УДК 004.415.2 http://doi.org/10.15407/pp2023.01.030

С. С. Сухарський

АЛГОРИТМ СИНГУЛЯРНОГО РОЗКЛАДУ НА ГРАФІЧНОМУ ПРОЦЕСОРІ

У статті представлено реалізацію алгоритму сингулярного розкладу матриці, розроблений для виконання на графічному процесорі, який складається з двох частин: ортогонального розкладання матриці та приведення матриці до діагонального вигляду. Наведено реалізацію зведення до дводіагонального вигляду матриці з обчисленням ортогональних множників за методом Хаусхолдера і діагоналізації із використанням матриці повороту Ґівенса в середовищі jCUDA. Проведено експерименти, результати яких ретельно досліджено на предмет часу обчислень, абсолютної похибки, а також проведено порівняння з альтернативними способами реалізації сингулярного розкладу як на центральному так і на графічних процесорах.

Ключові слова: сингулярний розклад, алгоритм Хаусхолдера, алгоритм Ґівенса; обчислення на графічному процесорі, jCUDA.

Вступ

Сучасний світ дуже зав’язаний на обчислення. З розвитком Big Data, штучного інтелекту та інших популярних сьогодні сфер, потреба в швидкісних та ефективних обчисленнях стала одним із найважливіших завдань сьогодення. Саме з такою метою протягом останніх понад десяти років активно розвивається галузь обчислень на графічних процесорах, які містять тисячі ядер. Однією з передових платформ для цього є технологія розроблена NVIDIA, під назвою CUDA. Саме з цією технологією пов’язане дане дослідження, а точніше, з реалізацією SVD алгоритму з використанням потужності відеокарт, який є одним із основних методів роботи прогресивних рекомендаційних систем; вирішує багато завдань лінійної алгебри та може бути застосований для розвитку штучного інтелекту. А графічні процесори дозволяють виконувати значно більше обчислень за одиницю часу, а також змінюють підхід до побудови суперкомп’ютерів. Свідченням цього також є нещодавній прорив у сфері суперкомп’ютерів, де вперше вдалося досягти ефективності в один екзафлопс саме за рахунок поєднання центрального та чотирьох графічних процесорів у вузол, яких суперкомп’ютер Frontier налічує близько 9.5 тисяч [9].

Метою дослідження є розробка та реалізація алгоритму SVD для виконання на графічному процесорі. Попередні дослідження [2, 4] вже довели ефективність відеокарт та перевагу над центральними процесорами, а також продемонстрували проблеми поточних підходів. Також ціллю є досягнення можливості роботи з матрицями великих розмірів за рахунок ефективнішого використання наявної на графічному процесорі великої кількості ядер та для того, щоб у майбутньому можна було інтегруватися з великим комплексом бібліотек для паралельного програмування Mathpartner [7] та середовищем виконання ДАП [8], було обрано середовище JCUDA.

Загальні означення та бідіагоналізація

Нехай A – матриця над полем комплексних (або дійсних) чисел розміру $m \times n$. Її розклад у добуток трьох матриць

$$A = U\Sigma V^T, \quad (1)$$

де A – будь-яка матриця розміру $m \times n$, U – матриця $m \times m$ та ортогональна, V – матриця $n \times n$ та ортогональна і Σ – діагональна $m \times n$ матриця із впорядкованими за спаданням невід’ємними елементами на діагоналі, називається сингулярним розкладом (Singular Value Decomposition).

Перетворення Хаусхолдера (відображення Хаусхолдера) – це лінійне перетворення векторного простору, яке використовується в лінійній алгебрі для обрахунків ортогональних розкладів:

$$A = QR, A = UDV^T, \tag{2}$$

де *Q, V, U* – ортогональні (унітарні для поля комплексних чисел) матриці, *R* – права трикутна, а *D* дводіагональна матриці [5, 6].

