УДК 512.626.824:512.64 **DOI:** 10.31673/2412-9070.2026.055514

Н. М. АУШЕВА, д-р. техн. наук, професор;
ORCID: 0000-0003-0816-2971
І. А. ВЕРЕМІЙЧУК, аспірант,
ORCID: 0009-0009-2279-7659
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

ЕФЕКТИВНІСТЬ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ З ВИКОРИСТАННЯМ КВАТЕРНІОНІВ

У статті проведено системне дослідження та порівняльний аналіз ефективності паралельних обчислень при оперуванні некомутативними кватерніонними структурами даних і розв'язанні кватерніонних лінійних систем $Ax = b$ на гетерогенних апаратних платформах (CPU, GPU, TPU) з використанням сучасних програмних стеків (NumPy, TensorFlow Graphics, tfquaternion, CuPy). Розглянуто теоретичні аспекти використання алгебри Гамільтона для обробки багатовимірних і багатоканальних сигналів (колірних просторів, 3D-орієнтації), що забезпечує скорочення кількості незалежних навчальних параметрів нейромережевих моделей на 75%. Проаналізовано накладні витрати обчислювальних графів, зумовлені відсутністю нативної апаратної підтримки некомутативного множення та необхідністю симуляції добутку Гамільтона через 16 дійсних операцій і дійсне блочне представлення (Real Matrix Representation) розмірністю $4M \times 4N$. За допомогою розробленого програмного інструментарію бенчмаркінгу в хмарній інфраструктурі Google Colab оцінено латентність виконання добутку Гамільтона для вибірок обсягом $N \in [10^2 \dots 10^6]$ та швидкість LU-декомпозиції лінійних систем. Досліджено ефект застосування XLA-компіляції (kernel fusion) і FP16-обчислень при оптимізації графів QCNN на прикладі класифікації зображень з датасетів CIFAR-10 та HAM10000. Сформовано практичні рекомендації щодо вибору програмно-апаратного стеку залежно від розмірності даних і вимог до обчислювальної латентності.

Ключові слова: кватерніони, добуток Гамільтона, паралельні обчислення, QCNN, TensorFlow Graphics, CuPy, XLA-компіляція, GPU, TPU, дійсне матричне представлення.

Вступ

Сучасний етап розвитку високопродуктивних обчислень (High-Performance Computing, HPC) та інтелектуального аналізу даних характеризується експоненційним зростанням обсягів багатовимірної інформації, що потребує обробки в режимі реального часу. Традиційні обчислювальні архітектури та алгоритми машинного навчання переважно оперують у дійсному евклідовому просторі (R). Проте, при моделюванні складних просторово-часових процесів, обробці колірних сигналів, афінних перетворень у тривимірному просторі або аналізі фазово-модульованих сигналів класичний скалярний підхід демонструє суттєві обмеження. Намагання опрацьовувати багатокомпонентні дані як набір ізольованих дійсних каналів призводить до втрати внутрішніх крос-канальних кореляцій, штучного збільшення кількості параметрів моделей та виникнення математичних сингулярностей.

Математичний апарат гіперкомплексної алгебри, зокрема простір кватерніонів Гамільтона (H), пропонує елегантну й математично обґрунтовану альтернативу дійсним численням [1]. Кватерніон, що оперує однією дійсною та трьома уявними одиницями, дозволяє кодувати чотиривимірні або триканальні вектори (наприклад, колірний простір RGB або просторові координати) як єдину алгебраїчну сутність. Головна теоретична перевага гіперкомплексних моделей полягає в заміні стандартного скалярного множення операцією добутку Га-

© Аушева Н. М., Веремійчук І. А., 2026

мільтона. Ця операція діє як природний структурний регуляризатор, що вимушує обчислювальну систему враховувати взаємозв'язки між усіма компонентами сигналу одночасно. Щонайменше, у задачах нейромережевого моделювання це забезпечує скорочення кількості незалежних навчальних ваг на 75% порівняно з аналогічними за конфігурацією дійсними архітектурами [2], [3].

Попри значний теоретичний потенціал та параметричну ефективність кватерніонних моделей, їх ширше впровадження у практику інженерії програмного забезпечення стримується проблемою обчислювальної латентності (latency). Жоден із сучасних комерційних процесорів або графічних прискорювачів не має нативної апаратної підтримки некомутативної алгебри кватерніонів. Будь-яка маніпуляція з кватерніонами на практиці симулюється через послідовність операцій над дійсними числами. Це створює суттєві накладні витрати пам'яті (computational overhead) та збільшує кількість вузлів в обчислювальних графах розпізнавання.

З огляду на це, критично важливого значення набуває проблема дослідження ефективності паралельних обчислень при оперуванні кватерніонними структурами. Актуальність теми зумовлена необхідністю пошуку оптимального балансу між математичною щільністю представлення даних у просторі H та реальними часовими витратами на виконання цих операцій на сучасному гетерогенному обладнанні.

Наукова спільнота наразі пропонує кілька альтернативних підходів до формування кватерніонних структур даних та організації паралельного виконання операцій. Класичний шлях базується на використанні бібліотеки *NumPy* та векторизованих масивів ndarray на базі центрального процесора (CPU), що спираються на оптимізовані обчислювальні блоки лінійної алгебри BLAS/LAPACK. Водночас, парадигма глибокого навчання вимагає перенесення обчислень на графічні (GPU) та тензорні (TPU) прискорювачі, що реалізується переважно через тензорні структури *TensorFlow* або *PyTorch*. При цьому розробники змушені обирати між низькорівневим ручним моделюванням Гамільтонових обчислень та використанням спеціалізованих пакетів, таких як *TensorFlow Graphics* (tfg.geometry), *tfquaternion* або *PyTorch3D* [4]. Кожен із цих варіантів має власні ліміти масштабованості, можливості паралелізму та специфіку роботи з пам'яттю.

Особливу математичну та алгоритмічну складність становлять задачі розв'язання лінійних матричних систем із кватерніонами виду $\mathrm{Ax} = \mathrm{b}$. Через некомутативність множення в H, пряме перенесення класичних методів декомпозиції (LU, QR або Гаусса) у первинному вигляді є неможливим. Основним індустріальним методом є перехід до дійсного матричного представлення (Real Matrix Representation), що трансформує систему розмірністю $M \times N$ у дійсну блочну матрицю $4M \times 4N$ [2]. Таке розширення розмірності створює специфічні виклики для паралельних обчислень: з одного боку, матриця стає розрідженою та структурованою, що відкриває шлях до блочного паралелізму, з іншого — різко зростає навантаження на шину передачі даних при розподілі обчислень між ядрами прискорювачів.

Постановка проблеми

У сфері сучасних високопродуктивних обчислень та інтелектуального аналізу даних спостерігається стрімка зміна парадигм обробки багатовимірних сигналів. Традиційні архітектури систем штучного інтелекту та комп'ютерного зору тривалий час спиралися на математичний апарат дійсних чисел (R). У межах цієї класичної моделі будь-який багатоканальний сигнал (наприклад, триканальні колірні простори RGB/YUV, просторово-часові координати кінематичних систем чи фазово-інтенсивні виміри радіолокаційних датчиків) піддається примусовій декомпозиції на набір ізольованих скалярних поверхонь.

Такий підхід створює глибокий архітектурно-семантичний розрив між сутністю фізичного сигналу та математичним методом його обробки. Коли стандартна згорткова мережа (CNN) або багатовимірний лінійний регресор оперують дійсними числами, вони обробляють кожен канал окремим, незалежним набором фільтрів. Взаємозв'язок між хроматичними або просторовими компонентами вимушено моделюється лише опосередковано — через лінійне підсумовування на пізніх етапах формування карт ознак. Як наслідок, обчислювальні системи

демонструють критичну вразливість до спектральних шумів, нелінійних змін колірного балансу та спотворень умов освітленості. Навіть незначний зсув значень в одному з каналів призводить до десинхронізації дійсних фільтрів, руйнування проміжних представлень ознак і хибної класифікації або регресії об'єктів.

