	172.64	5.418	31.86
2048	2179.85	40.975	53.19

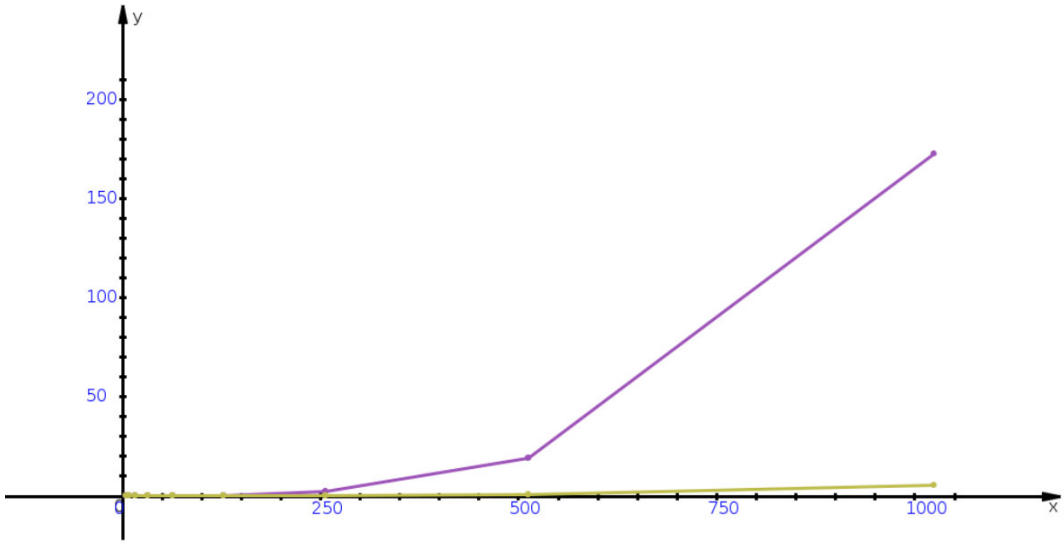

Рис. 1. Порівняння часу виконання послідовного та GPU алгоритмів

Також було проведено дослідження часу виконання на відеокартах різної потужності.

Таблиця 2

Порівняння часу виконання відеокарт різної потужності

Розмір матриці	GeForce MX150	GTX 1660	Пришвидшення (у *n* разів)
4	0.002	0.002	1
8	0.003	0.005	0.6
16	0.006	0.007	0.85
32	0.012	0.013	0.92
64	0.024	0.024	1
128	0.051	0.063	0.8
256	0.172	0.117	1.47
512	0.859	0.359	2.39
1024	5.418	1.639	3.3
2048	40.975	10.824	3.78

Як бачимо, GTX 1660 за рахунок більшої кількості CUDA ядер, пропускної здатності пам'яті та розміру пам'яті виграє за швидкістю виконання. Причому чим більший розмір матриці, тим помітнішою є різниця. Відеокарта приблизно в 4 рази потужніша, і як видно з табл. 2, результати кращі також приблизно в 4 рази на великих розмірах вхідної матриці.

Далі було оцінено часову залежність від розміру матриці та операційної системи. Ці дослідження вже проводили на графічному процесорі GeForce GTX 1660. Числа в матриці – від 0 до 10. Час виконання наведено в табл. 3 (дані в секундах). Експерименти виконано на операційних системах Windows 10 та Arch Linux.

Результати показують, що програма виконується трошки швидше на Arch Linux, проте різниця незначна, особливо вона стає менш помітною на більших розмірах вхідних даних.

Таблиця 3

Порівняння часу виконання на різних операційних системах

Розмір матриці	Windows 10	Arch Linux	Пришвидшення (у *n* разів)
4	0.002	0.002	1
8	0.005	0.002	2.5
16	0.007	0.004	1.75
32	0.013	0.008	1.63
64	0.024	0.016	1.5
128	0.063	0.037	1.7
256	0.117	0.07	1.6
512	0.359	0.261	1.37
1024	1.639	1.448	1.13
2048	10.824	10.593	1.02

Після цього, залишивши тільки результат на операційній системі Arch Linux, було проведено дослідження часової складності. Очікувана складність n^3. Провівши дослідження, виявили схожу криву, з невеликою різницею, що пояснюється певними особливостями обрахунків на графічному процесорі, а саме підбором розмірів сітки та блоку, а також зчитуванням та записом необхідних елементів. Було підібрано параметри та степінь, для побудови кривої, яка відповідає кривій із часом обрахунків. Формула вийшла такою: $1.210^{-9}x^3$.