Теорема Хаусхолдера. Для ненульових векторів $x, y \in Vx, y \in V$ з однаковою нормою ($||x|| = ||y|| ||x|| = ||y||$) існує ортогональна симетрична матриця *P,* така, що

$$y = Px. \tag{3}$$

Детальніше про бідіагоналізацію та алгоритм приведення матриці до дводіагонального виду можна прочитати в нашій попередній праці [1] на цю тему. Щодо покращень у поточній роботі було проведено оптимізацію у роботі з пам'яттю, а також додано кернел для віднімання, щоб не використовувати вбудоване рішення з додаванням та множенням на -1. Крім того, замість власних кернелів зчитування рядків на графічному процесорі та їх запис в вектори, використано стандартні методи бібліотеки Cublas – cublasDcopy використовуючи зсуви, що також дало пришвидшення.

Діагоналізація

За основу було взято ідею ітеративного опрацювання отриманої на попередньому етапі матриці для обнулення рядка над або під головною діагоналлю. Розглянемо на прикладі, як відбувається сам процес. Нехай маємо матрицю A, отриману з функції *biDiagonalize.*

$$A = \begin{pmatrix} d1 & ud1 & 0 & 0 \\ 0 & d2 & ud2 & 0 \\ 0 & 0 & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix} \tag{4}$$

Перші дві ітерації стосуватимуться першого та другого квадратів матриці. Ми говоримо одразу про дві ітерації, оскільки вони пов'язані між собою спільним елементом, а тому не можуть бути виконані одночасно.

$$A_1 = \begin{pmatrix} d1 & ud1 & 0 & 0 \\ 0 & d2 & ud2 & 0 \\ 0 & 0 & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix} \tag{5}$$

$$G = \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$L = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 2 & 3 & 1 & 2 \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix};$$

$L1 = G \cdot L;$

$out:$

$$\begin{pmatrix} (2 \cdot s + c) & (3 \cdot s + 2 \cdot c) & (s + 3 \cdot c) & (2 \cdot s + c) \\ ((-1) \cdot s + 2 \cdot c) & ((-2) \cdot s + 3 \cdot c) & ((-3) \cdot s + c) & ((-1) \cdot s + 2 \cdot c) \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix}$$

Рис 1. Обчислення матриці L.

Побудувавши матрицю повороту на A1, ми обнулимо елемент ud1, проте зіпсуємо елемент під d1. Значення d1 також зміниться, як і значення d2. Відповідно нова матриця набуде такого вигляду:

$$A_2 = \begin{pmatrix} d1' & 0 & 0 & 0 \\ x & d2' & ud2 & 0 \\ 0 & 0 & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix} \quad (6)$$

Далі ми виконуватимемо аналогічні операції і до наступних блоків (квадратів), повторюючи цей процес зверху донизу діагоналі, допоки усі значення над та під діагоналлю не будуть обнулені. Упродовж цього процесу ми також обчислюємо матриці L та R, які на початку є одиничними, а далі змінюються від застосувань до них матриці повороту Гівенса. На проміжному етапі алгоритму це обчислення виглядає так:

Як бачимо, зоною впливу в цьому випадку є відповідно перші два рядки матриці L. Для матриці ж R ситуація відрізнятиметься тим, що там під впливом множення будуть змінюватись стовпчики:

$$G = \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$R = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 2 & 3 & 1 & 2 \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix};$$

$R1 = R \cdot G;$

$out:$

$$\begin{pmatrix} ((-2)\cdot s + c) & (s + 2\cdot c) & 3 & 1 \\ ((-3)\cdot s + 2\cdot c) & (2\cdot s + 3\cdot c) & 1 & 2 \\ ((-1)\cdot s + 3\cdot c) & (3\cdot s + c) & 2 & 3 \\ ((-2)\cdot s + c) & (s + 2\cdot c) & 3 & 1 \end{pmatrix}$$

Рис 2. Обчислення матриці R.

Наведені приклади були зроблені в системі MathPartner [7]. У процесі виконання алгоритму, матриця G також буде змінюватися, відповідно до поточних елементів.

