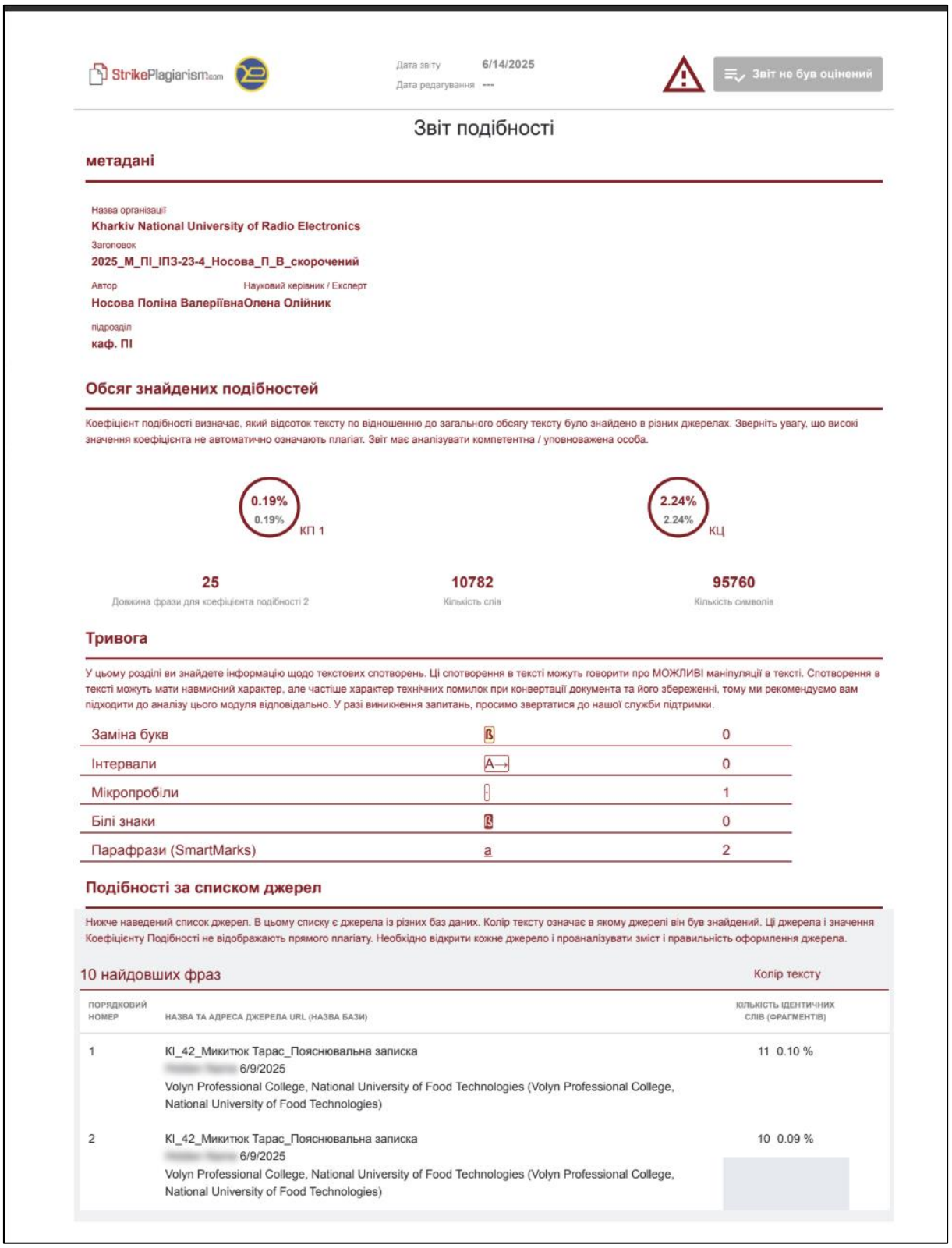
ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



StrikePlagiarism.com

Дата звіту 6/14/2025
Дата редагування ---

Звіт не був оцінений

Звіт подібності

метадані

Назва організації
Kharkiv National University of Radio Electronics
Заголовок
2025_М_ПІ_ІПЗ-23-4_Носова_П_В_скорочений
Автор — Науковий керівник / Експерт
Носова Поліна ВалеріївнаОлена Олійник
підрозділ
каф. ПІ

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.19% 0.19% КП 1

2.24% 2.24% КЦ

25	10782	95760
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	В	0
Інтервали	A→	0
Мікропробіли	·	1
Білі знаки	¶	0
Парафрази (SmartMarks)	a	2

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	КІ_42_Микитюк Тарас_Пояснювальна записка [redacted] 6/9/2025 Volyn Professional College, National University of Food Technologies (Volyn Professional College, National University of Food Technologies)	11 0.10 %
2	КІ_42_Микитюк Тарас_Пояснювальна записка [redacted] 6/9/2025 Volyn Professional College, National University of Food Technologies (Volyn Professional College, National University of Food Technologies)	10 0.09 %

з бази даних RefBooks (0.00 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)

з домашньої бази даних (0.00 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)

з програми обміну базами даних (0.19 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	КІ_42_Микитюк Тарас_Пояснювальна записка [redacted] 6/9/2025 Volyn Professional College, National University of Food Technologies (Volyn Professional College, National University of Food Technologies)	21 (2) 0.19 %

з Інтернету (0.00 %)

ПОРЯДКОВИЙ НОМЕР	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)

Список прийнятих фрагментів (немає прийнятих фрагментів)

ПОРЯДКОВИЙ НОМЕР	ЗМІСТ	КІЛЬКІСТЬ ОДНАКОВИХ СЛІВ (ФРАГМЕНТІВ)

ВСТУП

Мобільні технології кардинально змінили спосіб взаємодії людей з цифровою інформацією. За останнє десятиліття смартфони перетворилися з простих засобів комунікації на потужні обчислювальні платформи, які супроводжують користувачів протягом усього дня. У 2024 році кількість користувачів смартфонів у світі перевищила 6,8 мільярда людей, а середній час використання мобільних додатків становить понад 4 години на день [1]. Сучасні мобільні додатки характеризуються складністю архітектури, інтенсивним використанням мережевих ресурсів та високими вимогами до продуктивності. Платформа Android, яка займає понад 70% світового ринку мобільних операційних систем, надає розробникам широкі можливості для створення інноваційних рішень, проте водночас висуває жорсткі вимоги щодо ефективного використання системних ресурсів та забезпечення плавного користувацького досвіду.

Одним із головних пріоритетів сучасної мобільної розробки є створення додатків, які забезпечують швидкий та плавний користувацький досвід. Кожен користувач мобільного пристрою очікує миттєвого відгуку від додатку, особливо при взаємодії з візуальним контентом. Дослідження показують, що затримка відгуку понад 100 мілісекунд сприймається користувачами як помітна, а затримка понад 1 секунди призводить до втрати концентрації та негативного враження від додатку [2].

Серед усіх категорій мобільних додатків особливе місце займають соціальні мережі та додатки для обміну візуальним контентом. Такі платформи, як Instagram, Facebook, TikTok та інші, щодня обслуговують мільярди користувачів, забезпечуючи безперервний потік зображень, відео та іншого медіаконтенту. Швидка та ефективна робота цих додатків може значно вплинути на задоволеність користувачів, тривалість сесій та, зрештою, на комерційний успіх платформи.

