УДК 004.415.5
DOI: 10.18523/2617-3808.2021.4.16-22

Семилітко М. Ю., Малашонок Г. І.

ПАРАЛЕЛЬНИЙ SVD АЛГОРИТМ ДЛЯ ТРИДІАГОНАЛЬНОЇ МАТРИЦІ НА ВІДЕОКАРТІ З ВИКОРИСТАННЯМ АРХІТЕКТУРИ NVIDIA CUDA

Ця робота пропонує реалізацію паралельного алгоритму SVD для тридіагональної матриці на відеокарті з використанням архітектури Nvidia CUDA для роботи з великими матрицями. Для цього було досліджено роботу послідовного алгоритму, розроблено модель паралельного алгоритму на Java, який враховує особливості роботи відеокарти, і реалізовано та протестовано алгоритми для відеокарти з використанням різних типів пам'яті відеокарти, які можна використовувати у програмах на Java та C/C++.

Ключові слова: сингулярний розклад матриці, SVD, Nvidia CUDA, Java, C++.

Вступ

Алгоритм SVD (Singular Value Decomposition – сингулярний розклад матриці) використовують у рекомендаційних системах, машинному навчанні, обробленні зображення, різних алгоритмах роботи з матрицями, які можуть бути дуже великого розміру, та Big Data. Тому, враховуючи особливості роботи цього алгоритму, він може виконуватися на великій кількості обчислювальних потоків, які мають тільки відеокарти. Такий підхід дасть змогу зменшити час обчислень, а отже й зменшити кількість ресурсів та грошових витрат.

Паралельний SVD алгоритм для тридіагональної матриці

Послідовний алгоритм

SVD алгоритм для тридіагональної матриці M приводить до діагонального виду та повертає три матриці: L, діагональну матрицю M' та R, добуток яких повертає початкову тридіагональну матрицю M.

Тридіагональна матриця має ненульові елементи на головній діагоналі та на діагоналі, яка розташована над або під головною діагоналлю. На рис. 1 зображено приклад тридіагональної матриці M із ненульовими елементами над головною діагоналлю.

Алгоритм складається з двох кроків, які виконуються для кожного квадрата на головній діагоналі, які зображені на рис. 2. Ці кроки виконуються доти, доки всі елементи нижньої та верхньої діагоналей не стануть нульовими.

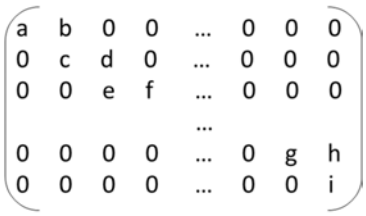


Рис. 1. Приклад тридіагональної матриці M із ненульовими елементами над головною діагоналлю

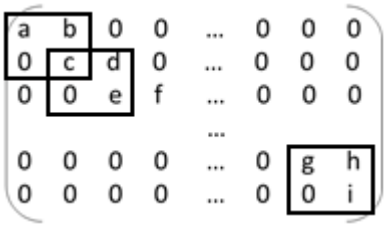


Рис. 2. Елементи матриці, які оброблюються алгоритмом

На першому кроці, до першого квадрата з рис. 2 застосовується матриця обертання Ґівенса, що обнулює значення над або під головною діагоналлю. Для цього обчислюється косинус та синус для матриці повороту за такою формулою:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix},$$

$$r = \sqrt{a^2 + b^2}, c = a/r, s = b/r,$$

і матриця Ґівенса матиме такий вигляд (рис. 3).

При множенні вхідної матриці на матрицю Ґівенса, вхідна матриця набере такого вигляду (рис. 4), де значення верхнього правого елементу квадрата стало нульовим, а значення нижнього стало ненульовим:

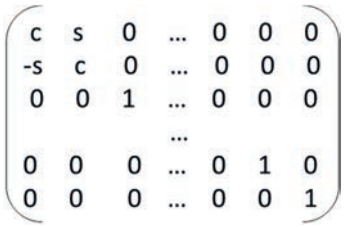


Рис. 3. Матриця обороту Ґівенса для першого квадрата матриці M

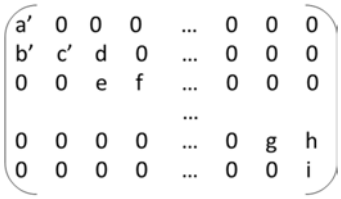


Рис. 4. Матриця M після множення на матрицю Ґівенса

На другому кроці отримана матриця повороту також множиться на матрицю L або R, залежно від того, обнулюємо ми значення над або під головною діагоналлю, тобто якщо ми обнулюємо елемент над головною діагоналлю, тоді матриця Ґівенса множиться на матрицю R, інакше на матрицю L. На початку роботи алгоритму матриці L та R – це одиничні матриці, такого самого розміру, як і вхідна матриця M.