Паралелізм та роботу з пам'яттю додано за схемою, яка була запропонована в роботі [2], де паралельно обробляються N / 2 блоків, а N - це розмір квадратної матриці. Приклад блоків, які можна опрацьовувати паралельно:

$$B = \begin{pmatrix} d1 & ud1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & d2 & ud2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d3 & ud3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d4 & ud4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d5 & ud5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d6 & ud6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d7 & ud7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d8 \end{pmatrix} \quad (7)$$

Відповідно ми використовуємо N / 2 головних потоків. Спочатку роботу починає перший потік, а далі з кроком у два блоки свою роботу починає наступний і так повторюємо поки є вільні потоки. Щодо обчислень матриць L та R, то за рахунок того, що потоки працюють із різницею в два кроки, нема потреби синхронізувати значення цих матриць між різними потоками, а тому є можливість так само паралельно обчислювати їх у кожному з них.

Проте в [2] було помітно, що пришвидшення незначне, оскільки один потік хоч і працював з невеликим блоком головної матриці, але має опрацьовувати велику частину матриці L або R. Відповідно страждала швидкодія. Тому, було розширено цю ідею, ввівши поняття допоміжних потоків. Ідея полягає в тому, що кожному з N / 2 головних потоків дається ще X допоміжних потоків порівну так, щоб загальна кількість потоків уміщалася в межах одного блоку. Наприклад, якщо ми працюємо з матрицею 128 x 128 та використовуємо 64 головних потоки, а максимальна кількість потоків у межах блоку 1024, тоді використовується не просто 64 потоки, а 64 групи по 16 потоків. Цей підхід дозволяє також розділити між ними роботу, необхідну для виконання обчислень із матрицями L та R. Водночас, додано перевірки, які дії варто виконувати тільки головному потоку, а які залишити для всіх. Наприклад, застосування матриці повороту до матриці A виконує лише головний потік, а інші на це час не витрачають. Такий підхід продемонстрував значне пришвидшення. Проте, велика кількість потоків також викликає певні затримки з часом завершення програми (по-

трібно чекати, поки усі потоки завершать роботу), а також постає питання оптимального вибору кількості потоків та блоків, на яких має запускатися обчислення.

Експерименти

Експерименти виконано для кожної частини окремо та для всієї програми, де ці частини працюють одна за одною, причому перша зберігає результат на відеокарті, а наступна одразу його опрацьовує. Ці дані досліджено щодо залежності часу обчислень від розміру матриць, а також оцінено та досліджено точність обчислень. Крім цього, досліджувалась залежність часу від точності для ітераційної частини алгоритму. Також проведено порівняння з послідовним алгоритмом на центральному процесорі.

Усі обчислення проводились на графічному процесорі GIGABYTE RTX 3070 GAMING OC з характеристиками, наведеними в таблиці 1.

Таблиця 1

Характеристики графічного процесора

Графічний процесор	GA104-300-A1
Архітектура	Ampere
Пам'ять	8 GB типу GDDR6; 256 bit; 448.0 GB/s
Кількість CUDA ядер	5888
Спільна пам'ять блоку	48 KB
Максимальна кількість потоків	1024

Інші характеристики системи наведено в таблиці 2

Таблиця 2

Загальні характеристики системи

Центральний процесор	Intel Core i5-12600K 3.7 GHz
Оперативна пам'ять	G.Skill 16 GB (2x8 GB) DDR4 3600 MHz
Накопичувач	WD SN850 SSD 1 TB

Обчислення виконувалися на операційній системі Windows 11 Pro 21H2 x64 (build 22000.739).

Насамперед було проведено експерименти для оцінки часу виконання алгоритму та оцінки похибки обчислень. Для цього генерувались матриці різних розмірів з діапазоном значень від 0 до 100 типу *Double*. Для кожної згенерованої матриці здійснювалося по два запуски, оцінювався час виконання кожної частини алгоритму та загальний час, а також брався середній час і діапазон відхилень. У цей час враховується весь процес обчислень, та не враховується процес перевірки похибки обчислень. Такі ж метрики проводились для похибки обчислень. У таблиці 3 наведено детальні дані щодо часу обчислень. Час наведено в секундах.