Завантаження та обробка зображень стали основою функціонування сучасних мобільних додатків соціальних мереж. При прокручуванні стрічки новин користувач може переглядати десятки зображень за хвилину, що створює значне навантаження на систему. Аналіз завантаження зображень у мобільних додатках – це складна процедура, яка потребує ефективного використання системних ресурсів, оптимального управління пам'яттю та мінімізації енергоспоживання.

Багато проблем продуктивності в мобільних додатках пов'язані з

ДОДАТОК Б

Слайди презентаці

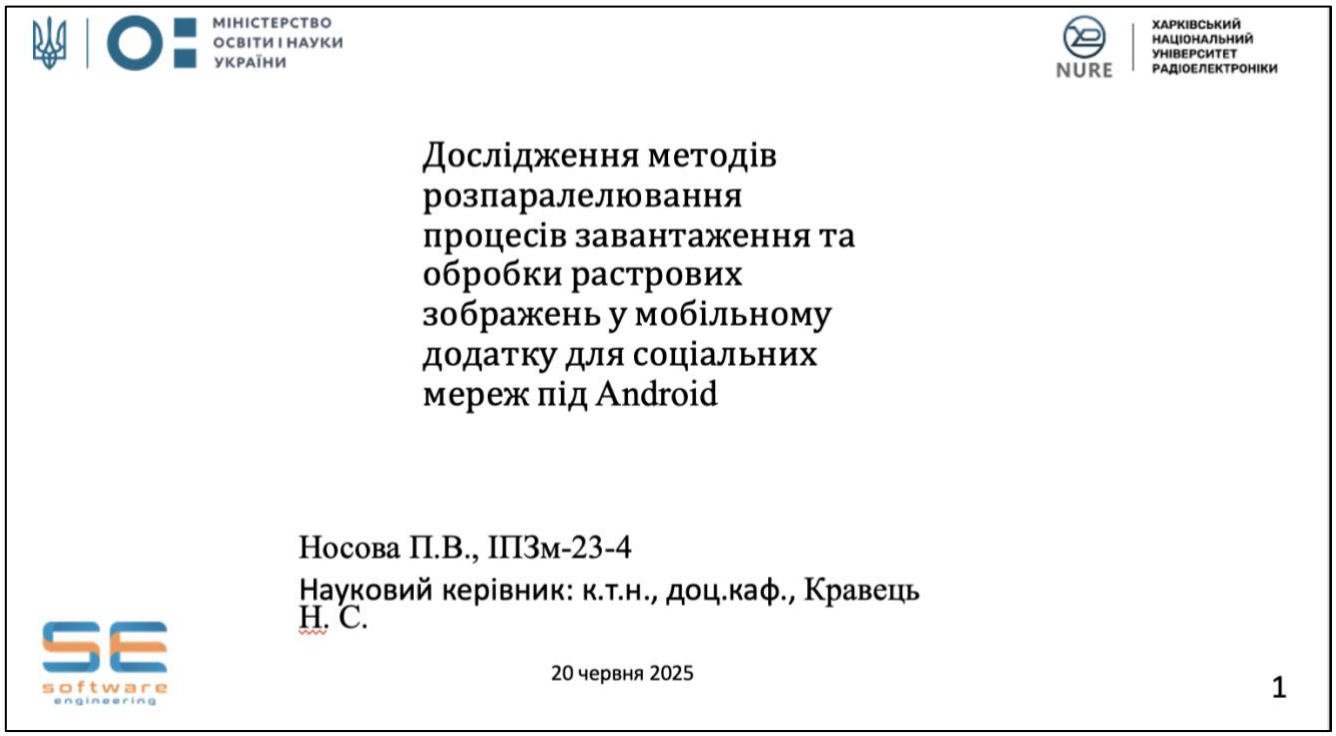


Рисунок Б.1 – Слайд 1 (рисунок створений самостійно)

Рисунок Б.2 – Слайд 2 (рисунок створений самостійно)

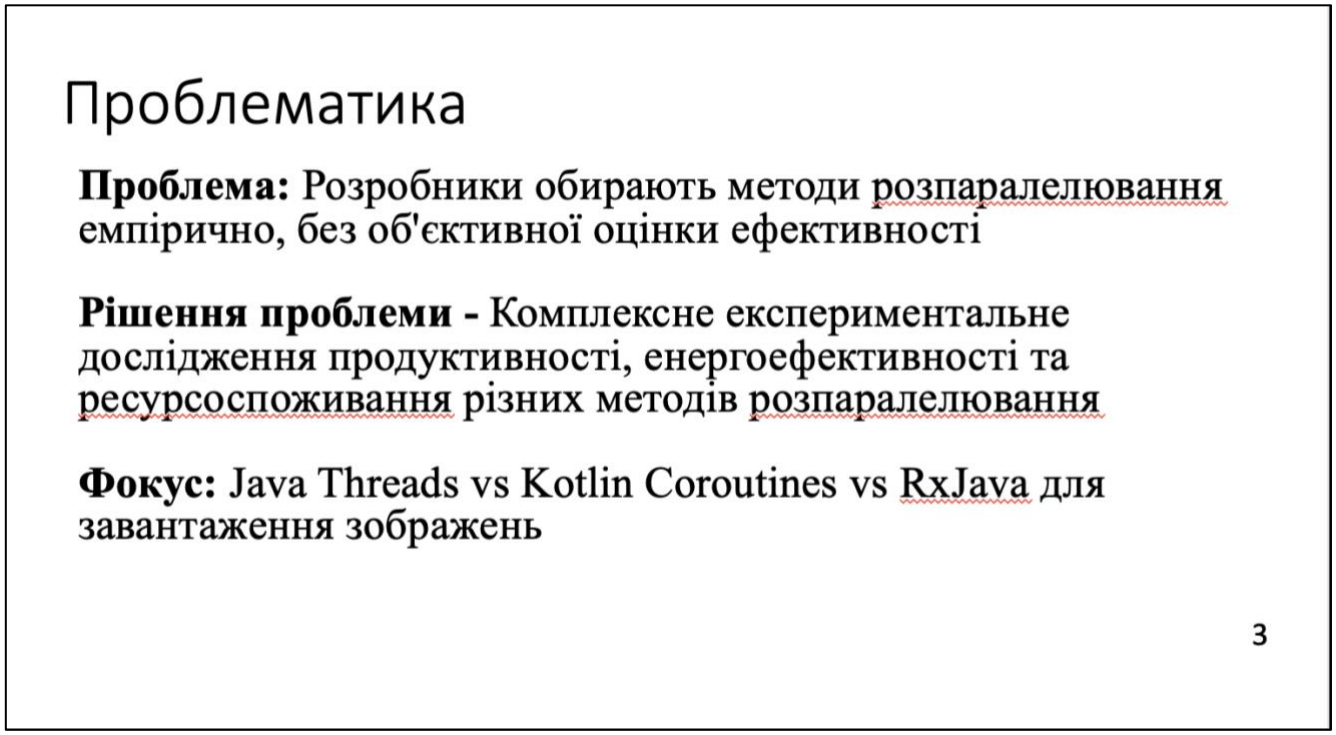


Рисунок Б.3 – Слайд 3 (рисунок створений самостійно)