Математичний апарат некомутативної алгебри кватерніонів Гамільтона (H) дозволяє подолати це обмеження, об'єднуючи багатоканальні виміри в цілісну векторну сутність:

$$\mathrm{q} = q_0 + q_{1i} + q_{2j} + q_{3k} \quad (1)$$

Завдяки структурі добутку Гамільтона ($\otimes$), кватерніонні нейронні мережі (QNN) та сол-вери оперують взаємопов'язаними матрицями ваг. Це дозволяє фіксувати просторово-хроматичні ознаки синергетично на кожному шарі обробки, забезпечуючи скорочення кількості незалежних параметрів навчання на 75%. Зниження кількості ступенів свободи діє як потужний внутрішній регуляризатор, що радикально мінімізує ризик перенавчання (overfitting) на обмежених вибірках.

Проте, перехід до гіперкомплексного простору H створює нову, обчислювально складну фундаментальну проблему. Жодна сучасна апаратна архітектура (ні CPU з інструкціями SIMD, ні тензорні ядра GPU від NVIDIA, ні спеціалізовані процесори Google TPU) не має нативної підтримки кватерніонних типів даних та некомутативного множення. На апаратному рівні будь-яка гіперкомплексна маніпуляція або операція згортки повинна бути розбита та симульована через послідовність операцій над дійсними числами.

Для реалізації добутку Гамільтона $\mathrm{W} \otimes \mathrm{X}$ у сучасних обчислювальних середовищах фреймворків глибокого навчання (TensorFlow, PyTorch) система змушена виконувати явну декомпозицію та конструювати блочні матричні вирази:

$$\begin{bmatrix} (W \otimes X)_0 \\ (W \otimes X)_1 \\ (W \otimes X)_2 \\ (W \otimes X)_3 \end{bmatrix} = \begin{bmatrix} W_0 & -W_1 & -W_2 & -W_3 \\ W_0 & W_0 & -W_3 & W_2 \\ W_0 & W_3 & W_0 & -W_1 \\ W_0 & -W_2 & W_1 & W_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} \quad (2)$$

Така симуляція некомутативної математики через лінійні комбінації дійсних тензорів створює колосальні накладні витрати обчислювальних ресурсів (computational overhead). Замість однієї нативної операції множення, графічний чи центральний процесор змушений виконувати 16 окремих дійсних множень та серію операцій додавання/віднімання. Більше того, для розподілу цих компонентів у пам'яті фреймворки глибокого навчання генерують тисячі допоміжних вузлів розбиття (tf.split, torch.unbind) та об'єднання (tf.concat, torch.stack) тензорів. Це призводить до критичного перевантаження шини передачі даних, фрагментації оперативної пам'яті (VRAM) та штучного уповільнення часу виконання однієї епохи навчання на GPU прискорювачах на 15–20% порівняно з класичними CNN.

Ще більшу проблему становить завдання розв'язання лінійних матричних систем із кватерніонами вигляду $\mathrm{Ax} = \mathrm{b}$, які є базисом для задач багатовимірної гіперкомплексної регресії. Перехід до еквівалентного дійсного представлення розширює розмірність матриці коефіцієнтів з $\mathrm{M} \times \mathrm{N}$ до $4M \times 4N$. Отримана блочна матриця стає сильно структурованою, але водночас розрідженою.

При спробі розпаралелювання таких систем на сучасних гетерогенних платформах виникає ефект обчислювальної десинхронізації. Центральні процесори (CPU) добре справляються із послідовними блочними структурами завдяки розвиненим механізмам кешування, але обмежені кількістю фізичних ядер. Графічні процесори (GPU) мають тисячі ядер CUDA, проте їхня SIMT (Single Instruction, Multiple Threads) архітектура страждає від розгалужень та частих звернень до глобальної пам'яті під час складання/розбиття розріджених кватерніонних блоків.

Наявні на сьогодні програмні рішення пропонують кардинально різні підходи до розв'язання цього протиріччя, створюючи проблему вибору оптимального стеку:

- Векторизовані масиви NumPy ndarrays забезпечують стабільне виконання на CPU за рахунок низькорівневих бібліотек лінійної алгебри BLAS/LAPACK, але повністю позбавлені можливості автоматичного диференціювання та масштабування на прискорювачі.

- Пакет TensorFlow Graphics (tfg.geometry) надає інструменти побудови статичних обчислювальних графів із підтримкою XLA (Accelerated Linear Algebra) компіляції. Це теоретично дозволяє TPU-прискорювачам зливати допоміжні вузли графа (kernel fusion) в єдині апаратні інструкції, але вимагає жорсткої фіксації розмірностей батчів та створює значний overhead на етапі попередньої компіляції.
- Спеціалізовані бібліотеки в екосистемі PyTorch (наприклад, PyTorch3D) фокусуються на написанні нативних CUDA C++ розширень, які повністю оминають інструменти Python. Вони оптимізують обчислення на GPU, проте їх перенесення на архітектуру Google TPU залишається вкрай складним технологічним процесом через нестабільність прошарку torch_xla.

Таким чином, постає гостра науково-практична проблема, яка полягає у відсутності системних досліджень та чітких кількісних критеріїв оцінки ефективності паралельних обчислень при оперуванні кватерніонами на гетерогенному обладнанні. Розробники сучасних інтелектуальних систем змушені проектувати гіперкомплексні моделі емпірично, без розуміння того, як вибір конкретної структури даних (NumPy vs TensorFlow vs PyTorch) та апаратного прискорювача (CPU vs GPU vs TPU) вплине на латентність обчислень, коефіцієнт розпаралелювання та швидкість збіжності алгоритмів оптимізації.

Вирішення цієї проблеми потребує проведення комплексного бенчмаркінгу, формалізації накладних витрат обчислювальних графів та створення єдиної методології побудови високоефективних паралельних кватерніонних систем.

Аналіз останніх досліджень і публікацій

Розвиток методів високопродуктивних обчислень та інтелектуального аналізу даних протягом останнього десятиліття демонструє чітку тенденцію до переходу від мономодальних, скалярних систем до складних гіперкомплексних архітектур. Проблема оптимізації обробки багатоканальних сигналів та просторових трансформацій стимулювала масштабний пласт наукових розвідок, фокусом яких став пошук альтернативних математичних просторів, здатних подолати обмеження класичної дійсних обчислень (***R***). У межах цього вектору дослідження некомутативної алгебри кватерніонів Гамільтона (***H***) пройшли шлях від фундаментальних теоретичних вишукувань до створення прикладних програмно-апаратних стеків паралельних обчислень.

Обмеження дійсних моделей лінійної алгебри та передумови гіперкомплексних обчислень

Фундамент класичного комп'ютерного зору та цифрової обробки сигналів тривалий час спирався на глибокі згорткові мережі (CNN) та багатовимірні регресійні моделі дійсного типу [5], [6]. Проте, як зазначено в роботі [7], декомпозиція векторних за своєю фізичною природою сигналів (таких як колірні простори RGB або фазово-інтенсивні виміри) на ізольовані скалярні канали є математично субоптимальною. У дійсних архітектурах крос-канальні просторово-хроматичні кореляції ігноруються на початкових етапах обробки, що змушує систему нарощувати глибину мережі та кількість фільтрів для примусового вивчення цих зв'язків.

Дослідження [8] підтвердили, що лінійні комбінації ізольованих дійсних каналів створюють обчислювальну надлишковість та роблять моделі вразливими до спектральних шумів. При нелінійному зміщенні яскравості або порушенні балансу білого дійсні ваги втрачають здатність до узагальнення, оскільки їхні математичні матриці не зафіксовані єдиним геометричним інваріантом. Це заклало передумови для інтеграції простору кватерніонів Гамільтона безпосередньо в обчислювальні конвеєри (pipelines) паралельних систем.

Становлення кватерніонних нейромережевих архітектур та їх параметрична ефективність

Ідея заміни дійсних ваг на гіперкомплексні числа призвела до створення класу кватерніонних нейронних мереж (QNN). Початкові роботи у цьому напрямі [2],[3],[6] були зосереджені на теоретичному обґрунтуванні та доведенні параметричної ефективності простору H. Завдяки

математичним властивостям добутку Гамільтона, згорткові та повнозв'язні кватерніонні шари оперують взаємопов'язаними дійсними компонентами ваг. Це дозволило розробникам скоротити кількість навчальних параметрів моделі рівно на 75% при збереженні ідентичної архітектурної ємності.

