Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів розпаралелювання процесів завантаження та обробки растрових зображень у мобільному додатку для соціальних мереж під Android
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-4

Поліна НОСОВА
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Наталя КРАВЕЦЬ
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Носовій Поліні Валеріївні
(прізвище, ім'я, по батькові)

1. Тема роботи«Дослідження методів розпаралелювання процесів завантаження та обробки растрових зображень у мобільному додатку для соціальних мереж під Android»
Затверджена наказом по університету від15.04. 2025р. № 290 Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії20.06.2025
3. Вихідні дані до роботи опис досліджуваних методів розпаралелювання (Java Threads, Kotlin Coroutines, RxJava), вимоги до розробки експериментального мобільного додатку для проведення досліджень за обраною предметною областю, мова програмування Kotlin, технології Android SDK, бібліотеки RxJava, Kotlin Coroutines, середовище розробки Android Studio 2024.2.2
4. Перелік питань, що потрібно опрацювати в роботі
аналіз та порівняння існуючих методів розпаралелювання в Android (Java Threads, Kotlin Coroutines, RxJava), вибір підходящих технологій для експериментального дослідження, проектування архітектури тестового мобільного додатку для проведення порівняльних досліджень продуктивності, розробка програмних рішень для завантаження та обробки растрових зображень з використанням різних методів розпаралелювання, створення методології експериментального дослідження з визначенням метрик оцінювання, проведення експериментів з вимірювання часу виконання операцій, навантаження на процесор та енергоспоживання, аналіз отриманих результатів та формулювання практичних рекомендацій щодо вибору оптимальних методів розпаралелювання

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Отримання завдання	16.04.2025	виконано
2.	Аналіз предметної галузі і постановка задачі	17.04.2025	виконано
3.	Назва за розділами теоретичного і практичного дослідження	20.04.2025	виконано
4.	Назва за розділами теоретичного і практичного дослідження	22.04.2025	виконано
5.	Підготовка до апробації результатів дослідження. Публікація матеріалів	04.05.2025	виконано
6.	Назва за розділами теоретичного і практичного дослідження	10.05.2025	виконано
7.	Підготовка пояснювальної записки	18.05.2025	виконано
8.	Підготовка презентації та доповіді	05.06.2025	виконано
9.	Перевірка на плагіат	14.06.2025	виконано
10.	Нормоконтроль	15.06.2025	виконано
11.	Рецензування	17.06.2025	виконано
12.	Попередній захист	17.06.2025	виконано
13.	Занесення диплома в електронний архів	18.06.2025	виконано
14.	Допуск до захисту у зав. кафедри	18.06.2025	виконано

Дата видачі завдання 16.04.2025р

Здобувач (ка) _____ Поліна НОСОВА_____
(підпис)

Керівник роботи _____ доц. Наталя КРАВЕЦЬ
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 98 с.,11 рис., 6 табл., 21 джерело.

АСИНХРОННИЙ КОД, ОБРОБКА ЗОБРАЖЕНЬ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ,JAVATHREADS, KOTLIN COROUTINES, ОС ANDROID, RXJAVA.

Об'єктом дослідження є процеси завантаження та обробки растрових зображень у мобільних додатках соціальних мереж на платформі Android.

Метою роботи є розробка науково обґрунтованих рекомендацій щодо вибору оптимальних методів розпаралелювання для завантаження зображень у мобільних додатках соціальних мереж на основі комплексного експериментального аналізу продуктивності, енергоефективності та ресурсоспоживання Java Threads, Kotlin Coroutines та RxJava.

Методами дослідження є експериментальний аналіз продуктивності, порівняльне тестування алгоритмів розпаралелювання, профілювання ресурсоспоживання мобільних додатків, статистичний аналіз результатів експериментів.

У результаті кваліфікаційної роботи було розроблено експериментальний мобільний додаток для Android, що реалізує завантаження та обробку зображень з використанням трьох різних методів розпаралелювання.

ASYNCHRONOUS CODE, IMAGE PROCESSING, PARALLEL COMPUTATIONS, JAVA THREADS, KOTLIN COROUTINES, ANDROID OS, RX JAVA.

The object of research is the processes of loading and processing raster images in mobile social network applications on the Android platform.

The purpose of the work is to develop scientifically grounded recommendations for choosing optimal parallelization methods for image loading in mobile social network

applications based on comprehensive experimental analysis of performance, energy efficiency and resource consumption of Java Threads, Kotlin Coroutines and RxJava.