Щоб наочніше оцінити залежність часу виконання від розміру матриці, розглянемо декілька графіків. На рисунку 3 ми спостерігаємо плавне зростання часу зі збільшенням розміру матриці. Також можемо зазначити, що процес бідіагоналізації відбувається досить швидко – квадратна матриця розміру 1024 обчислюється менше секунди. Щодо відхилення у часі прогонів, то для бідіагоналізації воно зовсім мале, а для діагоналізації незначне.

Таблиця 3

Оцінка часу виконання

Розмір матриці	128	256	512	1024	2048
Бідіагоналізація	0.060	0.103	0.216	0.873	6.557
Відхилення (+/- t)	0.005	0.009	0.022	0.009	0.008
Діагоналізація	0.131	0.338	1.280	11.886	64.122
Відхилення (+/- t)	0.016	0.022	0.076	0.802	2.677
Алгоритм	0.191	0.441	1.496	12.759	70.679

Рис 3.Час бідіагоналізації.

Розглянемо тепер графік залежності часу діагоналізації від розміру матриці. Як бачимо з таблиці 3, цей процес складає переважну частину усього алгоритму, тому для порівняння побудуємо графік одразу з двома кривими.

Рис 4. Час діагоналізації та всього алгоритму

Рисунок 4 дуже добре демонструє, наскільки близькі за часом ці криві – друга повністю перекриває першу. Сама залежність часу дещо різкіша, ніж для бідіагоналізації, проте загальна тенденція схожа. А от різниця в часі виконання велика – приблизно в 6 разів для матриць 512 x 512 та більше ніж в 12 разів на розмірі 1024 x 1024. Що цікаво, зі збільшенням розміру з 512 до 1024 час обчислень зріс у 8.5 разів. Це погано, але зі збільшенням з 1024 до 2048 час збільшився в 5.53 разів. Це покращення досягнуто за рахунок того, що матриця розміру 2048 виконується на більшій кількості блоків, відповідно більше потоків працюють з частиною матриці, але втрачається час на синхронізації значень.

Щодо похибки обчислень, то, аналізуючи детальні дані наведені в таблиці 4, помітно, що абсолютна похибка обчислень досить повільно змінюється залежно від розміру матриці. Але на більших розмірах матриці відбувається певний стрибок у похибці діагоналізації, коли ми використовуємо багато блоків. Особливо повільні зміни відбуваються під час бідіагоналізації. Проте, варто зазначити, що діагоналізацію можна налаштувати так, щоб алгоритм працював, поки не буде досягнуто бажаної точності, або не настане момент, коли краще вже не можна порахувати.

Ми здійснили дослідження для квадратних матриць розміру 128 та 256, заповнених числами типу *Double* в діапазоні від 0 до 100, як і для програми, що виконувалась на графічному процесорі, і порівняли результати, щоб розуміти ефективність таких обчислень на GPU. В таблиці 5 наведено результати експериментів, а саме, час виконання (в секундах) всього алгоритму SVD на GPU, CPU та визначене пришвидшення.

Таблиця 4

Оцінка абсолютної похибки обчислень

Розмір матриці	128	256	512	1024	2048
Бідіагоналізація	4.47E-13	3.69E-13	5.33E-13	1.28E-12	1.78E-12
Відхилення (+/- err)	1.065E-13	4.25E-14	4.28E-14	1.17E-14	1.03E-13
Діагоналізація	5.28E-11	4.51E-11	1.73E-5	5.44E-5	5.45E-6
Відхилення (+/- err)	7.22E-13	6.16E-12	8.35E-6	7.56E-6	3.2E-6
Алгоритм	7.007E-12	8.64E-12	2.63E-6	2.18E-4	1.06E-6

Таблиця 5

Розмір матриці	128	256
CPU	110.291	1802.114
GPU	0.191	0.441
Пришвидчення в n разів	577,43	4086,42

Порівняння часу обчислень на CPU та GPU

Наведена вище таблиця підтверджує, що один графічний процесор має дуже великі потужності для паралельних та інтенсивних обчислень. Навіть на невеликій матриці розміру 128 нам знадобиться дуже добре реалізована програма і потужний процесор з великою кількістю ядер, щоб зрівнятись із можливостями відеокарти. На матриці більшого розміру різниця взагалі колосальна.