Є випадки, коли на першому кроці у квадраті два недіагональні елементи нульові. В такому разі ітерація не виконується.

Коли всі елементи на верхній та нижній діагоналях стануть нульовими, алгоритм завершує свою роботу.

Паралельний алгоритм

Паралельний алгоритм виконує два кроки одночасно для кількох блоків, при цьому кількість опрацьованих блоків збільшується кожні 2 ітерації. На першій ітерації опрацьовується один квадрат, на третій – два, і так далі до максимальної кількості потоків, як зображено на рис. 5, де кожен потік – це окремий прямокутник. Максимальна кількість потоків, яка може використовуватись, дорівнює половині розміру матриці.

Тепер розглянемо, як змінюються матриці L та R при виконанні другого кроку. Нехай маємо квадратні матриці L та R, розмір яких 6, і вони складаються з будь-яких елементів, і матрицю повороту Ґівенса G. Перемноживши L та R на матрицю G, отримаємо результати, які зображено на рис. 6 і 7.

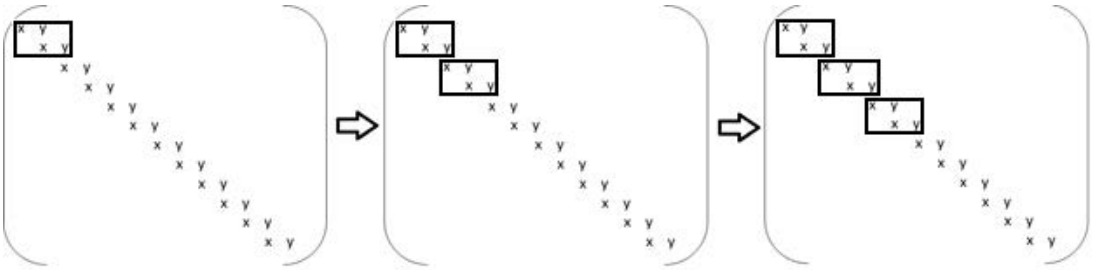


Рис. 5. Приклад збільшення кількості одночасних ітерацій над діагоналлю, де перша матриця – це перша ітерація, друга – третя ітерація, третя – п'ята ітерація

Рис. 6. Результат множення матриць G та L

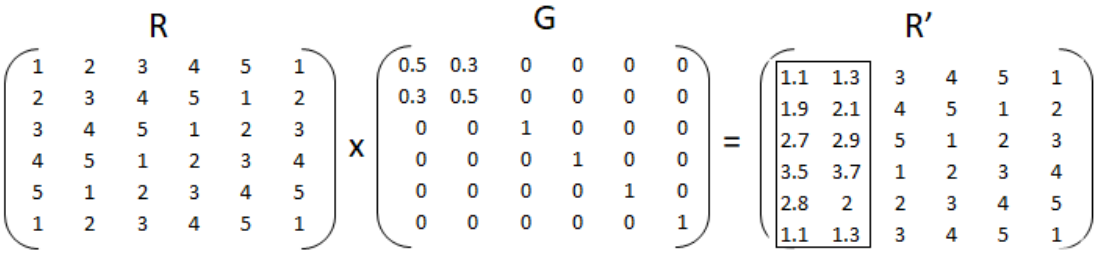


Рис. 7. Результат множення матриць R та G

Із результатів видно, що при множенні матриць G на L та R на G для матриці L змінюються ті самі рядки, як і в тридіагональній матриці, а в матриці R – ті самі стовпчики. Отже, якщо всі потоки будуть синхронізовано пересуватись по діагоналі матриці з кроком у два елементи, тоді синхронізувати доступ до матриць M, L та R нам не потрібно.

Оптимізація роботи з пам'яттю

В алгоритмі використовують два типи пам'яті – глобальну та спільну пам'ять потоків, причому друга є набагато швидшою, але й меншою за розміром [3].