Насамкінець, було досліджено похибку обчислень. На CPU вона коливалась у проміжку від 10^{-17} до 10^{-15}. На GPU (GTX 1660, Arch Linux) ситуація вийшла іншою. Похибка обчислень з'являється вже на розмірі 128 і дорівнює 10, при цьому вона зникає на 256. Далі знову з'являється на розмірах 512 і більше і коливається у межах від 20 до 28. Було виявлено, що ламаються лише декілька блоків зі всієї матриці, часто останні.

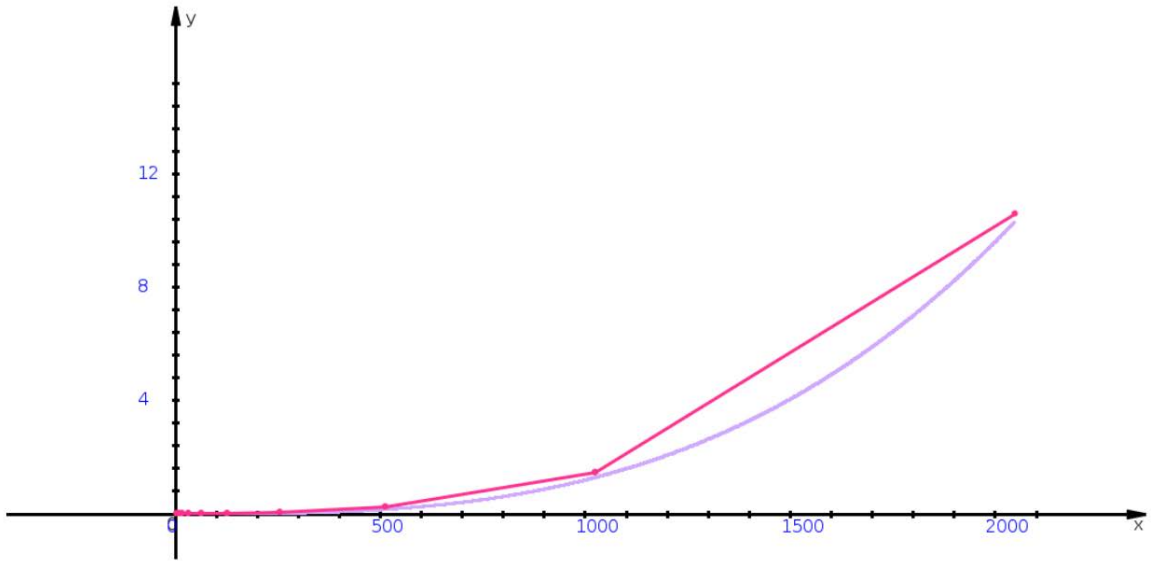

Рис. 2. Оцінка часової складності реалізованого на GPU алгоритму

Наразі ми пов'язуємо це з синхронізацією даних під час виконання, а також особливістю підбору розмірів блоку та сітки для певних розмірностей матриці.

Висновки

Було досліджено алгоритм Хаусхолдера для бідіагоналізації матриці та тридіагоналізації симетричної матриці. Алгоритм було реалізовано на графічному процесорі за допомогою платформи CUDA та середивища JCuda мовою Java. Крім цього, було використано послідовний алгоритм мовою Java для проведення порівняльного аналізу.

Проведено дослідження швидкості виконання залежно від різних розмірів вхідної матриці на відеокарті та на центральному процесорі. Максимальний розмір матриці був 2048, за якого досяглось 53-кратне пришвидшення щодо обчислень на одному центральному процесорі. Також досліджувалась похибка обчислень. Результати показали значну перевагу в швидкості обчислень на графічному процесорі, проте накопичення похибки також було більшим саме на відеокарті. Також було порівняно дві відеокарти різної потужності, і на кращій із них, згідно з характеристиками, ефективність обчислень була ще вищою. Метод Хаусхолдера показав гарні результати, і весь агоритм SVD із використанням цього методу виглядає багатообіцяльним. Наступні дослідження буде проведено після реалізації другого етапу – діагоналізації отриманої на першому етапі бідіагональної матриці.