The research methods are experimental performance analysis, comparative testing of parallelization algorithms, profiling of mobile application resource consumption, statistical analysis of experimental results.

As a result of the qualification work, an experimental Android mobile application was developed that implements image loading and processing using three different parallelization methods.

Завідувачу кафедри

ПІ_____

(скорочена назва кафедри)

проф. Кирилу СМЕЛЯКОВУ

(вчене звання, сласне ім'я, прізвище)

ЗАЯВА

щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Носова Поліна Валеріївна_____

(прізвище, ім'я, по батькові)

здобувач вищої освіти на другому (магістерському) рівні вищої освітиак адемічної групи ІПЗм-23-4

кафедра_____ програмної інженерії _____,

(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему

Дослідження методів розпаралелювання процесів завантаження та обробки растрових зображень у мобільному додатку для соціальних мереж під Android.

(назва роботи)

що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата Підпис

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ADB – Android Debug Bridge

API – Application Programming Interface

ART – Android Runtime

CPU – Central Processing Unit

GC – Garbage Collection

GPU – Graphics Processing Unit

HTTP – HyperText Transfer Protocol

I/O – Input/Output

JSON – JavaScript Object Notation

JVM – Java Virtual Machine

MVP – Model-View-Presenter

MVVM – Model-View-ViewModel

OOM – Out Of Memory

RAM – Random Access Memory

SDK – Software Development Kit

UI – User Interface

URL – Uniform Resource Locator

XML – eXtensible Markup Language

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 неефективною реалізацією операцій завантаження та обробки зображень. Дослідження показують, що неоптимізовані алгоритми завантаження можуть збільшувати споживання енергії на 25-40% та призводити до блокування інтерфейсу користувача [3, 4].Завантаження зображень у соціальних мережах характеризується специфічними особливостями: високою інтенсивністю (до 100 зображень за сесію), різноманітністю форматів та розмірів (від мініатюр 150×150 до повнорозмірних 1080×1080), динамічністю контенту та необхідністю забезпечення плавного прокручування без затримок.

Одним з найефективніших підходів до вирішення проблем продуктивності є використання методів розпаралелювання, які дозволяють розподілити навантаження між кількома потоками виконання та значно підвищити швидкодію програми. Підходи до розпаралелювання в Android еволюціонували від AsyncTask та базових потоків Java до сучасних рішень, таких як RxJava та Kotlin Coroutines [5]. Проте розробники досі стикаються з проблемою вибору оптимального методу для конкретних сценаріїв, оскільки різні підходи демонструють відмінну ефективність залежно від типу операцій та характеристик пристрою.

Актуальність теми зумовлена зростанням популярності мобільних додатків соціальних мереж та підвищенням вимог користувачів до швидкодії інтерфейсу.Адже неефективна реалізація завантаження зображень може призводити до збільшення споживання енергії на 25-40% та блокування користувацького інтерфейсу [3, 4]. Відсутність комплексної методології вибору оптимальних методів розпаралелювання змушує розробників покладатися на емпіричний підхід, що часто призводить до неоптимальних архітектурних рішень.

Метою роботи є розробка науково обґрунтованих рекомендацій щодо вибору оптимальних методів розпаралелювання для завантаження зображень у мобільних додатках соціальних мереж на основі комплексного експериментального аналізу продуктивності, енергоефективності та ресурсоспоживання Java Threads, Kotlin Coroutines та RxJava.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- провести теоретичний аналіз принципів роботи та архітектурних особливостей Java Threads, Kotlin Coroutines та RxJava в контексті завантаження зображень у мобільних додатках;
- розробити методику експериментального дослідження для об'єктивного порівняння ефективності різних підходів до розпаралелювання з визначенням критеріїв оцінки та метрик вимірювання;
- створити експериментальний мобільний додаток та провести серію експериментів для вимірювання продуктивності, енергоспоживання та використання ресурсів різних підходів в умовах реального використання;
- проаналізувати отримані результати та виявити залежності ефективності різних методів від характеристик завдань та технічних параметрів пристроїв;
- розробити практичні рекомендації щодо вибору оптимальних методів розпаралелювання для різних сценаріїв завантаження зображень у мобільних додатках соціальних мереж.

Для вирішення поставлених завдань потрібно провести теоретичний аналіз наукової літератури та технічної документації за темою методів розпаралелення в мобільній розробці, провести експеримент для порівняння ефективності різних підходів до розпаралелювання, виміряти метрики продуктивності за допомогою профілювання додатку, провести аналіз отриманих результатів.