Тому в алгоритмі тридіагональна матриця (рис. 8) буде представлена у вигляді одновимірного масиву, в якому є лише три діагоналі (рис. 9), що дасть змогу зменшити розмір більше ніж у сім разів, а зміщення для елементів матриці будуть обчислюватись самостійно [1].

В останніх поколіннях відеокарт розмір спільної пам'яті блоків дещо коливається, тому візьмемо за основу звичний розмір для попереднього покоління у 48 Кбайт, у який поміщається 6144 елементи з плаваючою комою подвійної точності. Враховуючи оптимізацію зі зберігання тридіагональної матриці, в один блок поміщається квадратна матриця розміром 2046 на 2046.

Для деяких напрямів, таких як Big Data або рекомендаційні системи, матриці можуть бути більшого розміру, тому велика тридіагональна матриця буде розбита між блоками, як зображено на рис. 10, що дасть змогу збільшити кількість обчислювальних потоків та зберігати матрицю у спільній пам'яті, але виникає задача щодо синхронізації спільних значень, наприклад значення 28, яке міститься в першому та другому блоках.

Оскільки для всіх блоків кількість потоків завжди однакова, то і розміри матриць також мають бути однаковими. Якщо розмір матриці для останнього блоку менший, то він вирівнюється нульовими значеннями (рис. 11), що не впливає на швидкість роботи алгоритму.

Оптимізація множення для кроків алгоритму

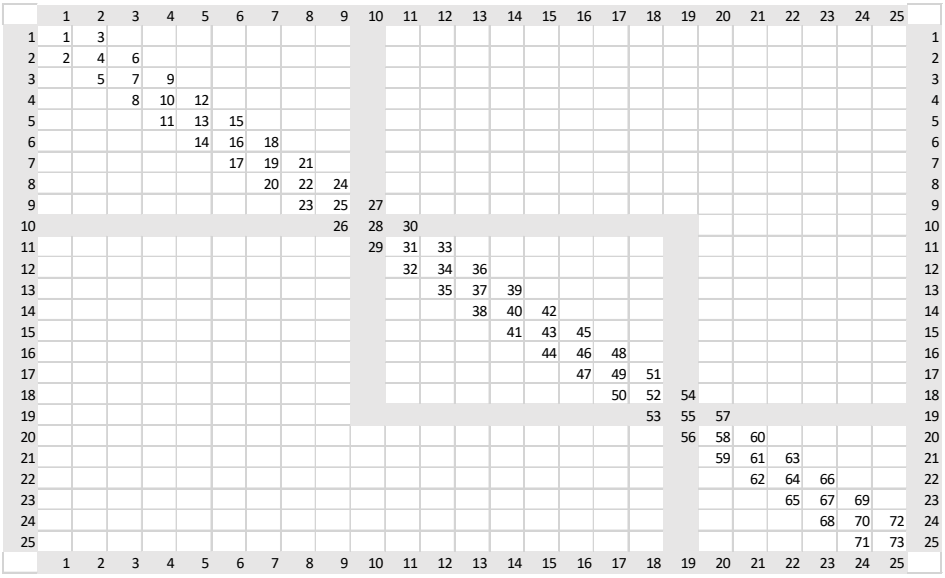
Для зберігання матриці Ґівенса досить знати два числа, які визначають відповідний 2×2 блок повороту. Цей блок розміщується на діагоналі. При множенні на таку матрицю справа змінюватимуться лише два стовпчики, а при множенні зліва будуть змінюватися тільки два рядки. Тому

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
1	1	3																								1
2	2	4	6																							2
3		5	7	9																						3
4			8	10	12																					4
5				11	13	15																				5
6					14	16	18																			6
7						17	19	21																		7
8							20	22	24																	8
9								23	25	27																9
10									26	28	30															10
11										29	31	33														11
12											32	34	36													12
13												35	37	39												13
14													38	40	42											14
15														41	43	45										15
16															44	46	48									16
17																47	49	51								17
18																	50	52	54							18
19																		53	55	57						19
20																			56	58	60					20
21																				59	61	63				21
22																					62	64	66			22
23																						65	67	69		23
24																							68	70	72	24
25																								71	73	25
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	

Рис. 8. Приклад тридіагональної матриці розміром 25 на 25

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
73																							

Рис. 9. Приклад збереження тридіагональної матриці розміром 25 на 25 у вигляді вектора