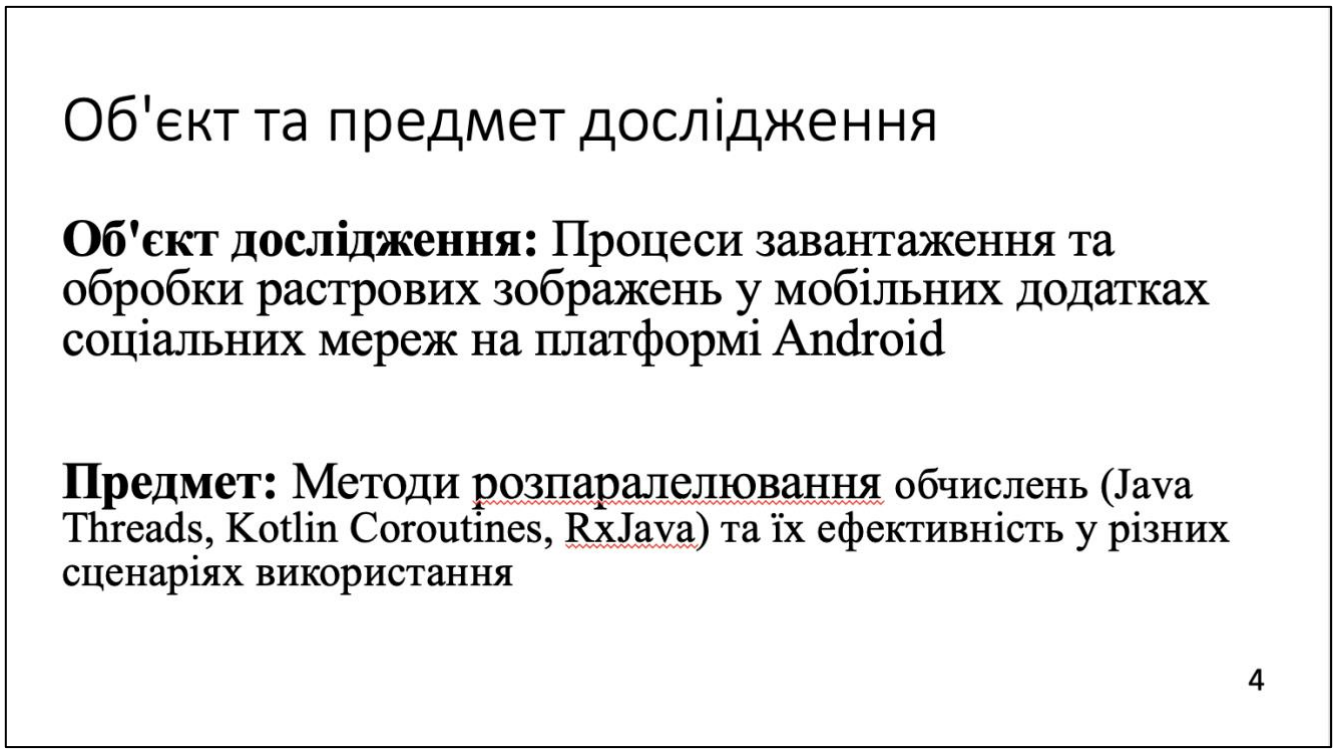


Рисунок Б.4 – Слайд 4 (рисунок створений самостійно)

Результати теоретичного дослідження

Характеристика	Threads	Kotlin Coroutines	RxJava
Використання	Складно	Проста	Середнє
Продуктивність	Низька	Висока	Середня
Споживання ресурсів	Високе	Низьке	Середнє
Енергоефективність	Низька	Висока	Середня

5

Рисунок Б.5 – Слайд 5 (рисунок створений самостійно)

Висновки з теоретичного дослідження

1. Значні прогалини в науковому обґрунтуванні вибору методів розпаралелювання для мобільних додатків.

2. Відсутня комплексна методика вибору оптимальних підходів саме для сценаріїв завантаження зображень у соціальних мережах

6

Рисунок Б.6 – Слайд 6 (рисунок створений самостійно)

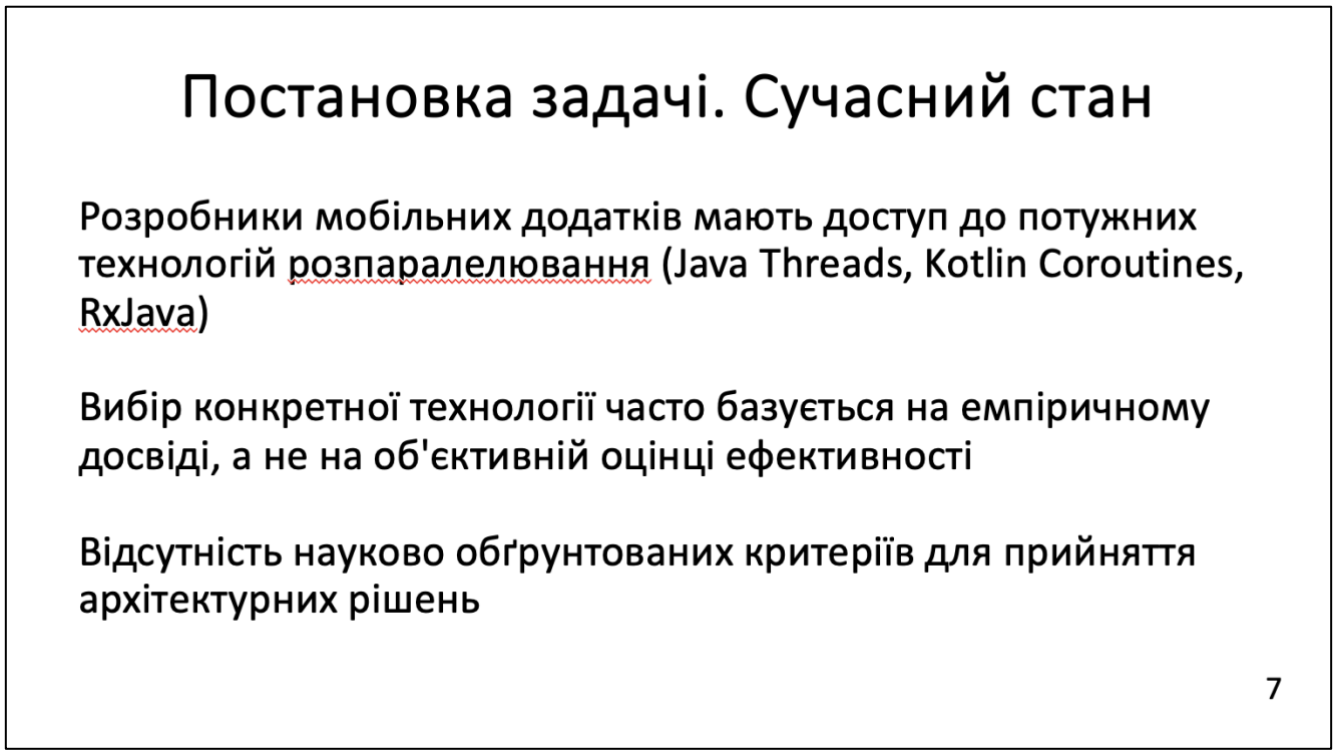


Рисунок Б.7 – Слайд 7 (рисунок створений самостійно)