Об'єктом дослідження є процеси завантаження та обробки растрових зображень у мобільних додатках соціальних мереж на платформі Android.

Предметом дослідження є методи розпаралелювання обчислень для оптимізації завантаження зображень, зокрема Java Threads, Kotlin Coroutines, RxJava та їх ефективність у різних сценаріях використання.

Новизна дослідження полягає у проведенні порівняльного дослідження трьох методів розпаралелювання (Java Threads, Kotlin Coroutines, RxJava) саме для завантаження зображень у мобільних додатках з урахуванням специфіки соціальних мереж. Удосконалено методологію експериментального дослідження продуктивності методів розпаралелювання в мобільному середовищі через введення комплексних метрик оцінювання (час виконання, навантаження на процесор, енергоефективність). Отримано нові кількісні дані про масштабованість різних підходів та встановлено критерії вибору оптимальних технологій залежно від характеристик завдань.

Результати дослідження можуть бути використані розробниками мобільних додатків для прийняття обґрунтованих архітектурних рішень щодо вибору методів розпаралелювання. Розроблені рекомендації дозволяють оптимізувати продуктивність та енергоефективність мобільних додатків для соціальних мереж. Методика дослідження може застосовуватися для порівняльного аналізу інших технологій паралельного програмування в мобільних системах.

Практична застосовність розроблених рекомендацій передбачає їх формулювання у вигляді, придатному для використання в реальних проектах мобільної розробки з конкретними критеріями вибору методів, що дозволить розробникам приймати обґрунтовані архітектурні рішення. Комплексність аналізу вимагає врахування всіх ключових аспектів ефективності, включаючи продуктивність (час виконання, пропускна здатність), споживання ресурсів (CPU, пам'ять, мережа) та енергоефективність, що забезпечить всебічну оцінку досліджуваних методів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз сучасного стану мобільної розробки

Мобільна індустрія переживає період безпрецедентного зростання та технологічного розвитку.Платформа Android займає домінуючу позицію на світовому ринку мобільних операційних систем із часткою понад 70%[6]. Ця перевага обумовлена відкритою архітектурою системи, широкою екосистемою розробницьких інструментів та можливістю розгортання додатків на різноманітних пристроях від різних виробників. Однак саме ця різноманітність створює додаткові виклики для розробників щодо забезпечення стабільної роботи додатків на пристроях з різними технічними характеристиками.

Сучасні мобільні додатки характеризуються високою складністю архітектури та інтенсивним використанням візуального контенту. Особливо це стосується додатків соціальних мереж, які стали невід'ємною частиною повсякденного життя мільярдів користувачів. Платформи, такі як Instagram, Facebook, TikTok, Snapchat щодня обробляють великий об’єм візуального контенту, забезпечуючи безперервний потік зображень, відео та мультимедійних матеріалів для своїх користувачів.Користувачі соціальних мереж переглядають значну кількість зображень за короткий проміжок часу, що створює високі вимоги до швидкості завантаження та відображення візуального контенту.

Технічні характеристики сучасних мобільних пристроїв значно покращилися порівняно з попередніми поколіннями. Флагманські смартфони 2024 року оснащені багатоядерними процесорами з частотою понад 3 ГГц, обсягом оперативної пам'яті від 8 до 16 ГБ та високопродуктивними графічними процесорами. Однак навіть при таких технічних можливостях, неоптимізовані додатки можуть демонструвати проблеми з продуктивністю, особливо при обробці великих обсягів візуального контенту.

Ця суперечність між зростаючими технічними можливостями пристроїв та постійними проблемами продуктивності вказує на ключову проблему сучасної

мобільної розробки: недостатньо ефективне використання доступних ресурсів через неоптимальні алгоритми та підходи до програмування. Особливо гостро це проявляється в операціях завантаження та обробки зображень, які є критично важливими для додатків соціальних мереж. Аналіз архітектури Instagram демонструє широке використання методів оптимізації продуктивності, включаючи горизонтальне масштабування через додавання серверів, використання балансувальників навантаження для розподілу трафіку, та оптимізацію критичних функцій через Cython та C/C++ для зменшення використання CPU [7]. Успішні рішення активно використовують методи розпаралелювання для оптимізації роботи з візуальним контентом. Це включає асинхронне завантаження зображень, багаторівневе кешування, інтелектуальне передзавантаження та адаптивну якість контенту[7, 8]. Проте вибір конкретних технологій розпаралелювання часто базується на емпіричному досвіді команд розробки, а не на науково обґрунтованих критеріях ефективності.