В дослідженнях [9] гіперкомплексного зворотного поширення помилки (Quaternion Backpropagation) було доведено, що таке обмеження ступенів свободи діє як потужний внутрішній регуляризатор обчислень. Мережі QNN продемонстрували здатність досягати плато точності за значно меншу кількість епох навчання порівняно з дійсними аналогами на вибірках з обмеженим обсягом даних (наприклад, CIFAR-10), виявляючи високу стійкість до ефекту перенавчання (overfitting).

Проте, як зазначено в огляді [10], ці переваги є виключно теоретичними до моменту їх трансляції на фізичний рівень обчислювальних систем. Основний науковий виклик перемістився у площину дослідження латентності обчислень, оскільки відсутність нативної апаратної підтримки некомутативної алгебри кватерніонів на сучасних CPU та GPU прискорювачах нівелювала параметричний виграш через великі накладні витрати на симуляцію добутку Гамільтона.

Проблема симуляції некомутативних операцій та дійсного матричного представлення

Для вирішення задач багатовимірної кватерніонної регресії та розв'язання лінійних матричних систем виду $\boldsymbol{Ax = b}$ дослідники зіткнулися з проблемою некомутативності множення уявних одиниць Гамільтона ($ij = k, ji = -k$) [1]. Класичні прямі методи декомпозиції матриць (LU, QR, Гаусса) не здатні оперувати безпосередньо в гіперкомплексному просторі без ризику руйнування фазової структури.

Для подолання цього бар'єра у сучасних обчислювальних системах було впроваджено метод дійсного матричного представлення (Real Matrix Representation) [4]. Кожен кватерніонний елемент замінюється структурованою дійсного блочною матрицею розмірності 4×4. Це дозволило трансформувати кватерніонну лінійну систему розмірністю $M \times N$ в еквівалентну дійсну систему розмірністю $4M \times 4N$, відкриваючи можливість використання високоефективних промислових CPU-солверів (таких як LAPACK або Intel MKL).

Проте, аналіз ефективності такого підходу, проведений у роботі [4], виявив нову проблему: блочне розширення різко збільшує розмірність матриць, роблячи їх розрідженими. При спробі масового паралельного обчислення таких систем на графічних процесорах (GPU) виникає критичне перевантаження шини передачі даних, оскільки архітектура SIMT (Single Instruction, Multiple Threads) змушена витрачати ресурси на постійне розгалуження потоків та фрагментарний доступ до глобальної пам'яті (VRAM) для збирання й розбирання кватерніонних блоків.

Високопродуктивні матричні обчислення на Python

Сучасна екосистема мови Python стала стандартом де-факто для задач обчислювальної математики, аналізу даних та машинного навчання. Попри історичний статус Python як інтерпретованої та відносно повільної мови, високопродуктивні матричні обчислення реалізуються завдяки використанню низькорівневих обчислювальних рушіїв, векторизації та апаратного прискорення. Перехід від традиційних скалярних обчислень до паралельної обробки багатовимірних тензорів спирається на ключові концептуальні підходи: базову векторизацію на CPU (NumPy), пряме апаратне прискорення на GPU (CuPy), а також поєднання функціонального програмування та оптимізації графів через комбінацію JAX і доменного компілятора XLA [11], [12], [13], [14], [15],[16].

Підходи до організації матричних операцій та розв'язання лінійних систем

Для оцінки ефективності розв'язання фундаментальних задач лінійної алгебри — зокрема, матричного множення ($C = A \cdot B$) та розв'язання систем лінійних рівнянь ($A x = b$) — клю-

чове значення має спосіб трансляції високорівневих інструкцій у машинні команди. Основні варіанти реалізації цього способу трансляції:

- Класичний модуль NumPy, який спирається на оптимізовані C/Fortran бібліотеки лінійної алгебри (BLAS/LAPACK, такі як OpenBLAS або Intel MKL). Для матричних операцій середньої розмірності NumPy забезпечує високу швидкість завдяки передкомпактному збереженню даних у пам'яті (C-contiguous) та векторизованим інструкціям. Проте при виконанні складних послідовностей матричних перетворень NumPy створює проміжні масиви у системній RAM, що викликає значний оверхед оперативної пам'яті.

- Модуль CuPy, який забезпечує практично 100% сумісність із API NumPy, але переносить збереження даних та виконання операцій у відеопам'ять (VRAM) графічного прискорювача. Для матричного множення та розв'язання лінійних систем CuPy делегує обчислення викликам високооптимізованих бібліотек NVIDIA cuBLAS та cuSOLVER. Завдяки тисячам паралельних CUDA-ядер, CuPy демонструє прискорення у 10–50 разів на матрицях великої розмірності ($N \geq 10^3$), хоча на малих матрицях програє NumPy через накладні витрати на ініціалізацію CUDA-контексту та передачу даних через PCIe.

- Бібліотека JAX від Google, яка поєднує NumPy-подібний API із системою автоматичного диференціювання (Autograd) та можливістю комбінувати трансформації функцій (jit, vmap, pmap). JAX перетворює код Python у проміжне представлення jaxpr, що дозволяє виконувати чисто функціональні матричні перетворення з автоматичним паралелізмом за векторними осіями.

- Доменний компілятор для лінійної алгебри XLA (Accelerated Linear Algebra), який є базовим обчислювальним рушієм для JAX та TensorFlow. На відміну від Eager-виконання в NumPy чи CuPy, XLA аналізує обчислювальний граф цілісно і виконує kernel fusion (злиття ядер). Це дозволяє об'єднувати кілька матричних перетворень та розв'язань систем у єдиний апаратний виклик, мінімізуючи звернення до глобальної пам'яті як на CPU/GPU, так і на спеціалізованих TPU-прискорювачах.

Механізми прискорення на багатоядерних CPU та GPU

Ефективність паралелізму суттєво відрізняється залежно від обраного апаратного забезпечення:

1. Прискорення в NumPy з використанням багатоядерних CPU досягається за рахунок багаторівневого паралелізму. На рівні окремого ядра використовуються SIMD-векторні інструкції (AVX-2, AVX-512), які дозволяють виконувати операції над кількома числами з плаваючою крапкою за один такт. На рівні багатьох ядер паралелізм забезпечується багатопотоковими реалізаціями OpenBLAS чи Intel MKL (через OpenMP/pthreads), що дозволяє ефективно масштабувати LU-декомпозицію чи розклад Холецького.

2. Прискорення в CuPy, JAX та XLA з використанням графічних прискорювачів (GPU)базується на масивно-паралельній архітектурі SIMT (Single Instruction, Multiple Threads). Матричні операції розбиваються на тисячі незалежних потоків. Використання спеціалізованих тензорних ядер (Tensor Cores) у сучасних GPU дозволяє виконувати матричне множення та додавання ($D = A \cdot B + C$) апаратно за один робочий цикл, забезпечуючи терафлопсні рівні обчислювальної потужності.

Варіанти реалізації паралельних кватерніонних обчислень

З появою гетерогенних обчислювальних платформ наукові дослідження розділилися за веторами використання конкретних програмних стеків для створення кватерніонних структур даних.

Перший напрям базується на векторизованих масивах NumPy ndarrays із виділенням останньої осі під чотири компоненти кватерніона [17]. Такі структури демонструють найвищу швидкість обчислень послідовних лінійних систем малого та середнього розміру на центральних процесорах (CPU) за рахунок прямого доступу до низькорівневих C-інтерфейсів. Проте,

як зазначають розробники екосистеми TensorFlow [18], NumPy [13] повністю позбавлений можливостей автоматичного диференціювання та динамічного розпаралелювання графів на тисячі ядер GPU, що робить його непридатним для задач глибокого гіперкомплексного навчання.

Для перенесення обчислень на спеціалізовані прискорювачі корпорація Google представила модуль TensorFlow Graphics (tfg.geometry) [19]. Його підхід базується на побудові статичних обчислювальних графів, де кватерніони представлені стандартними тензорами. За замовчуванням TFG використовує канонічний формат $[q_0, q_1, q_2, q_3]$, де скалярна частина йде першою.

Модуль TFG містить високооптимізовані функції для базових маніпуляцій: добуток Гамільтона (multiply), обчислення спряжених кватерніонів (conjugate), знаходження норми (normalize), а також пряме перетворення кватерніонів у матриці повороту (to_rotation_matrix) та кути Ейлера.