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
1	1	3																								1
2	2	4	6																							2
3		5	7	9																						3
4			8	10	12																					4
5				11	13	15																				5
6					14	16	18																			6
7						17	19	21																		7
8							20	22	24																	8
9								23	25	27																9
10									26	28	30															10
11										29	31	33														11
12											32	34	36													12
13												35	37	39												13
14													38	40	42											14
15														41	43	45										15
16															44	46	48									16
17																47	49	51								17
18																	50	52	54							18
19																		53	55	57						19
20																			56	58	60					20
21																				59	61	63				21
22																					62	64	66			22
23																						65	67	69		23
24																							68	70	72	24
25																								71	73	25
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	

Рис. 10. Приклад збереження тридіагональної матриці розміром 25 на 25 у трьох блоках

														Block 1												
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
														Block 2												
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
														Block 3												
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	0	0	0	0	0	0	0	0	0

Рис. 11. Приклад збереження оптимізованої тридіагональної матриці розміром 25 на 25 у трьох блоках розміру 28

процедура множення на матрицю Ґівенса вимагає тільки 4·n операцій множення. Отже, для всіх оптимізацій маємо дві функції:

1) функція для оброблення квадрата, коли елемент над діагоналлю не є нульовим;
2) функція для оброблення квадрата, коли елемент під діагоналлю не є нульовим.

Оскільки множення матриці M на G та L, R на G відбуваються в одному потоці, можна провести два кроки алгоритму відразу після першого.

Функції працюють у такий спосіб:

1) обчислення зміщення для елементів у тридіагональній матриці;
2) перевірка значення елемента, який буде обнулений; якщо абсолютне значення менше точності, тоді перетворення не відбувається, у такому разі переходимо до пункту 3;
3) обчислення синуса та косинуса для матриці повороту;
4) множення елементів;
5) запис нових значень у тридіагональну матрицю;
6) обчислення зміщення щодо позиції квадрата для матриці L (R);
7) множення елементів та запис нових значень до матриці L (R) у глобальній пам'яті.

Синхронізація значень, потоків та блоків

Одна із найважчих та найважливіших задач паралельного алгоритму, який використовує декілька блоків, – це міжблочна синхронізація потоків і синхронізація спільних елементів матриці для блоків.

Потоки у блоках синхронізуються вбудованою функцією синхронізації __syncthreads(), при цьому коли потік опрацьовує значення, яке є у двох блоках, він спочатку зчитує нове значення з глобальної пам'яті, проводить обчислення, після чого оновлює значення у глобальній пам'яті. Для цього були розроблені функції запису та читання з глобальної пам'яті [2].

Функція запису в глобальну пам'ять оновлює значення, а потім перевіряє, щоб це значення там з'явилось. Такий процес триватиме, доки не з'явиться нове значення.

Функція оновлення значення зчитує значення з глобальної пам'яті доти, доки не зчитається значення, яке відрізняється від поточного.

Використання цих функцій забезпечує коректний запис та оновлення даних при роботі з глобальною пам'яттю, тому їх використання дасть змогу синхронізувати потоки та блоки таким чином:

- усі потоки в блоках синхронізуються за допомогою функції __syncthreads();

- один потік у середині блоку чекатиме на оновлене значення від іншого блоку, у такий спосіб ніби зависаючи;
- коли потоки зчитують нове значення, вони доходять до функції __syncthreads() і таким чином запускають інші потоки.

Завершення роботи алгоритму

Робота алгоритму закінчується тоді, коли останній блок завершить свою роботу. Для того щоб кожен блок міг завершити свою роботу, потрібні такі умови:

1. Попередній блок завершив свою роботу (не стосується першого блоку). Ця умова перевіряється за допомогою змінної лічильника у глобальній пам'яті.
2. Кожен потік у блоці має пройти по всій діагоналі і не зробити жодного перетворення матриці, тобто всі елементи над і під діагоналлю мають бути рівні нулю. Якщо за оброблення свого квадрата потік не робить ітерації, тоді він збільшує локальний лічильник пустих ітерацій. Як тільки кількість пустих ітерацій рівна розміру матриці, потік збільшує значення лічильника змінної, за допомогою атомарного додавання, яка міститься у спільній пам'яті блоку і слугує для підрахунку завершених потоків у блоці. Тільки-но кількість завершених потоків більше або дорівнює кількості потоків у блоці, всі потоки в блоці завершують свою роботу. Оскільки використовується функція __syncthreads(), яка працює як бар'єр, потоки мають завершити свою роботу одночасно.