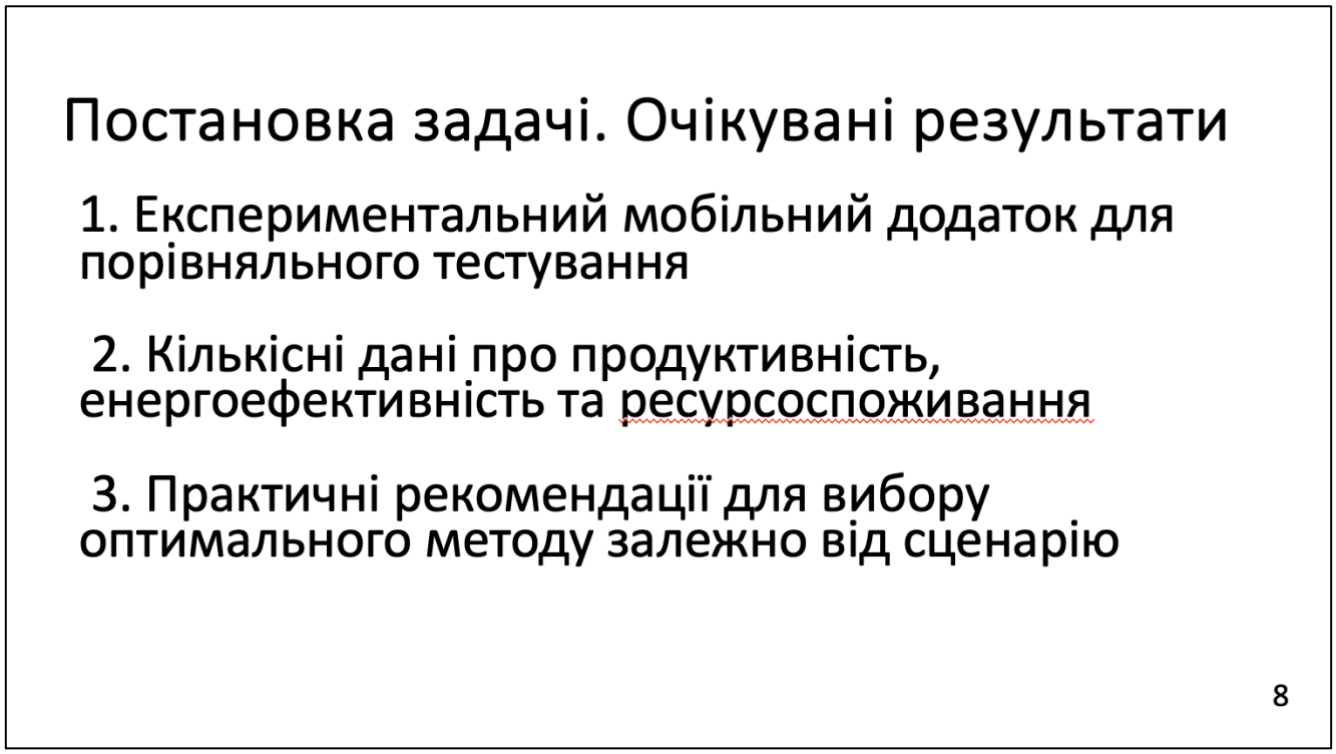


Рисунок Б.8 – Слайд 8 (рисунок створений самостійно)

Методика експерименту

Експериментальний аналіз продуктивності - практичне порівняння ефективності різних підходів до розпаралелювання

Порівняльне тестування алгоритмів розпаралелювання в ідентичних умовах

Профілювання ресурсоспоживання мобільних додатків для вимірювання CPU, пам'яті та енергії

9

Рисунок Б.9 – Слайд 9 (рисунок створений самостійно)

Інструментарій та технології

Програмне забезпечення:

Android Studio Ladybug 2024.2.2 + JDK 23.0.2

Kotlin 2.0.0, RxJava 3.1.8, Kotlin Coroutines 1.9.0

Picsum Photos API для тестових зображень

Тестове обладнання:

Motorola Moto g24 (Android 14, 8GB RAM, Helio G85)

Інструменти вимірювання:

Android Studio Profiler - моніторинг CPU, пам'яті, енергоспоживання

System.nanoTime() - точні вимірювання часу виконання операцій

10

Рисунок Б.10 – Слайд 10 (рисунок створений самостійно)

Експериментальні сценарії

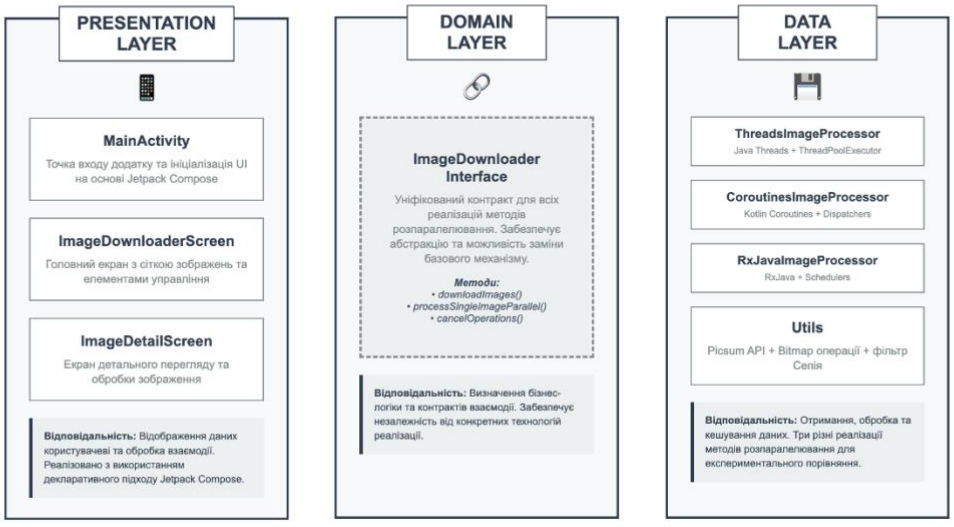
Сценарій 1: I/O-інтенсивне навантаження
Джерело: Picsum Photos API
Розміри: 150×150, 640×640, 1080×1080 пікселів
Формат: JPEG (8КБ - 200КБ)

Сценарій 2: CPU-інтенсивне навантаження
Операція: Піксельна обробка з математичними перетвореннями
Паралелізація: Розбиття на частини по кількості ядер процесора
Тестові розміри: 150×150, 640×640, 1080×1080 пікселів

11

Рисунок Б.11 – Слайд 11 (рисунок створений самостійно)