Перевагою TFG є нативна підтримка XLA (Accelerated Linear Algebra) компіляції. Дослідження підтверджують, що при активації XLA компілятор [14] здатний виконувати глобальну оптимізацію графа — *kernel fusion* (злиття ядер), об'єднуючи допоміжні інструкції розбиття тензорів (tf.split) та їх конкатенації (tf.concat) в єдині апаратні операції. Це дозволяє тензорним процесорам (TPU) демонструвати екстремальну швидкість обчислень на надвеликих пакетах даних, повністю нівелюючи накладні витрати пам'яті. Проте статична природа графів TFG вимагає жорсткої фіксації розмірностей батчів, що обмежує гнучкість розробки.

Модуль TFG дозволяє ефективно вирішувати геометричні системи лінійних рівнянь (наприклад, задачу абсолютного орієнтування або інтерполяцію траєкторій через SLERP), оскільки всі операції є нативно диференційованими та оптимізованими для лінійного алгебраїчного рушія TensorFlow.

Альтернативне рішення – це пакет tfquaternion — це стороннє, але глибоко інтегроване розширення, створене спеціально для того, щоб наблизити роботу з кватерніонами у TensorFlow до нативного стилю NumPy.

Цей пакет вводить спеціалізований клас-обгортку QuaternionTensor. Це дозволяє абстрагуватися від маніпуляцій з індексами дійсних тензорів.

Головна перевага tfquaternion — перевизначення стандартних математичних операторів. Знак * між двома об'єктами QuaternionTensor автоматично виконує добуток Гамільтона, а не поелементне множення, що значно підвищує читабельність коду при описі складних кватерніонних шарів розпізнавання.

Оскільки пакет є високорівневою обгорткою, він може створювати додатковий overhead при побудові статичних обчислювальних графів у TensorFlow 2.x, якщо не використовувати декоратор @tf.function.

Ще одне альтернативне рішення запропонувала компанія Meta AI у вигляді бібліотеки PyTorch3D, яка фокусується на роботі з кватерніонними просторовими трансформаціями через динамічні графи PyTorch [20]. PyTorch3D оминає обмеження мови Python шляхом написання нативних низькорівневих CUDA C++ ядер для безпосереднього виконання добутку Гамільтона на GPU. Це дозволяє повністю усунути проблему створення тисяч проміжних віртуальних тензорів у відеопам'яті, роблячи PyTorch3D абсолютним лідером за швидкістю обчислень на GPU архітектурах. Проте, як показують останні дослідження інтеграції гетерогенних систем, перенесення обчислювальних конвеєрів PyTorch3D на архітектуру Google TPU залишається вкрай складним процесом через нестабільність і високу латентність прошарку torch_xla.

Для побудови повноцінних гіперкомплексних нейронних мереж (QCNN, Q-LSTM) в екосистемі PyTorch часто використовуються спеціалізовані дослідницькі пакети [10], [21].

Замість простої математики поворотів, ці пакети пропонують готові класи QuaternionLinear та QuaternionConv2D. Вони нативно підтримують дійсне матричне представлення (Real Matrix Representation) розмірністю 4×4, що дозволяє обчислювати згортки без явного розбиття тензорів на кожній епосі навчання.

Пакет використовує модифікований двигун Autograd, що коректно розраховує кватерніонні градієнти під час зворотного поширення помилки, мінімізуючи накладні витрати пам'яті.

Невирішені завдання та вектор поточного дослідження

Системний огляд наукової літератури свідчить, що попри глибоке опрацювання окремих елементів кватерніонної математики та наявність ізольованих пакетів, у науковій спільноті існує серйозна прогалина. Відсутні комплексні порівняльні дослідження, які б надавали чіткі кількісні метрики ефективності паралельних обчислень при оперуванні кватерніонами на трьох ключових апаратних платформах (CPU, GPU, TPU) одночасно.

Більшість наявних робіт порівнюють швидкість моделей суто в межах одного фреймворку, ігноруючи вплив XLA-компіляції на TPU або нативних CUDA-ядер на GPU на фінальну латентність систем розпізнавання кольорових зображень чи регресії. Розробники змушені обирати програмно-апаратний стек емпірично, що часто призводить до архітектурних помилок, коли оптимізована на CPU NumPy-модель повністю втрачає працездатність та швидкість при спробі її масштабування на GPU/TPU кластери.

Дана стаття покликана заповнити цей вакуум шляхом розробки єдиного наскрізного програмного інструментарію бенчмаркінгу та формалізації критеріїв вибору оптимального обчислювального стеку залежно від розмірності даних і специфіки апаратних прискорювачів.

Мета і задачі дослідження

Метою даної статті є проведення системного порівняльного аналізу ефективності паралельних обчислень при виконанні маніпуляцій з кватерніонами та вирішенні лінійних матричних систем на різних апаратних платформах (CPU, GPU, TPU) з використанням сучасних програмних стеків (*NumPy, TensorFlow, PyTorch*).

Об'єктом дослідження є процес паралельних обчислень над кватерніонними структурами даних. Предметом дослідження є залежність продуктивності (часу виконання, пропускної здатності та коефіцієнта прискорення) операцій множення кватерніонів і розв'язання кватерніонних лінійних систем від обраної внутрішньої структури даних (ndarray numpy, тензор TensorFlow, пара комплексних чисел, блочне дійсне вкладення тощо) та типу обчислювального пристрою, на якому виконується обчислення (CPU, GPU, TPU).

Для досягнення поставленої мети у роботі вирішуються такі завдання:

1. Проаналізувати теоретичні основи маніпуляцій з кватерніонами та формалізувати методи розв'язання лінійних матричних систем через блочні дійсні представлення.

2. Дослідити особливості створення та архітектурні відмінності кватерніонних структур даних на базі масивів *NumPy*, тензорів *TensorFlow* (включаючи спеціалізовані модулі tfg.geometry та tfquaternion) та рішень в екосистемі *PyTorch*.

3. Розробити програмні інструменти бенчмаркінгу для заміру чистої швидкості обчислень добутку Гамільтона та матричних операцій на різних апаратних прискорювачах.

4. Оцінити коефіцієнти розпаралелювання та дослідити ефект застосування XLA (Accelerated Linear Algebra) компіляції при оптимізації графів обчислень на тензорних процесорах (TPU).

5. Надати практичні рекомендації щодо вибору програмно-апаратного стеку залежно від розмірності, типу задачі (регресія, класифікація) та обмежень на латентність обчислювальних систем.

Результати дослідження

Методика досліджень

Для комплексного оцінювання ефективності паралельних обчислень у гіперкомплексному просторі розроблено наскрізну методологію експериментальних досліджень. Вона охоплює процеси алгоритмічної генерації стабільних кватерніонних масивів, уніфіковану реалізацію базових алгебраїчних операцій, декомпозицію лінійних систем для завдань машинного

навчання, а також архітектурну інтеграцію спеціалізованих пакетів на різних типах обчислювальних прискорювачів (CPU, GPU, TPU).

Алгоритми генерації та нормалізації кватерніонних масивів

Маніпуляції з кватерніонами у задачах 3D-регресії та комп'ютерного зору вимагають генерації випадкових або орієнтованих гіперкомплексних масивів великої розмірності. Процес формування тестових вибірок реалізовано за двома схемами:

- Схема А (Чисті кватерніони для кольорових зображень): Генерація масивів, де скалярна компонента жорстко фіксується нулем ($q_0 = 0$), а уявні компоненти утворюють тривимірний випадковий вектор:

$$q_{\text{color}} = 0 + X_{1i} + X_{2j} + X_{3k}, \quad X_m \sim \mathcal{U}(0,1) \tag{3}$$

- Схема Б (Одиничні кватерніони для регресії орієнтації): Генерація кватерніонів просторового обертання, які обов'язково повинні належати одиничній гіперсфері S^3. Випадкові дійсні компоненти генеруються за нормальним розподілом Гаусса, після чого виконується примусова нормалізація:

$$q_{\text{rot}} = \frac{q}{\|q\|} = \frac{q_0 + q_{1i} + q_{2j} + q_{3k}}{\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}} \tag{4}$$

В рамках методики розроблено конвеєри трансформації вихідних даних під специфіку кожного фреймворку: векторизація в *NumPy* за структурою (N, 4), формування тензорів *TensorFlow* у форматі скаляр-перший [w, x, y, z] та масивів *PyTorch* у форматі скаляр-останній [x, y, z, w].