Алгоритм SVD для тридіагональної матриці

```
1: bdm_offset = blockIdx.x * (bdm_size - 1)
2: l_r_offset = blockIdx.x * (m_size - 1)
3: copy three diagonal matrix from global memory to shared
s_bdm
4: id = –2 * threadIdx.x; top = threadIdx.x % 2 == 0; firstCycle =
true; void_cycles = 0; all_done = false;
5: repeat
6: if id > –1then
7: if id == 0 && blockIdx.x != 0) || (id == m_size - 2 &&
blockIdx.x != block_cnt - 1) then
8: update_pos = id * 3; sec_pos = id * 3 + 1;
9: if id != 0 then
10: update_pos += 3
11: end if
12: if threadIdx.x != 0 || firstCycle == false || id == 0 then
13: updated matrix in shared memory with new values from
global memory
14: end if
15: if top then
16: if abs(s_bdm[id * 3 + 2]) < eps then
17: void_cycles = void_cycles + 1
18: else
19: Use Givents rotation for s_bdm[id * 3 + 2] element
20: void_cycles = 0
21: end if
22: else
23: if abs(s_bdm[id * 3 + 1]) < eps then
24: void_cycles = void_cycles + 1
25: else
26: Use Givents rotation for s_bdm[id * 3 + 1] element
27: void_cycles = 0; sec_pos = sec_pos + 1
28: end if
29: end if
30: update global memory with new values in matrix
31: end if
32: else
33: if top then
34: if abs(s_bdm[id * 3 + 2]) < eps then
35: void_cycles = void_cycles + 1
36: else
37: Use Givents rotation for s_bdm[id * 3 + 2] element
38: void_cycles = 0
39: end if
40: else
41: if abs(s_bdm[id * 3 + 1]) < eps then
42: void_cycles = void_cycles + 1
43: else
44: Use Givents rotation for s_bdm[id * 3 + 1] element
45: void_cycles = 0; sec_pos = sec_pos + 1
46: end if
47: end if
48: if ++id == m_size – 1 then
49: id = 0; firstCycle = false
50: end if
51: if void_cycles > m_size + 1 then
52: if all threads in block are finished and (previous block is
finished or blockIdx.x == 0) then
53: all_done = true
54: end if
55: end if
56: until all_done == false
57: copy three diagonal matrix from shared memory to global
memory
```

Результати тестування

У таблицях нижче наведено результати дослідження швидкості роботи алгоритму для різних розмірів матриці та для різної точності обчислень. Всі дослідження проводили декілька разів, а отримані результати подано у вигляді середнього арифметичного.

Таблиця 1

Результати для розміру матриці 200 на 200 елементів на 100 потоках із точністю обчислень 10-8

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	1145220	2612	0.00228
[0.00, 5.00]	3294780	5047	0.00153
[0.00, 10.00]	1154460	2943	0.00255
[0.00, 50.00]	2988360	5855	0.00196
[0.00, 1000.00]	4945880	9331	0.00189

Таблиця 2

Результати для розміру матриці 500 на 500 елементів на 250 потоках із точністю обчислень 10-8

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	4004050	9391	0.00235
[0.00, 5.00]	14768500	25234	0.00171
[0.00, 10.00]	6393350	14917	0.00233
[0.00, 50.00]	8896750	19424	0.00218
[0.00, 1000.00]	49579550	72864	0.00147

Таблиця 3

Результати для розміру матриці 1000 на 1000 елементів на 500 потоках із точністю обчислень 10-8

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	23802900	38635	0.00162
[0.00, 5.00]	42754900	72422	0.00169
[0.00, 10.00]	42652600	67994	0.00159
[0.00, 50.00]	40920200	83327	0.00203
[0.00, 1000.00]	35695900	79761	0.00223

Таблиця 4

Результати для розміру матриці 200 на 200 елементів на 100 потоках із різною точністю обчислень