Архітектура система для проведення експериментального дослідження



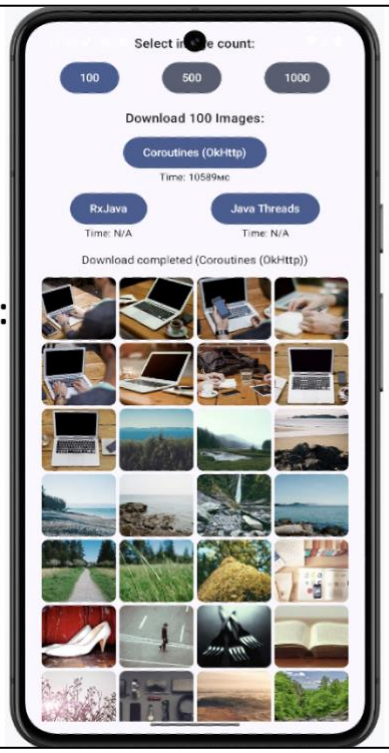
12

Рисунок Б.12 – Слайд 12 (рисунок створений самостійно)

Інтерфейс користувача

Головний екран

Вибір кількості зображень (100/500/1000)
Три кнопки методів розпаралелювання:
Coroutines - Kotlin Coroutines
RxJava - Реактивне програмування
Java Threads - Традиційні потоки
Прогрес-бар завантаження
Сітка зображень після завантаження
Відображення часу виконання



13

Рисунок Б.13 – Слайд 13 (рисунок створений самостійно)

Інтерфейс користувача

Екран деталей

Повноекранний перегляд зображення
Вибір розміру для обробки (Small/Medium/Large)
Кнопки застосування фільтру Сепія
Фіксація часу обробки

14

Рисунок Б.14 – Слайд 14 (рисунок створений самостійно)

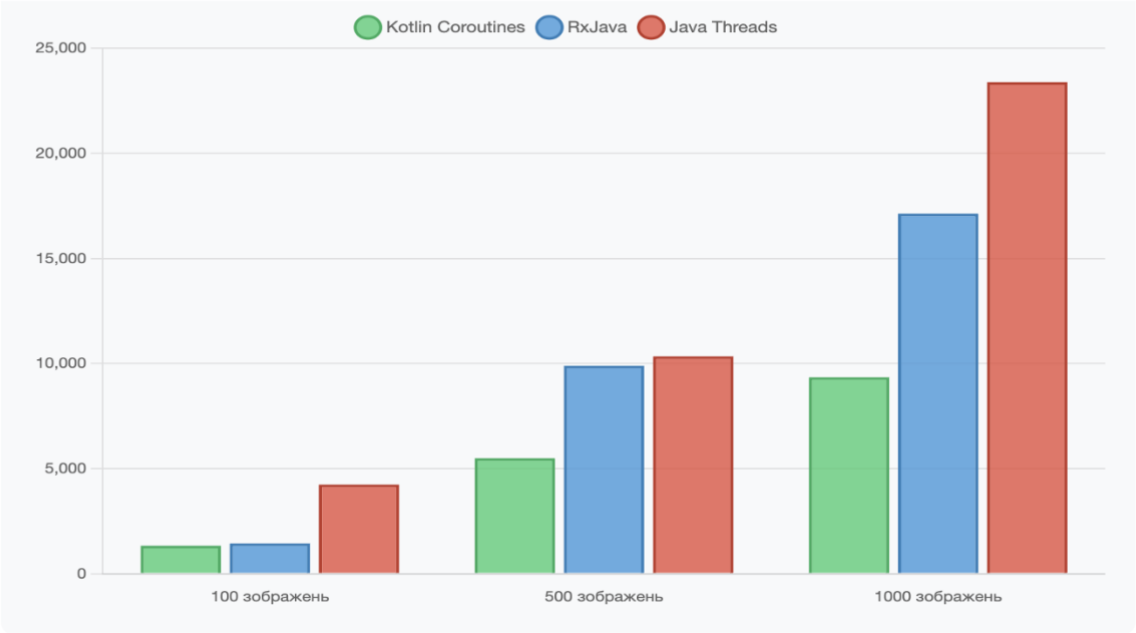
Процес тестування

Етап	Дія	Фіксація
Підготовка	Стандартизація стану пристрою	Wi-Fi, 80%+ батареї
Виконання	Тестування методу + навантаження	System.nanoTime()
Моніторинг	Android Studio Profiler	CPU%, енергія
Фіксація	Запис метрик в UI та логи	3 метрики
Повторення	5 разів для статистики	Усереднення

15

Рисунок Б.15 – Слайд 15 (рисунок створений самостійно)

Результати експерименту



16

Рисунок Б.16 – Слайд 16 (рисунок створений самостійно)

Аналіз отриманих результатів

Kotlin Coroutines демонструє найвищу ефективність ($p < 0.05$)

Зниження ресурсоспоживання в 2-2.5 рази

Рекомендовано для впровадження в Android-проектах

Метрика	Kotlin Coroutines	Java Threads	Поліпшення
Час виконання (мс)	1247 ± 89	3118 ± 156	**2.5×**
CPU навантаження (%)	32 ± 4	78 ± 7	**58%**
Енергоспоживання (мкА·год)	42.3 ± 3.1	105.7 ± 8.2	**60%**

17

Рисунок Б.17 – Слайд 17 (рисунок створений самостійно)

Практичні рекомендації

НОВІ ПРОЕКТИ

- **Kotlin Coroutines** як основний метод асинхронного програмування
- Архітектура **MVVM** з StateFlow/SharedFlow
- Використання **structured concurrency**
- Інтеграція з **Jetpack Compose**

ІСНУЮЧІ ПРОЕКТИ

- **Поетапна міграція** з Java Threads/AsyncTask
- **Гібридний підхід** на перехідному етапі
- Пріоритет **I/O-операцій** та мережевих запитів
- Рефакторинг **callback-based** коду

18

Рисунок Б.18 – Слайд 18 (рисунок створений самостійно)

КРИТЕРІЇ ВИБОРУ МЕТОДУ РОЗПАРАЛЕЛЮВАННЯ

Тип завдання	Рекомендований метод	Обґрунтування
I/O-операції, мережеві запити	**Kotlin Coroutines**	Найвища ефективність, зниження споживання на 60%
CPU-інтенсивні завдання	**Kotlin Coroutines**	Зниження навантаження на CPU на 58%
Реактивні потоки даних	**RxJava + Coroutines**	Комбінований підхід для складних трансформацій
Legacy системи	**Поступова міграція**	Мінімізація ризиків, контрольоване впровадження

19

Рисунок Б.19 – Слайд 19 (рисунок створений самостійно)

Галузевий вплив та перспективи розвитку

ГАЛУЗЕВА ЦІННІСТЬ

- Рекомендації для **70%+ світового ринку** мобільних пристроїв
- **Науково обґрунтований** вибір технологій для розробників
- Економія ресурсів для **мільйонів користувачів** соціальних мереж
- Підвищення **конкурентоспроможності** Android-додатків

ПЕРСПЕКТИВИ РОЗВИТКУ

- Тестування на **різних типах пристроїв** (бюджетні, флагманські)
- Дослідження впливу **версій Android** та API рівнів
- Порівняння з **новими технологіями** (KMP, Compose Multiplatform)
- Розширення на **інші типи додатків** та платформи