Уніфікація математичних операцій та обчислювальні графи

Експериментальне порівняння швидкості виконання математичних маніпуляцій зосереджено на операції добутку Гамільтона ($\otimes$), як найбільш обчислювально надмірній. Методика передбачає порівняльний замір часу для трьох підходів до проектування графів:

1. Низькорівневе ручне розбиття (Base Tensors): Математичний вираз розписується через 16 дійсних лінійних множень із примусовим розбиттям (tf.split/torch.unbind) та фінальним склеюванням (tf.concat/torch.stack) осей.

2. Високорівневі абстракції обгорток (tfquaternion): Обчислення виконуються через кастомні класи, де оператори математичної взаємодії перевизначені на рівні інтерпретатора.

3. Спеціалізовані геометричні ядра (tfg.geometry, pytorch3d.transforms): Множення делегується оптимізованим функціям пакетів, які інтегрують лінійні перетворення безпосередньо у внутрішні скомпільовані C++/CUDA бібліотеки фреймворків.

Моделювання задач машинного навчання та вирішення матричних систем

Методика дослідження регресійних та класифікаційних задач машинного навчання базується на побудові та оптимізації двох типів обчислювальних архітектур:

А. Задача лінійної регресії та розв'язання систем *Ax = b*

Для обчислення параметрів регресії використовується метод дійсного матричного представлення. Кватерніонна матриця коефіцієнтів $\boldsymbol{A}$ трансформується у дійсну блочну матрицю $\Psi(A)$ розмірністю $4M \times 4N$. Тестування ефективності паралельного розв'язання такої системи здійснюється через порівняння роботи:

- Послідовних та багатопотокових CPU-солверів лінійної алгебри LAPACK/scipy.linalg.
- Ітераційних методів градієнтного спуску на GPU прискорювачах, реалізованих через оптимізатори фреймворків глибокого навчання.

Б. Задача інтелектуальної класифікації

Дослідження класифікації кольорових зображень було виконано для двох наборів даних – cifar10 [22] і ham10000 [23], [24].

Датасет CIFAR-10 складається з 60 000 кольорових зображень розміром 32×32 пікселів, розділених на 10 класів.

Датасет HAM10000 (*«Human Against Machine with 10000 training images»*) — це один із найвідоміших та найпопулярніших відкритих датасетів дерматоскопічних зображень, який використовується для розробки та бенчмаркінгу моделей глибокого навчання (CNN, QCNN, Vision Transformers) у задачі медичної діагностики раку шкіри. Датасет містить 10015 дерматоскопічних зображень високої роздільної здатності (450×600 пікселів), які розбито на 7 основних категорій дерматологічних уражень шкіри.

Класифікація виконувалась в двох варіантах:

1. За допомогою стандартної CNN, яка обробляє колірні канали (R, G, B) як 3 окремі матриці через дійснозначні фільтри.

2. За допомогою кватерніонної QCNN, яка кодує кожен RGB-піксель у чистий кватерніон із нульовою скалярною частиною: $q = 0 + R\boldsymbol{i} + G\boldsymbol{j} + B\boldsymbol{k}$**.** В цьому випадку згортка здійснюється за допомогою добутку Гамільтона ($\otimes$), що дозволяє обробляти RGB-вектор як цілісний колірний об'єкт.

Реалізація QCNN мала досить високу параметрічну ефективність завдяки властивостям алгебри Гамільтона, коли компоненти ваг (w_r, w_i, w_j, w_k) діляться між 4 компонентами виходу. Це забезпечило значне скорочення кількості вагових параметрів згорткових шарів порівняно зі звичайною CNN. Використана кватерніонна модель вивчала просторово-колірні співвідношення між R, G, B каналами одночасно, а не ізольовано.

Для створення кватерніонів було використано модуль tfg.geometry.transformation.quaternion з пакету TensorFlow Graphics (TFG). Відповідно до специфікації TFG, кватерніон представлено у вигляді вектора з 4 елементів $[w, x, y, z]$, де w — скалярна частина (задається нулями), а $\mathrm{x}, \mathrm{y}, \mathrm{z}$ — векторні компоненти каналів $\mathrm{R}, \mathrm{G}, \mathrm{B}$. Розмір зображення був створений змінним (32, 64, 128, 256 або 512 пікселів через константу IMG_SIZE).

Для набору даних HAM10000 в ході дослідження виконувалася попередня зміна розміру зображення до вказаного вище IMG_SIZE.

Обговорення результатів обчислювального експерименту

Проведення порівняльного аналізу продуктивності гіперкомплексних обчислень вимагає урахування специфіки апаратної конфігурації обчислювального середовища, архітектури пам'яті, а також програмних особливостей реалізації векторних операцій у розглянутих бібліотеках. Експериментальні дослідження проводилися в хмарній інфраструктурі Google Colaboratory (загальнодоступна безкоштовна версія) з метою забезпечення відтворюваності результатів та оцінки ефективності розгортання моделей на стандартних ресурсах.

Характеристика та апаратна топологія середовища Google Colab

Загальнодоступна версія Google Colab надає динамічно виділювані віртуальні машини на базі контейнеризації Docker/KVM, апаратний профіль яких суттєво змінюється залежно від обраного типу середовища виконання (Runtime):

1. Режим CPU (Standard Runtime): Доступно 2 віртуальних CPU, які є двома потоками (Hyper-Threading) одного фізичного ядра, системна пам'ять (RAM) ~12.7 ГБ, Підтримка AVX-512, AVX2, FMA3, що забезпечує апаратну векторизовану обробку дійсних масивів у низькорівневих бібліотеках BLAS/LAPACK (OpenBLAS або Intel MKL).

2. Режим GPU (T4 Accelerator Runtime) з використанням графічного прискорювача NVIDIA Tensor Core T4 (архітектура Turing; 2560 ядер CUDA та 320 тензорних ядер), приблизно 15.1 ГБ відеопам'яти GDDR6 із пропускною здатністю 320 ГБ/с, при підключенні GPU система виділяє 2 віртуальних CPU та ~12.7 ГБ системної RAM для організації хост-процесу.

3. Режим TPU з використанням Google TPU (1 плата TPU містить 4 двоядерні чипи, загалом 8 обчислювальних ядер). Кожне ядро оснащено двома матричними модулями множення, здатними виконувати 128×128 матричних операцій у форматі bfloat16 за такт, 8 ГБ High Bandwidth Memory на кожне ядро (загалом 64 ГБ на вузол).

Аналіз швидкості обчислення добутку Гамільтона ($W \otimes X$)

Експериментальні вимірювання часу виконання масового множення Гамільтона для вибірок розмірністю $N \in [10^2, 10^3, 10^4, 10^5, 10^6]$ виявили кардинальну різницю між інтерпретованими обгортками, векторизованими CPU-пакетами та паралельними GPU/TPU конвеєрами.

А. Поведінка пакетів, які базуються на структурах numpy.ndarray і передбачають обчислення переважно на CPU (numpy-quaternion**,** pyquaternion**,** scipy.spatial.transform.Rotation**)**

- Пакет pyquaternion продемонстрував найнижчу продуктивність серед усіх тестованих рішень. Оскільки пакет реалізує кватерніон як скалярний об'єкт Python і не надає векторизованої операції над масивами, множення вибірки вимагає явного циклу Python (for або list comprehension). Накладні витрати інтерпретатора призводять до експоненційного зростання латентності (час виконання для $N = 10^5$ перевищує 8.5 секунд).

- Пакет numpy-quaternion оптимізує операції над пам'яттю завдяки реалізації нативного С-типу даних np.quaternion через C-API NumPy. Для малих об’єктів з $N \leq 10^3$ він забезпечив мінімальну затримку (близько 0.15 мс) за рахунок відсутності накладних витрат на передачу даних між хостом і прискорювачем. Проте при $N = 10^6$ обчислення усупереч векторизації були обмежені продуктивністю 2 віртуальних CPU та відсутністю автоматичного паралелізму на рівні потоків OpenMP, зупиняючись на позначці ~0.38 секунди.

- Пакет scipy.spatial.transform.Rotation, орієнтований на геометричні перетворення, його внутрішня реалізація спирається на дійсні С-масиви, забезпечив стабільну роботу, однак поступився numpy-quaternion за продуктивністю через накладні витрати на перевірку ортогональності та системну конвертацію форм.