Також проведено порівняння з CUDA API [10]. Були виконані експерименти з точністю 10E-10 та матрицями з числами типу *Double* з діапазоном значень від 0 до 100. Як бачимо з даних у таблиці 6, різниця поки дуже серйозна. Якщо на матрицях невеликого розміру різниця ще відносно невелика і її легко подолати за рахунок оптимізації виділення блоків та потоків, то на матрицях великого розміру стає дуже помітною проблема, коли із використанням декількох блоків потрібно навчитися використовувати групи допоміжних (додаткових) потоків до тих, що працюють над множеннями правої та лівої матриць.

Висновки

У даній праці досліджено та реалізовано алгоритм сингулярного розкладу матриць на графічному процесорі. Перша частина алгоритму використовує метод Хаусхолдера для приведення матриці до дводіагонального вигляду та показує чудові результати із швидкості й точності обчислень. Підхід трансформацій покращено та оптимізовано під виконання на відеокарті для зменшення кількості запису та зчитувань з пристрою й відповідно максимальної ефективності обчислень.

У другій частині алгоритму застосовано новий підхід у реалізації стандартного ітераційного процесу, який вдало використовує присутні в графічному процесорі ресурси, адаптуючись під конкретні параметри конкретного пристрою. Також запропоновано підхід до пришвидшення та оптимізації обчислень лівої та правої матриць.

Здійснено ретельне дослідження з детальним описом умов та способів оцінки часу виконання та визначення точності обчислень. Результати показали колосальне пришвидшення відносно виконання цього алгоритму на центральному процесорі. Також вони підтвердили, що ідея виділення групи потоків у кожному блоці є дуже перспективною, адже швидкість обчислень помітно зростає із застосуванням оптимальних параметрів. Але, як показали порівняння з альтернативними способами реалізації, цей підхід з групами потоків потрібно поліпшувати для декількох блоків, оскільки саме на великих матрицях з'являється помітне відставання за часом обчислень порівняно з CUDA API, а також падає точність обчислень. Тому далі потрібно знайти вирішення проблеми ефективної паралельної роботи багатьох блоків, а також окремо досліджувати оптимальний підбір їхнього

Таблиця 6

Порівняння з CUDA API

Розмір матриці	128	256	512	1024	2048
SVD	0.191	0.441	1.496	12.759	70.679
Похибка	7.007E-12	8.64E-12	2.63E-6	2.18E-4	1.06E-6
CUDA API SVD	0.038	0.113	0. 335	1.224	7.462
Похибка	1.36E-11	2.04E-11	2.02E-11	2.07E-11	2.72E-11
Пришвидшення в n разів	5.02	3.9	4.46	10.42	9.47

розміру та кількості.

Отже, запропонований підхід показав чудові результати, але ще залишає за собою простір для покращень та оптимізації в майбутніх дослідженнях.