20

Рисунок Б.19 – Слайд 19 (рисунок створений самостійно)

ДОДАТОК В

Апробація результатів кваліфікаційної роботи

ΛΌΓΟΣ

Ricerche scientifiche e metodi della loro realizzazione: esperienza mondiale e realtà domestiche

SEZIONE 17.
INFORMATICA E INGEGNERIA DEL SOFTWARE

DOI 10.36074/logos-06.06.2025.053

OPTIMIZING MOBILE APPLICATION PERFORMANCE BY CHOOSING AN IMAGE PROCESSING PARALLELIZATION METHOD

Polina Nosova[1]
Scientific supervisor: Natalia Kravets[2]

1. higher education student at the Faculty of Computer Science
Kharkiv National University of Radio Electronics, UKRAINE
ORCID ID: 0009-0006-7522-3999

2. Ph.D, Associate Professor of Department of Software Engineering
Kharkiv National University of Radio Electronics, UKRAINE
ORCID ID: 0000-0002-6753-3333

Nowadays, mobile applications are becoming increasingly functional, and raster image processing is one of the most resource-intensive operations. Improper implementation of such operations can lead to application delays, accelerated battery drain, and device overheating. One of the most effective optimization methods remains the use of multithreading, which allows distributing the load across multiple execution threads.

The Android ecosystem presents unique challenges for image processing optimization due to the diversity of hardware configurations, ranging from budget devices with limited computational resources to flagship smartphones with multiple CPU cores and dedicated AI processing units. Modern Android devices typically feature multi-core processors, making parallel processing strategies essential for maximizing performance while maintaining energy efficiency[1].

The relevance of this research is further amplified by several contemporary technological trends. The rapid adoption of augmented reality (AR) and virtual reality (VR) applications demands real-time image processing capabilities that can handle complex visual computations without compromising user experience. Social media platforms increasingly rely on sophisticated image filters and AI-powered photo enhancement features that require efficient on-device processing to maintain user privacy and reduce server load. Additionally, the emergence of

All rights reserved | CC BY-SA 4.0

Experimental Design: The thread pool size for all methods was uniformly set to 4 threads, corresponding to the number of available CPU cores. A method for dividing images into fragments was implemented, where each fragment covers the full image width while receiving a unique vertical portion. Testing was conducted on images sized 800×600 and 1920×1080 pixels with 10-fold repetition to obtain statistically significant results.

Measurement Protocol: Android Profiler was used for metric collection, measuring operation execution time (ms), processor load (%), and energy consumption (μAh). Each test involved processing images with brightness adjustment of 250 units, with both single-threaded and multi-threaded implementations evaluated to assess parallelization benefits.

According to the experimental findings, the effectiveness of parallelization methods significantly depends on the size of the processed image. For small images (800×600 pixels), Kotlin Coroutines demonstrate the best execution time (240±8 ms with 4-thread processing), outperforming Java Threads (255±10 ms) and RxJava (285±12 ms). Meanwhile, Kotlin Coroutines also provide the lowest energy consumption (45.8 μAh versus 50.3 μAh for Java Threads and 58.2 μAh for RxJava).

For larger images (1920×1080 pixels), Kotlin Coroutines maintain their performance advantage, providing the highest speed (1120±45 ms), outperforming Java Threads (1180±55 ms) by 5.1% and RxJava (1350±65 ms) by 17.0%. However, for energy efficiency with large images, Java Threads demonstrate the best results (89.5 μAh versus 95.2 μAh for Kotlin Coroutines and 102.7 μAh for RxJava).

CPU load analysis showed that Kotlin Coroutines are characterized by the most uniform load distribution on the CPU when processing images of different sizes, indicating more efficient optimization of computational task distribution. The acceleration factor analysis revealed that all methods achieve similar parallelization efficiency, with acceleration factors of approximately 3.8x when moving from single-threaded to 4-threaded processing (Fig. 1).

Based on the obtained results, practical recommendations can be formulated for different application scenarios. Kotlin Coroutines demonstrate consistent performance advantages across both small and large images, making them suitable for applications requiring high processing speed and stable performance. Their uniform CPU load distribution indicates efficient resource management, particularly beneficial for complex multi-threaded processing scenarios.

Java Threads show the best energy efficiency for large image processing tasks, making them the optimal choice for applications with critical energy consumption requirements or when operating on devices with limited battery capacity. This advantage becomes particularly important for applications processing high-resolution images frequently.

All rights reserved | CC BY-SA 4.0

mobile machine learning frameworks like TensorFlow Lite and ML Kit has made on-device computer vision tasks more accessible, creating a growing need for optimized image preprocessing pipelines[2].

The objective of this research is to identify the most effective methods for parallelizing raster image processing in Android mobile applications through comprehensive comparative analysis of Kotlin Coroutines, Java Threads, and RxJava based on performance, energy consumption, and CPU load criteria[6]. This study aims to provide practical guidelines for mobile developers and contribute to the optimization of image processing workflows in mobile applications.

The research methodology was designed to ensure maximum objectivity and reproducibility of results. All tests were conducted under identical conditions using the same basic image processing algorithm, hardware and software configuration, standardized input data, and unified measurement system. This approach minimizes the influence of external factors and allows focusing exclusively on differences in the performance of various parallelization methods.

The experimental study was conducted using a brightness adjustment algorithm as the basic image processing operation. This choice was motivated by several factors: brightness adjustment is computationally intensive, requiring sequential processing of each pixel in the image (over two million operations for 1920×1080 resolution), each pixel can be processed independently making it ideal for parallelization evaluation, and it represents a common operation used in photo editors, computer vision systems, and social media applications.

Software Environment:

- Android Studio Ladybug Feature Drop 2024.2.2 Patch 1
- Java Development Kit (JDK) 23.0.2
- Kotlin 2.0.0, RxJava 3.1.8, Kotlin Coroutines 1.9.0

Hardware Configuration: Testing was performed on Android Studio Nexus 5X emulator (API Level 30, RAM: 2GB, CPU: x86) hosted on Apple M2 processor with 16GB RAM running macOS 15 Sequoia. Emulator parameters included cold boot before each test series, disabled Quick Boot and background processes, and identical graphics acceleration settings.

Implementation Approach: For each parallelization method, a corresponding wrapper class was developed that adapts the basic brightness adjustment algorithm to the specific approach. The Kotlin Coroutines implementation uses Dispatchers.Default context with parallelism limitation, creating separate coroutines for each image fragment through async functions. The RxJava approach implements reactive programming paradigm using Single objects and Schedulers.computation() thread pool. The Java Threads implementation involves direct thread creation and management with manual synchronization.