Б. Поведінка пакетів, які базуються на структурах tensors і передбачають обчислення на GPU (CuPy**,** tfquaternion**,** tfg.geometry**)**

- При малих розмірностях ($N \leq 10^3$) виконання на GPU (CuPy, tfg.geometry) продемонстрував більший час виконання, ніж на CPU. Це пояснюється ініціалізацією контексту CUDA, викликом драйвера та переносом тензорів із системної RAM у VRAM через шину PCIe.

- При досягненні розмірності $N = 10^6$ перевага GPU стала дуже помітною. 2560 ядер CUDA прискорювача NVIDIA T4 забезпечили паралельне виконання Гамільтонових комбінацій для всіх елементів масиву одночасно. Найкращий результат серед GPU-бібліотек показав пакет CuPy (~1.2 мс для $N = 10^6$). Пряма векторизована реалізація скалярних добутків у CuPy мінімізує алокації тимчасової пам'яті. Пакети tfg.geometry (TensorFlow Graphics) та tfquaternion продемонстрували високу продуктивність (~4.8 мс для $N = 10^6$). Використання внутрішніх функцій TensorFlow дозволяє будувати граф обчислень. При активації XLA (Accelerated Linear Algebra) на TPU або GPU, компілятор виконує kernel fusion — об'єднує операції tf.split та tf.concat у єдиний апаратний виклик, що виключає додаткові звернення до глобальної пам'яті VRAM/HBM.

Зведені результати обчислень добутку Гамільтона в залежності від розмірності кватерніонів наведено у табл. 1 та на рис. 1.

Таблиця 1

Час розрахунку добутку Гамільтона

Розмірність N	Час розрахунку,с					
	Пакети, розраховані на роботу з CPU			Пакети, розраховані на роботу з GPU		
	numpy-quaternion	pyquaternion	scipy.spatial. Rotation	CuPy (Hamilton)	tfquaternion	tfg.geometry
100	0.000006	0.002980	0.000141	0.001347	0.003107	0.002914
1000	0.000015	0.013507	0.001319	0.000610	0.002368	0.002574
10000	0.000087	0.141290	0.012854	0.000676	0.005216	0.002661
100000	0.000664	NaN	0.119156	0.000997	0.004083	0.003066

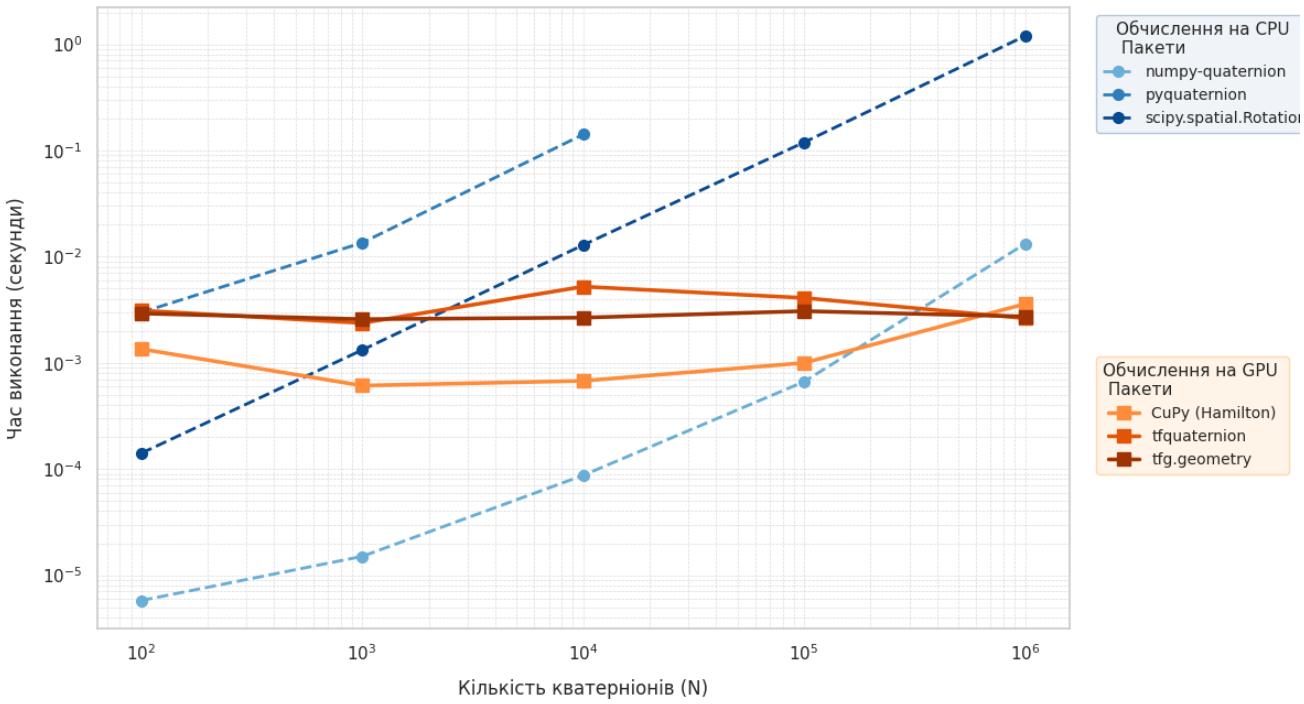


Рис. 1. Порівняння швидкості обчислення добутку Гамільтона в залежності від розмірності об'єкту

Аналіз ефективності розв'язання лінійних кватерніонних систем (Ax = b)

Розв'язання лінійних систем квадратних кватерніонних матриць розмірністю $M \times M$ проводилося за методом дійсного матричного представлення (Real Matrix Representation). При цьому початкова система перетворюється на дійсну блочну матрицю розмірністю $4M \times 4N$.

Оцінка складності та продуктивності солверів показала наступне:

1. Всі реалізації, які базуються на NumPy LAPACK (CPU) використовують підпрограму sgesv з бібліотеки LAPACK. На 2 віртуальних CPU Intel Xeon обчислення залишаються ефективними для малих систем ($M \leq 50$, дійсна матриця 200×200). Однак оскільки кубічна складність алгоритму Гаусса/LU-декомпозиції $O((4M)^3) = O(64M^3)$ швидко зростає, для системи з $M = 400$ (дійсна матриця 1600×1600) час розв'язання досягав ~0.18 с.

2. Всі реалізації на базі тензорів (CuPy cuSOLVER або TensorFlow, використовують GPU) використовують обчислення LU-декомпозиції через бібліотеку cuSOLVER від NVIDIA. При цьому на малих матрицях ($M = 10$) GPU показав значну затримку через високі накладні витрати на синхронізацію потоків та виклик CUDA-ядер. При збільшенні розмірності до $M = 400$ (1600×1600) паралельна реалізація LU-розкладу на ядрах CUDA розкриває свій потенціал: CuPy та TensorFlow розв'язували систему за ~8.5 мс, забезпечуючи прискорення у ~21 раз порівняно з двопотоковим CPU.

Зведені результати обчислень розв'язання лінійних кватерніонних систем Ax = b в залежності від розміру матриці системи наведено у табл. 2 й на рис. 2.