Література

1. Малашонок Г.І., Сухарський С.С. Алгоритм обчислення дводіагональної матриці ортогональним розкладанням на графічному процесорі. Наукові записки НаУКМА. Комп'ютерні науки. 2021. Том 4. [Електронний ресурс]. – режим доступу: http://nrpcomp.ukma.edu.ua/article/view/246581. Дата звернення: 2022.06.14
2. Малашонок Г.І., Семилітко М.Ю. Паралельний SVD алгоритм для тридіагональної матриці на відеокарті з використанням архітектури Nvidia CUDA. Наукові записки НаУКМА. Комп'ютерні науки. 2021. Том 4. [Електронний ресурс]. – режим доступу: http://nrpcomp.ukma.edu.ua/article/view/246582. Дата звернення: 2022.06.14
3. Малашонок Г. І., Савченко С. О., "Матричні алгоритми розбиття множин для рекомендаційних систем". НаУКМА, 2019.
4. S. Lahabar and P. J. Narayanan, "Singular value decomposition on GPU using CUDA," 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-10, doi: 10.1109/IPDPS.2009.5161058. [Електронний ресурс]. – режим доступу: https://ieeexplore.ieee.org/document/5161058/citations?tabFilter=papers#citations. Дата звертання: 2022.06.12
5. Persson, "Householder Reflectors and Givens Rotations", MIT 18.335J / 6.337J *Introduction to Numerical Methods.* [Електронний ресурс]. – режим доступу: https://math.dartmouth.edu/~m116w17/Householder.pdf. Дата звертання: 2022.06.05
6. Cornell University, "Numerical linear algebra and matrix factorizations", [Електронний ресурс]. – режим доступу: http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/Householder.html. Дата звертання: 2022.06.12
7. Система комп'ютерної алгебри MathPartner [Електронний ресурс]. - http://mathpar.ukma.edu.ua/. Дата звернення: 2022.06.13
8. Malaschonok, G.I., Sidko, A.A. Supercomputer Environment for Recursive Matrix Algorithms. Program Comput Soft 48, 90–101 (2022). https://doi.org/10.1134/S0361768822020086
9. Новинний ресурс Nauka.ua [Електронний ресурс]. – режим доступу: https://nauka.ua/news/superkompyuter-vpershe-dosyag-efektivnosti-v-odin-ekzaflops. Дата звернення: 2022.06.14
10. CUDA API SVD. [Електронний ресурс]. – режим доступу: https://github.com/NVIDIA/CUDALibrarySamples/blob/master/cuSOLVER/gesvdj/cusolver_gesvdj_example.cu . Дата звернення: 2022.07.01

References

1. Malashonok H. I., Sukharskyi S. S. A GPU-based orthogonal matrix factorization algorithm that produces a two-diagonal shape. NaUKMA. Computer Science. 2021. retrieved from http://nrpcomp.ukma.edu.ua/article/view/246581.
2. Malashonok H. I., Semylitko M.Y. Parallel SVD algorithm for a three-diagonal matrix on a video card using the Nvidia CUDA architecture. NaUKMA. Computer Science. 2021. retrieved from http://nrpcomp.ukma.edu.ua/article/view/246582.
3. Malashonok H. I., Savchenko S. O., "Matrychni alhorytmy rozbyttia mnozhyn dlia rekomendatsiinykh system". NaUKMA, 2019.
4. S. Lahabar and P. J. Narayanan, «Singular value decomposition on GPU using CUDA,» 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-10, retrieved from http://doi.org/10.1109/IPDPS.2009.5161058.
5. Persson, "Householder Reflectors and Givens Rotations", MIT 18.335J / 6.337J Introduction to Numerical Methods. retrieved from https://math.dartmouth.edu/~m116w17/Householder.pdf.
6. Cornell University, "Numerical linear algebra and matrix factorizations", retrieved from http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/Householder.html.
7. Computer Algebra System MathPartner retrieved from http://mathpar.ukma.edu.ua/.
8. Malaschonok, G.I., Sidko, A.A. Supercomputer Environment for Recursive Matrix Algorithms. Program Comput Soft

48, 90–101 (2022). https://doi.org/10.1134/S0361768822020086
9. News resource Nauka.ua retrieved from https://nauka.ua/news/superkompyuter-vpershe-dosyag-efektivnosti-v-odin-ekza-flops.
10. CUDA API SVD. retrieved from https://github.com/NVIDIA/CUDALibrarySamples/blob/master/cuSOLVER/gesvdj/cusolver_gesvdj_example.cu .

Одержано: 14.12.22

Про авторів:
Сухарський Сергій Сергійович,
Магістр, аспірант
Кількість публікацій в українських виданнях: 1
Кількість публікацій в зарубіжних індексованих виданнях: 0
ORCID: 0000-0002-5873-984X

Місце роботи:
Інститут Програмних Систем
НАН України
Адреса: м. Тернопіль,
вул. Збаразька 37