All rights reserved | CC BY-SA 4.0

6 giugno, 2025;
Bologna, Repubblica Italiana

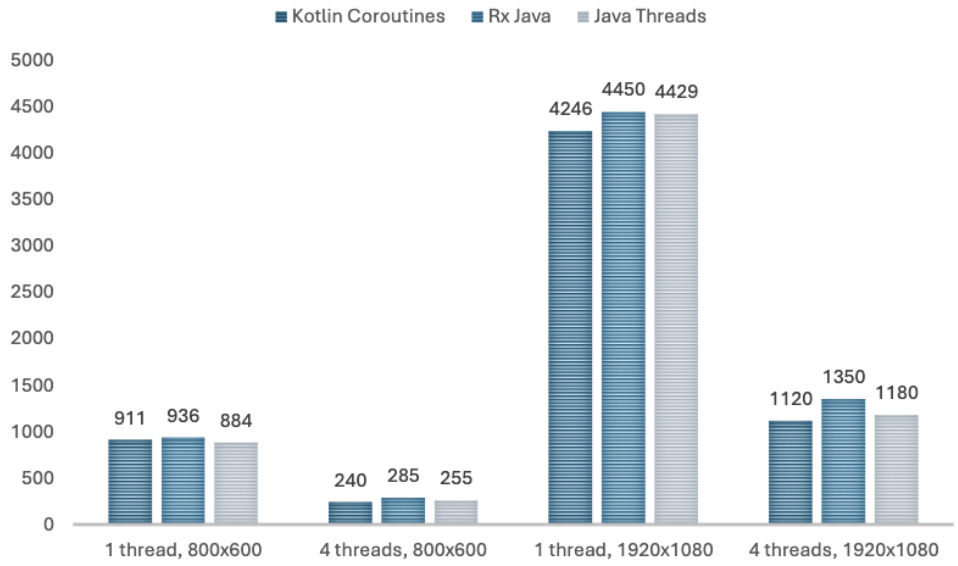


Fig. 1. **Comparative analysis of execution time for different parallelization methods across image sizess**

RxJava remains a viable compromise solution for developers who prioritize the declarative programming approach and require sophisticated data stream transformation mechanisms, despite showing the highest execution times in most test scenarios.

The research reveals that the choice of parallelization method should be based on specific application requirements: performance-critical applications benefit most from Kotlin Coroutines, energy-sensitive applications should consider Java Threads especially for large image processing, and applications requiring complex asynchronous data transformations may justify the performance trade-offs of RxJava for its programming paradigm advantages.

The obtained results have practical significance for mobile application optimization and can be used for further research in the field of parallel computing on mobile platforms, especially in the context of the growing popularity of machine learning and artificial intelligence technologies at the mobile device level.

REFERENCES:

[1] Vysotska, V., Lytvyn, V., Osypov, M., Semotyuk, O., Kovalchuk, V., & Keberle, N. (2024). Fast color images clustering for real-time computer vision and AI system. *COLINS*, (1), 161–177. https://doi.org/10.31110/COLINS/2024-1/012

All rights reserved | CC BY-SA 4.0

[2] Khudov, H., Ruban, I., Makoveichuk, O., Pevtsov, H., Khudov, V., Khizhnyak, I., Fryz, S., Podlipaiev, V., Polonskyi, Y., & Khudov, R. (2020). Development of methods for determining the contours of objects for a complex structured color image based on the ant colony optimization algorithm. *Physics and Engineering*, 1, 3–47. https://doi.org/10.21303/2461-4262.2020.001108
[3] Maddukuri, N. (2025). Workflow optimization for mobile computing. *International Journal of Science and Research Archive*, *14*(1), 340–346. https://doi.org/10.30574/ijsra.2025.14.1.0048
[4] Rodríguez, G. (2024). *Thriving in Android development using Kotlin: Use the newest features of the Android framework to develop production-grade apps*. Packt Publishing Ltd.
[5] Tomás Berjaga, A. (2022). *Microarchitectural-level simulator for parallel tile rendering on mobile GPUs* (PhD thesis). Technical University of Catalonia. https://upcommons.upc.edu/handle/2117/376125
[6] Zieliński, A. A. (2020). Comparative analysis of Kotlin coroutines with Java and Scala in parallel programming. *Journal of Computer Sciences Institute*, *16*, 241–246. https://doi.org/10.35784/jcsi.2002
[7] Grabowiec, M., Wiktor, S., & Smołka, J. (2024). Performance analysis of coroutines and other concurrency techniques in Kotlin language for I/O operations. *Journal of Computer Sciences Institute*, *33*, 306–312. https://doi.org/10.35784/jcsi.6353
[8] Göransson, A. (2014). *Efficient Android threading*. O'Reilly Media.
[9] Rua, R., & Saraiva, J. (2024). A large-scale empirical study on mobile performance: Energy, run-time and memory. *Empirical Software Engineering*, *29*(31), 1–52. https://doi.org/10.1007/s10664-023-10391-y
[10] Thabet, R., Mahmoudi, R., & Bedoui, M. H. (2015). Image processing on mobile devices: An overview. In *International Conference on Image Processing and Applications Systems* (pp. 1–6). https://doi.org/10.1109/IPAS.2014.7043267

All rights reserved | CC BY-SA 4.0

CERTIFICATO
DI PARTECIPAZIONE

Polina Nosova

ha partecipato alla VII Conferenza scientifica e pratica internazionale

«Ricerche scientifiche e metodi della loro realizzazione: esperienza mondiale e realtà domestiche»

e pubblicato articoli scientifici

OPTIMIZING MOBILE APPLICATION PERFORMANCE BY CHOOSING AN IMAGE PROCESSING PARALLELIZATION METHOD

Bologna
Repubblica Italiana

6 giugno
2025

0.6 ECTS credits (18 hours)
Recommended by the Academic Council of the «Institute of Scientific and Technical Integration and Cooperation».
Protocol № 22 from June 5th, 2025.

Atti della Conferenza scientifica e pratica internazionale sono pubblicati nella Raccolta di articoli scientifici AGITOS.

Manager delle Piattaforme
In Viaggio Con Il Lavoro
DAVIDE SQUARCHALUPI

Capo della Piattaforma
Presidente del comitato
MIRIAM GOLDENBLAT

ISPC № 060625-064

- La conferenza è inclusa nel **ACADEMIC RESEARCH INDEX**.
- La conferenza certificata secondo lo standard SCC 2000. **Euro Science Certificato** № 22903 del 03.04.2025
- La conferenza registrata presso l'«Istituto ucraino di competenza e informazione scientifica e tecnica» **UKRISTEI** Certificato № 218 del 12.06.2024
- libretto rilasciato secondo le norme ISO 2108:2005, ISO 1086:1981 e ISO 7220:1985.
- articolo pubblicato indicizzato in **CrossRef, Google Scholar, OUCI, OpenAIRE, WorldCat, Semantic Scholar, Mendeley; Scilit; PubPeer, Lens.org, Scite_** ecc.