Список літератури

1. CUBLAS documentation [Electronic resource]. – Mode of access: https://docs.nvidia.com/cuda/cublas/index.html.
2. Golub G. Calculating the Singular Values and Pseudo-Inverse of a Matrix SIAM J. / G. Golub, W. Kahan // Num. Anal. – 1965. – Vol. 2. – Pp. 205–224.
3. JCuda project [Electronic resource]. – Mode of access: http://jcuda.org/.
4. Lahabar S. Singular value decomposition on GPU using CUDA / S. Lahabar, P. J. Narayanan // 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–10 [Electronic resource]. – Mode of access: http://doi.org/10.1109/IPDPS.2009.5161058.
5. Malashonok H. I. Matrychni alhorytmy rozbyttia mnozhyn dlia rekomendatsiinykh system / H. I. Malashonok, S. O. Savchenko. – NaUKMA, 2019.
6. Persson. Householder Reflectors and Givens Rotations, MIT 18.335J / 6.337J *Introduction to Numerical Methods* [Electronic resource]. – Mode of access: https://math.dartmouth.edu/~m116w17/Householder.pdf.

References

CUBLAS documentation. Retrieved from https://docs.nvidia.com/cuda/cublas/index.html.

Golub, G., & Kahan, W. (1965). Calculating the Singular Values and Pseudo-Inverse of a Matrix. SIAM J. *Num. Anal*., *2*, 205–224.

JCuda project. Retrieved from http://jcuda.org/.

Lahabar, S., & Narayanan, P. J. Singular value decomposition on GPU using CUDA. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10. http://doi.org/10.1109/IPDPS.2009.5161058

Malashonok, H. I., & Savchenko S. O. (2019). Matrychni alhorytmy rozbyttia mnozhyn dlia rekomendatsiinykh system. NaUKMA.

Persson. Householder Reflectors and Givens Rotations, MIT 18.335J / 6.337J *Introduction to Numerical Methods.* Retrieved from https://math.dartmouth.edu/~m116w17/Householder.pdf.

G. Malaschonok, S. Sukharskyi

A GPU-BASED ORTHOGONAL MATRIX FACTORIZATION ALGORITHM THAT PRODUCES A TWO-DIAGONAL SHAPE

With the development of the Big Data sphere, as well as those fields of study that we can relate to artificial intelligence, the need for fast and efficient computing has become one of the most important tasks nowadays. That is why in the recent decade, graphics processing unit computations have been actively developing to provide an ability for scientists and developers to use thousands of cores GPUs have in order to perform intensive computations. The goal of this research is to implement orthogonal decomposition of a matrix by applying a series of Householder transformations in Java language using JCuda library to conduct a research on its benefits. Several related papers were examined. Malaschonok and Savchenko in their work have introduced an improved version of QR algorithm for this purpose [4] and achieved better results, however Householder algorithm is more promising for GPUs according to another team of researchers – Lahabar and Narayanan [6]. However, they were using Float numbers, while we are using Double, and apart from that we are working on a new BigDecimal type for CUDA. Apart from that, there is still no solution for handling huge matrices where errors in calculations might occur.

The algorithm of orthogonal matrix decomposition, which is the first part of SVD algorithm, is researched and implemented in this work. The implementation of matrix bidiagonalization and calculation of orthogonal factors by the Hausholder method in the jCUDA environment on a graphics processor is presented, and the algorithm for the central processor for comparisons is also implemented. Research of the received results where we experimentally measured acceleration of calculations with the use of the graphic processor in comparison with the implementation on the central processor are carried out. We show a speedup up to 53 times compared to CPU implementation on a big matrix size, specifically 2048, and even better results when using more advanced GPUs. At the same time, we still experience bigger errors in calculations while using graphic processing units due to synchronization problems. We compared execution on different platforms (Windows 10 and Arch Linux) and discovered that they are almost the same, taking the computation speed into account. The results have shown that on GPU we can achieve better performance, however there are more implementation difficulties with this approach.

Keywords: graphics processor, central processor, matrix, vector, Householder, CUDA.

Матеріал надійшов 14.06.2021