Кількість ітерацій	Точність обчислень	Похибка	Час виконання, 1мс	Середній час 1 ітерації, мс
783400	1.00E-02	9.00E-03	2311	0.002949962
1095800	1.00E-04	9.00E-05	3279	0.002992334
2878500	1.00E-06	8.00E-07	6990	0.002428348
13741200	1.00E-08	1.00E-08	20561	0.001496303
30175400	1.00E-10	1.00E-09	48619	0.001611213

Таблиця 5

Результати для розміру матриці 500 на 500 елементів на 250 потоках із різною точністю обчислень

Кількість ітерацій	Точність обчислень	Похибка	Час виконання, 1мс	Середній час 1 ітерації, мс
5280750	1.00E-02	9.00E-03	9897	0.001874166
12939250	1.00E-04	1.00E-04	25850	0.001997797
36864250	1.00E-06	9.00E-07	74534	0.00202185
404350250	1.00E-08	1.00E-08	530041	0.001310846
470843000	1.00E-10	2.00E-09	634310	0.001347179

Таблиця 6

Результати для розміру матриці 1000 на 1000 елементів на 500 потоках із різною точністю обчислень

Кількість ітерацій	Точність обчислень	Похибка	Час виконання, 1мс	Середній час 1 ітерації, мс
36641500	1.00E-02	1.00E-02	64966	0.001773017
97980500	1.00E-04	9.00E-05	310887	0.003172948
129814000	1.00E-06	1.00E-08	338894	0.002610612
315327500	1.00E-08	1.00E-08	572256	0.001814799
365050500	1.00E-10	2.00E-09	829576	0.002272497

Висновки

У цій роботі було розроблено та протестовано паралельний SVD алгоритм для тридіагональної матриці на відеокарті з використанням архітектури NVIDIA CUDA.

Розроблений алгоритм працює для будь-яких розмірів матриць, обмеження лише в апаратних можливостях графічного процесора (чіпу). API підтримує такі мови програмування, як C++ та Java, а також автоматично визначає оптимальну конфігурацію для алгоритму, базуючись на апаратних характеристиках відеокарти.

До переваг розробленого алгоритму можна віднести:

- високу точність обчислень;
- підтримку багатьох версій CUDA;
- оптимізовану роботу з пам’яттю та алгоритмами множення;
- API для C++ та Java, яке автоматично визначає конфігурацію для роботи та дає змогу їх змінювати;
- невелике навантаження відеокарти;
- лінійна залежність часу роботи обчислень від точності.

Список літератури

1. CUDA C++ Best Practices Guide: вебсайт. Режим доступу: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.
2. CUDA C++ Programming Guide: вебсайт. Режим доступу: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.
3. Parallel Thread Execution ISA Version 7.3: вебсайт. Режим доступу: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#operand-costs.

References

CUDA C++ Best Practices Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

CUDA C++ Programming Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.

Parallel Thread Execution ISA Version 7.3. Retrieved from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#operand-costs.

M. Semylitko, G. Malashonok

PARALLEL SVD ALGORITHM FOR A THREE-DIAGONAL MATRIX ON A VIDEO CARD USING THE NVIDIA CUDA ARCHITECTURE

SVD (Singular Value Decomposition) algorithm is used in recommendation systems, machine learning, image processing, and in various algorithms for working with matrices which can be very large and Big Data, so, given the peculiarities of this algorithm, it can be performed on a large number of computing threads that have only video cards.

CUDA is a parallel computing platform and application programming interface model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit for general purpose processing – an approach termed GPGPU (general-purpose computing on graphics processing units). The GPU provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. Other computing devices, like FPGAs, are also very energy efficient, but they offer much less programming flexibility than GPUs.

The developed modification uses the CUDA architecture, which is intended for a large number of simultaneous calculations, which allows to quickly process matrices of very large sizes. The algorithm of parallel SVD for a three-diagonal matrix based on the Givents rotation provides a high accuracy of calculations. Also the algorithm has a number of optimizations to work with memory and multiplication algorithms that can significantly reduce the computation time discarding empty iterations.

This article proposes an approach that will reduce the computation time and, consequently, resources and costs. The developed algorithm can be used with the help of a simple and convenient API in C ++ and Java, as well as will be improved by using dynamic parallelism or parallelization of multiplication operations. Also the obtained results can be used by other developers for comparison, as all conditions of the research are described in detail, and the code is in free access.

Keywords: Singular value decomposition, SVD, Nvidia CUDA, Java, C++.

Матеріал надійшов 14.06.2021