DOI 10.36074/logos-06.06.2025

ДОДАТОК Г

Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 20

Експертний висновок результатів перевірки кваліфікаційної роботи

студент (посада) | програмної інженерії (кафедра) | ІПЗм-23-4□ (група)

Поліна НОСОВА
(прізвище, ім'я, по батькові)

Зауваження

Пункт ДСТУ 3008-2015	Зміст пункту	Сторінка кваліфікаційної роботи
1	2	3
	7.1 Загальні положення	
	7.3 Нумерація сторінок звіту	
	7.5 Рисунки	
	7.6 Таблиці	
7.6.9	Якщо рядки або колонки таблиці виходять за межі формату сторінки, таблицю поділяють на частини, розміщуючи одну частину під іншою або поруч, чи переносять частину таблиці на наступну сторінку. У кожній частині таблиці повторюють її головку та боковик. У разі поділу таблиці на частини дозволено її головку чи боковик заміняти відповідно номерами колонок або рядків, нумеруючи їх арабськими цифрами в першій частині таблиці. Слово «Таблиця » подають лише один раз над першою частиною таблиці. Над іншими частинами таблиці з абзацного відступу друкують «Продовження таблиці » або «Кінець таблиці ____ » без повторення її назви.	38
	7.7 Переліки	
	7.8 Примітки	
	7.9 Виноски	
	7.10 Формули та рівняння	
7.10.6	Пояснення познак, які входять до формули чи рівняння, треба подавати безпосередньо під формулою або рівнянням у тій послідовності, у якій їх наведено у формулі або рівнянні. Пояснення познак треба подавати без абзацного відступу з нового рядка, починаючи зі слова «де» без двокрапки. Познаки, яким встановлюють визначення чи пояснення, рекомендовано ви-рівнювати у вертикальному напрямку.	47
	7.11 Посилання	
	7.13 Список авторів	
	7.14 Скорочення та умовні познаки	
	7.15 Додатки	

Експерт _____ (підпис) Вадим НЕЧВОЛОД (прізвище, ініціали)

15.06.2025

ДОДАТОК Д

Метод обробки зображення з RxJava

```
     override fun processSingleImageParallel(
     originalBitmap: Bitmap,
        targetWidth: Int,
        targetHeight: Int,
        onSuccess: (Bitmap, Long) -> Unit,
        onError: (Throwable) -> Unit
    ) {
        val startTime = System.nanoTime()

        val disposable = Single.fromCallable {
            val resizedBitmap = Utils.resizeBitmap(originalBitmap,
targetWidth, targetHeight)
            val originalWidth = originalBitmap.width
            val originalHeight = originalBitmap.height
            val parts = Utils.splitBitmap(originalBitmap,
NUM_PROCESSING_PARTS)

            Log.d("RxJavaImageProcessor", "Starting processing of
${parts.size} parts")

            val processedParts = parts.withIndex().map { (partIndex,
partData) ->
                val (partBitmap, originalX) = partData
                Single.fromCallable {
                    Log.d("RxJavaImageProcessor", "Processing part
$partIndex")
                    try {
                        val processedPart =
Utils.processBitmapPartBlocking(partBitmap)
                        Pair(processedPart, originalX)
                    } catch (e: Exception) {
                        Log.e(
                            "RxJavaImageProcessor",
                            "Error processing part $partIndex:
${e.message}",
                            e
                        )
                        throw e
                    }
                }.subscribeOn(Schedulers.computation())
            }

            val results = Single.zip(processedParts) { resultsArray ->
                resultsArray.map { it as Pair<Bitmap, Int> }
            }.blockingGet()

            val finalBitmap = Utils.combineBitmaps(originalWidth,
originalHeight, results)
            val timeNanos = System.nanoTime() - startTime
```

```
            Log.d("RxJavaImageProcessor", "Successfully processed image")
            Pair(timeNanos, finalBitmap)
        }
            .subscribeOn(Schedulers.computation())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({ (timeNanos, finalBitmap) ->
                onSuccess(finalBitmap, timeNanos / 1_000_000)
            }, { error ->
                Log.e(
                    "RxJavaImageProcessor",
                    "Error in processSingleImageParallel:
${error.message}",
                    error
                )
                onError(error)
            })

        disposables.add(disposable)
    }

    override fun cancelOperations() {
        disposables.clear()
    }
}
```

ДОДАТОК Е

Клас для обробки зображення з Java Thread

```
class ThreadsImageProcessor : ImageDownloader {
    private val executorService: ExecutorService =

Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors())

    private val uiHandler = Handler(Looper.getMainLooper())

    private val activeFutures = mutableListOf<Future<*>>()

    private val NUM_PROCESSING_PARTS =
Runtime.getRuntime().availableProcessors()

    override fun downloadImages(
        imageUrls: List<String>,
        onProgress: (Int, Bitmap?) -> Unit,
        onComplete: (Long) -> Unit,
        onError: (Throwable) -> Unit
    ) {
        val startTime = System.nanoTime()
        val latch = CountDownLatch(imageUrls.size)

        imageUrls.forEachIndexed { index, url ->
            val future = executorService.submit {
                try {
                    val bitmap = Utils.downloadImageBlocking(url)
                    uiHandler.post {
                        onProgress(index, bitmap)
                    }
                } catch (e: Exception) {
                    uiHandler.post {
                        onProgress(index, null)
                        onError(e)
                    }
                } finally {
                    latch.countDown()
                }
            }
            activeFutures.add(future)
        }

        val completionFuture = executorService.submit {
            try {
                latch.await()
                val endTime = System.nanoTime()
                uiHandler.post {
                    onComplete((endTime - startTime) / 1_000_000)
                }
            } catch (e: InterruptedException) {
                uiHandler.post {
                    onError(e)
```

```
                }
                Thread.currentThread().interrupt()
            }
        }
        activeFutures.add(completionFuture)
    }

    override fun processSingleImageParallel(
        originalBitmap: Bitmap,
        targetWidth: Int,
        targetHeight: Int,
        onSuccess: (Bitmap, Long) -> Unit,
        onError: (Throwable) -> Unit
    ) {
        val future = executorService.submit {
            try {
                val finalBitmap: Bitmap
                val timeMs: Long

                timeMs = measureNanoTime {
                    val resizedBitmap =
                        Utils.resizeBitmap(originalBitmap, targetWidth,
targetHeight)
                    val originalWidth = originalBitmap.width
                    val originalHeight = originalBitmap.height
                    val parts = Utils.splitBitmap(originalBitmap,
NUM_PROCESSING_PARTS)

                    val processedPartsWithCoords =
mutableListOf<Pair<Bitmap, Int>>()
                    val latch = CountDownLatch(parts.size)

                    parts.forEach { (partBitmap, originalX) ->
                        val partFuture = executorService.submit {
                            try {
                                val processedPart =
Utils.processBitmapPartBlocking(partBitmap)
                                synchronized(processedPartsWithCoords) {

processedPartsWithCoords.add(Pair(processedPart, originalX))
                                }
                            } catch (e: Exception) {
                                e.printStackTrace()
                            } finally {
                                latch.countDown()
                            }
                        }
                        activeFutures.add(partFuture)
                    }

                    latch.await()

                    finalBitmap = Utils.combineBitmaps(
                        originalWidth,
                        originalHeight,
                        processedPartsWithCoords
```

```
                    )
                }

                uiHandler.post {
                    onSuccess(finalBitmap, timeMs / 1_000_000)
                }
            } catch (e: Exception) {
                uiHandler.post {
                    onError(e)
                }
            }
        }
        activeFutures.add(future)
    }

    override fun cancelOperations() {
        for (future in activeFutures) {
            future.cancel(true)
        }
        activeFutures.clear()
    }
}
```