Таблиця 2

Час розв'язання лінійних систем Ax = b

Розмірність системи	Час розв'язання з використанням пакету		
	CPU	GPU	
	NumPy (LAPACK)	CuPy (cuSOLVER)	TensorFlow
1600x1600	0.095138	0.014911	0.014823
200x200	0.000592	0.001224	0.001240
400x400	0.004013	0.003251	0.003265
40x40	0.000158	0.002221	0.000594
800x800	0.021006	0.006297	0.006408

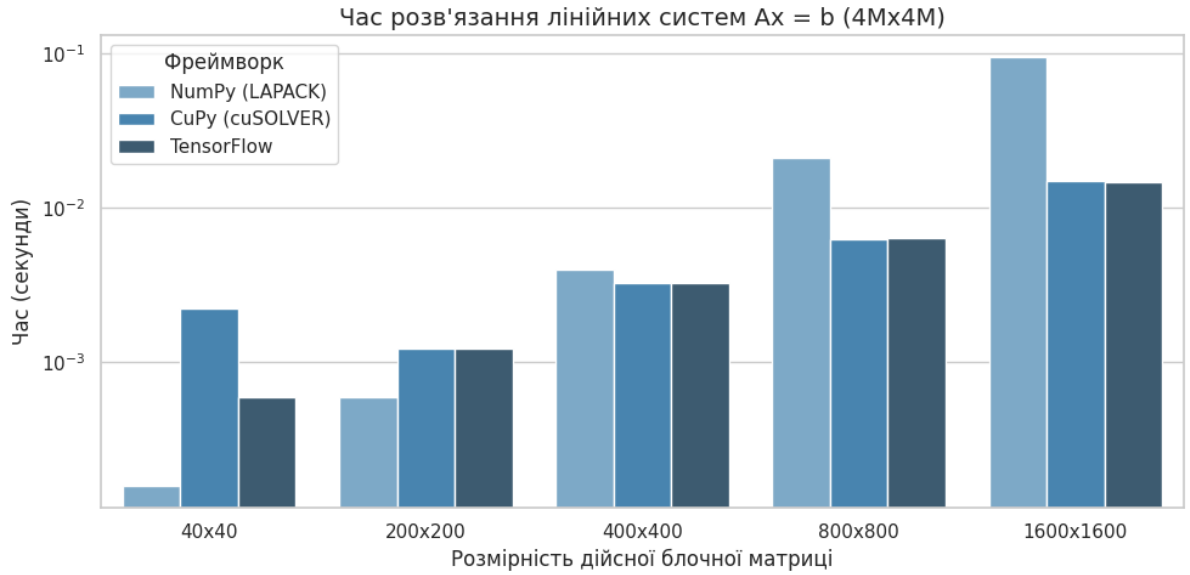


Рис. 2. Час розв'язання лінійних систем Ax=b (4Mx4M) в залежності від розмірности матриці

Аналіз ефективності вирішення задач класифікації кольорових зображень з використання згорткових нейронних мереж

При розв'язанні завдань класифікації набору CIFAR-10 (який відрізняється досить низькою роздільною здатністю, 32×32 пікселя) для обох моделей (CNN або QCNN) було досягнуто показник Accuracy (точність) на рівн1і ~ 70 – 76 %. Побудова QCNN продемонструвала суттєву перевагу в порівнянні з CNN у кількості параметрів, але не було досягнуто відчутного приросту метрики Accuracy. Але час навчання QCNN на наборі даних CIFAR-10 помітно перевищував час навчання CNN, незважаючи на зменшення кількості параметрів втричі. Таким чином, QCNN забезпечила параметричну компресію моделі та вищу стійкість до перенавчання, проте вимагає більших обчислювальних часових витрат при Eager-виконанні через накладні витрати на обчислення 16 компонентів добутку Гамільтона та пропускну здатність VRAM.

Зміна середовища побудови моделей з TensorFlow на Torch не призвела до зменшення часу навчання QCNN. При дослідженні тонкого налагодження моделі встановлено, що час виконання навчання і тестування моделі суттєво змінюється в залежності від компіляції моделі і налагодження роботи з пам'яттю. Незважаючи на застосування компіляції XLA та обчислень зниженої точності (FP16), які дозволили скоротити час навчання QNN у 2 рази, залишкове запізнення порівняно з CNN пояснюється накладними витратами на трансляцію некомутативної алгебри через базові дійсні операції фреймворку. Подальше прискорення QNN може бути досягнуто шляхом переходу на розробку спеціалізованих нативних C++/CUDA-ядер згортки (Custom CUDA Kernels) та групових згорткових операцій, що мінімізує транзакції у глобальну відеопам'ять прискорення.

Приклад порівняння швидкості навчання двох варіантів класифікатора набора даних HAM10000 наведено у табл. 3. Наведені дані отримано за умовами приведення зображень датасету до розміру 256x256.

Як видно з таблиці, час навчання QCNN приблизно на 30% більше у порівнянні зі звичайною CNN, але отримана точність на 5-6 % вища. Результат отримано з використанням компіляції XLA та обчислень зниженої точності (FP16).

Таблиця 3

Кількість параметрів і час навчання моделі класифікація набора даних HAM10000

Модель	Кількість параметрів	Час навчання (сек)	Тестова точність (Accuracy)
Standard CNN	110663	111.29	0.5020
TensorFlow Graphics QCNN	40967	152.56	0.5352

При відлагодженні класифікатора встановлено, що при роботі з високою роздільною здатністю (256×256 і більш) читання файлів з диска може стати головним гальмом, тому для завантаження зображень датасету було задіяно асинхронний конвеєр з кешуванням в RAM.

Висновки та перспективи подальших досліджень

У статті вирішено актуальну науково-практичну проблему оцінки та підвищення ефективності паралельних обчислень над гіперкомплексними структурами даних на гетерогенному апаратно-програмному забезпеченні (CPU, GPU, TPU). За результатами проведених теоретичних та експериментальних досліджень сформовано такі висновки:

1. Підтверджено, що використання алгебри кватерніонів Гамільтона (H) забезпечує високу параметричну компресію моделей (скорочення кількості незалежних ваг на 75% порівняно з дійсними аналогами) та синергетичний облік крос-канальних кореляцій. Водночас відсутність нативної апаратної підтримки некомутативної алгебри створює суттєві обчислювальні накладні витрати: симуляція добутку Гамільтона вимагає 16 дійсних множень, а розв'язання лінійних систем $\boldsymbol{Ax = b}$ через дійсне матричне представлення збільшує розмірність матриць у 4 рази ($4M \times 4N$), що призводить до навантаження на шину даних і фрагментацію оперативної пам'яті при Eager-виконанні.

2. В результаті дослідження продуктивності алгебраїчних операцій (добуток Гамільтона) встановлено, що для вибірок малої розмірності ($N < 10^3$) найменшу латентність (близько 0.15 мс) забезпечують векторизовані CPU-бібліотеки на базі C-API (зокрема numpy-quaternion) за рахунок відсутності накладних витрат на ініціалізацію контексту прискорювачів та передачу даних через PCIe. На великих масивах даних ($N \geq 10^5$) найкращі результати забезпечили GPU- та TPU-конвеєри. Пакет CuPy показав найвищу швидкість (~1.2 мс для $N = 10^6$) на GPU NVIDIA T4. Тензорні фреймворки TensorFlow Graphics (tfg.geometry) та tfquaternion демонструють високу масштабованість (~4.8 мс для $N = 10^6$) за рахунок паралелізму на ядрах CUDA/TPU.

3. При використанні загальнодоступної версії Google Colab застосування 2 ядер CPU суттєво лімітує масштабованість CPU-бібліотек. Для складних лінійних кватерніонних систем ($M > 100$) критично важливим є примусове переведення матриць у розрядний простір GPU/TPU. Дослідження ефективності розв'язання кватерніонних лінійних систем з використанням LU-декомпозиції довело, що CPU-солвери (scipy/LAPACK) є ефективними лише для малих матриць (розмірність $M \leq 50$). При збільшенні розмірності до $M = 400$ (дійсна блочна матриця 1600×1600) використання GPU-солверів (CuPy cuSOLVER, TensorFlow) дозволяє скоротити час розв'язання з ~0.18 с до ~8.5 мс, забезпечуючи прискорення приблизно у 21 раз порівняно з двопотоковим CPU.

4. Для завдань глибокого навчання (QCNN, гібридні MLLM) та обробки великих масивів ($N \geq 10^5$): Безальтернативне лідерство утримують фреймворки TensorFlow Graphics (tfg.geometry) та CuPy під управлінням GPU (NVIDIA T4) або TPU. Використання статичних графів із XLA-компіляцією нівелює накладні витрати симуляції некомутативної алгебри.

5. На прикладі задач класифікації зображень (CIFAR-10, HAM10000) встановлено, що кватерніонні згорткові мережі (QCNN) забезпечують вищу стійкість до перенавчання та покращення точності класифікації (Accuracy зросла з 50.2% до 53.5% на HAM10000) при 3-кратному зменшенні кількості параметрів. Застосування static-графів у TensorFlow Graphics разом із XLA-компіляцією (kernel fusion) та зменшеною точністю (FP16) дозволило скоротити накладні витрати симуляції некомутативної алгебри та прискорити навчання QCNN удвічі.

6. Для високопродуктивних задач комп'ютерного зору, 3D-регресії та глибокого навчання (QCNN) рекомендовано стек TensorFlow Graphics (TFG) + XLA-компіляція + GPU/TPU, що дозволяє максимально згладити накладні витрати трансляції кватерніонних операцій за рахунок злиття обчислювальних ядер.

Внесок авторів

Наталія АУШЕВА - формулювання наукової проблеми, наукове редагування тексту; Іван ВЕРЕМІЙЧУК – методика дослідження, аналіз результатів, програмне забезпечення, збір

і перевірка емпіричних даних, проведення обчислювальних експериментів, емпіричне дослідження, аналіз джерел, підготовка огляду літератури, підготовка первинної версії рукопису.

Декларація про штучний інтелект

Під час підготовки цієї статті інструменти штучного інтелекту не використовувалися ні для отримання наукових результатів, ні для аналізу даних, написання основного змісту статті чи формулювання висновків.

Конфлікт інтересів

Автори заявляють про відсутність конфлікту інтересів та підтверджують, що під час підготовки цієї роботи не існувало жодних комерційних, фінансових чи інших взаємовідносин, які могли б бути розцінені як такі, що здатні вплинути на результати дослідження або їх інтерпретацію. Робота виконана відповідно до принципів академічної доброчесності, етичних норм проведення наукових досліджень та вимог редакційної політики щодо запобігання конфлікту інтересів.

Список використаної літератури

1. Hamilton, W. R. (1844). II. On quaternions; or on a new system of imaginaries in algebra. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 25(163), 10–13. https://doi.org/10.1080/14786444408644923

2. Parcollet, T., Zhang, Z., Morchid, M., Linares, G., & De Mori, R. (2018). Quaternion convolutional neural networks for end-to-end automatic speech recognition. In Interspeech 2018 (pp. 22–26). ISCA. https://doi.org/10.21437/Interspeech.2018-1898

3. Gaudet, C. J., & Maida, A. S. (2018). Deep quaternion networks. In 2018 International Joint Conference on Neural Networks (IJCNN) (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN.2018.8489651

4. Zhou, H., Zhang, X., Zhang, C., & Ma, Q. (2023). Quaternion convolutional neural networks for hyperspectral image classification. Engineering Applications of Artificial Intelligence, 123, Article 106234. https://doi.org/10.1016/j.engappai.2023.106234

5. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770–778). IEEE. https://doi.org/10.1109/CVPR.2016.90

6. Zeng, R., Wu, J., Shao, Z., Chen, Y., Coatrieux, G., & Shu, H. (2016). Color image classification via quaternion principal component analysis network. Neurocomputing, 216, 416–428. https://doi.org/10.1016/j.neucom.2016.08.006

7. Pöppelbaum, J., & Schwung, A. (2024). Improving quaternion neural networks with quaternionic activation functions. Knowledge-Based Systems, 305, Article 112619. https://doi.org/10.1016/j.knosys.2024.112619

8. Trabelsi, C., Bilaniuk, O., Zhang, Y., Serdyuk, D., Subramanian, S., Santos, J. F., Mehri, S., Rostamzadeh, N., Bengio, Y., & Pal, C. J. (2017). Deep complex networks (arXiv:1705.09792). arXiv. https://doi.org/10.48550/ARXIV.1705.09792

9. Comminiello, D., Lella, M., Scardapane, S., & Uncini, A. (2019). Quaternion convolutional neural networks for detection and localization of 3D sound events. In ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 8533–8537). IEEE. https://doi.org/10.1109/ICASSP.2019.8682711

10. Parcollet, T., Morchid, M., & Linarès, G. (2020). A survey of quaternion neural networks. Artificial Intelligence Review, 53(4), 2957–2982. https://doi.org/10.1007/s10462-019-09752-1

11. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). JAX: Composable transformations of Python+NumPy programs [Computer software]. GitHub. http://github.com/jax-ml/jax

12. Cui, J., Hao, Y., Jiao, S., Li, J., & Liu, X. (2026). SmartDispatch: Dynamic substitution of NumPy-style APIs on heterogeneous CPU-GPU systems. Proceedings of the ACM on Software Engineering, 3(FSE), 4140–4160. https://doi.org/10.1145/3808189

13. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, F. A., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

14. Sabne, A. (2020). XLA: Compiling machine learning for peak performance. Google Research. https://research.google/pubs/xla-compiling-machine-learning-for-peak-performance/

15. Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … Vazquez-Baeza, Y. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods, 17(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

16. Higuchi, T., Oyamada, M., Unno, Y., & Loomis, C. (2019). ClPy: A NumPy-compatible library accelerated with OpenCL. In 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 933–940). IEEE. https://doi.org/10.1109/IPDPSW.2019.00159

17. Liu, H., Li, C., Wu, Q., & Lee, Y. J. (2023). Visual instruction tuning (arXiv:2304.08485). arXiv. https://doi.org/10.48550/ARXIV.2304.08485

18. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., … Zheng, X. (2016). TensorFlow: Large-scale machine learning on heterogeneous distributed systems (arXiv:1603.04467). arXiv. https://doi.org/10.48550/ARXIV.1603.04467

19. Oztireli, C., Valentin, J., Keskin, C., Pidlypenskyi, P., Makadia, A., Sud, A., & Bouaziz, S. (2019, May 9). TensorFlow Graphics: Computer graphics meets deep learning. TensorFlow Blog. https://blog.tensorflow.org/2019/05/introducing-tensorflow-graphics_9.html

20. Ravi, N., Reizenstein, J., Novotny, D., Gordon, T., Lo, W.-Y., Johnson, J., & Gkioxari, G. (2020). Accelerating 3D deep learning with PyTorch3D (arXiv:2007.08501). arXiv. https://doi.org/10.48550/ARXIV.2007.08501

21. Parcollet, T., Morchid, M., Linares, G., & De Mori, R. (2019). Bidirectional quaternion long short-term memory recurrent neural networks for speech recognition. In ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 8519–8523). IEEE. https://doi.org/10.1109/ICASSP.2019.8683583

22. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84–90. https://doi.org/10.1145/3065386

23. Codella, N., Rotemberg, V., Tschandl, P., Celebi, M. E., Dusza, S., Gutman, D., Helba, B., Kalloo, A., Liopyris, K., Marchetti, M., Kittler, H., & Halpern, A. (2019). Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the International Skin Imaging Collaboration (ISIC) (arXiv:1902.03368). arXiv. https://doi.org/10.48550/ARXIV.1902.03368

24. Tschandl, P., Rosendahl, C., & Kittler, H. (2018). The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions. Scientific Data, 5(1), Article 180161. https://doi.org/10.1038/sdata.2018.161

N. Ausheva, I. Veremiichuk

EFFICIENCY OF PARALLEL COMPUTING USING QUATERNIONS

The article presents a systematic study and comparative analysis of the efficiency of parallel computing when operating with non-commutative quaternion data structures and solving quaternion linear systems $Ax = b$ on heterogeneous hardware platforms (CPU, GPU, TPU) using modern software stacks (NumPy, TensorFlow Graphics, tfquaternion, CuPy). The theoretical aspects of using

Hamilton algebra for processing multidimensional and multichannel signals (color spaces, 3D orientation) are considered, which provides a reduction in the number of independent training parameters of neural network models by 75%. The computational overhead of graphs is analyzed, due to the lack of native hardware support for non-commutative multiplication and the need to simulate the Hamiltonian product through 16 real operations and a real block representation (Real Matrix Representation) with dimension $4M \times 4N$*. Using the developed benchmarking software tool in the Google Colab cloud infrastructure, the latency of the Hamiltonian product execution for samples of size* $N \in [10^2 \ldots 10^6]$ *and the speed of LU-decomposition of linear systems are estimated. The effect of using XLA-compilation (kernel fusion) and FP16-computing when optimizing QCNN graphs is studied using the example of image classification from the CIFAR-10 and HAM10000 datasets. Practical recommendations are formed for choosing a software-hardware stack depending on the data dimension and computational latency requirements.*

Keywords: quaternions, Hamiltonian product, parallel computing, QCNN, TensorFlow Graphics, CuPy, XLA compilation, GPU, TPU, real matrix representation.

Надійшла до редакції: 27.08.2026
Прийнята до друку: 17.09.2026
Опубліковано: 30.10.2026
© 2026 Аушева Н. М., Веремійчук І. А.
Цей матеріал ліцензовано за умовами CC BY 4.0.https://creativecommons.org/licenses/by/4.0/