Енергоефективність стала одним з найважливіших факторів якості мобільних додатків. Неоптимізовані операції з зображеннями можуть суттєво впливати на тривалість автономної роботи пристрою, що критично важливо для користувачів. Це створює додатковий виклик для розробників: необхідно забезпечити високу продуктивність, зберігаючи при цьому енергоефективність рішення.

Таким чином, сучасний стан мобільної розробки характеризується парадоксом: при наявності потужних технічних засобів та інструментів розробки, оптимізація завантаження зображень залишається складною технічною задачею, що вимагає глибокого розуміння принципів розпаралелювання та їх правильного застосування в специфічних умовах мобільного середовища.

1.2 Проблеми продуктивності при завантаженні та обробці зображень

Завантаження та обробка зображень у мобільних додатках представляє собою складний технічний процес, який включає численні етапи: мережеве завантаження, декодування, масштабування, кешування та відображення. Кожен з цих етапів

може стати вузьким місцем продуктивності, особливо при обробці великих обсягів візуального контенту в додатках соціальних мереж.

Одним з найбільш критичних аспектів є мережеве завантаження зображень. Сучасні додатки соціальних мереж можуть завантажувати десятки зображень одночасно, що створює значне навантаження на мережеву підсистему пристрою. При неефективній реалізації це може призводити до блокування основного потоку додатку, затримок у відгуку інтерфейсу користувача та негативного впливу на загальну продуктивність системи.

Дослідження показують, що неоптимізовані алгоритми завантаження зображень можутьсуттєво впливати на тривалість автономної роботи. Це пов'язано з неефективним використанням мережевих ресурсів, надмірною активністю процесора при декодуванні зображень та частими операціями читання/запису в пам'ять пристрою. Для користувачів це означає швидше розрядження батареї та погіршення загального досвіду використання пристрою.

Особливо гострою є проблема блокування інтерфейсу користувача під час обробки зображень. Android використовує однопоточну модель для роботи з UI, де всі операції інтерфейсу виконуються в головному потоці (UI thread). Якщо операції завантаження або обробки зображень виконуються безпосередньо в цьому потоці, це призводить до "заморожування" інтерфейсу, що сприймається користувачами як критична проблема продуктивності.

Обмеження ресурсів мобільних пристроїв включають наступні критичні фактори, що впливають на продуктивність обробки зображень:

Обмеження оперативної пам'яті. Навіть сучасні смартфони мають обмежений обсяг доступної оперативної пам'яті для окремих додатків. Android автоматично завершує роботу додатків, які перевищують встановлені ліміти пам'яті. Зображення, особливо високої роздільності, можуть займати значні обсяги пам'яті - наприклад, зображення розміром 1920×1080 пікселів у форматі RGBA займає близько 8 МБ оперативної пам'яті.

Обмеження процесорної потужності. Незважаючи на покращення технічних характеристик сучасних мобільних процесорів, декодування та обробка зображень залишаються ресурсоємними операціями. Особливо це стосується операцій масштабування, застосування фільтрів та перетворення форматів зображень.

Обмеження пропускної здатності мережі. Мобільні пристрої часто використовуються в умовах обмеженої або нестабільної мережевої пропускної здатності. Це вимагає від додатків інтелектуального управління завантаженням зображень, включаючи адаптивну якість, прогресивне завантаження та ефективне кешування.

Термальні обмеження. Інтенсивна обробка зображень може призводити до перегріву пристрою, що автоматично знижує частоту роботи процесора для захисту від пошкоджень. Це створює додаткові виклики для підтримання стабільної продуктивності протягом тривалих сесій використання додатку.

Специфіка завантаження зображень у додатках соціальних мереж включає кілька унікальних характеристик, які ускладнюють оптимізацію продуктивності:

Висока інтенсивність завантаження. Користувачі можуть переглядати до 100 зображень за одну сесію, причому кожне зображення може мати різний розмір, формат та вимоги до обробки.

Різноманітність форматів та розмірів. Сучасні соціальні мережі підтримують широкий спектр форматів зображень (JPEG, PNG, WebP, AVIF) та розмірів - від мініатюр 150×150 пікселів до повнорозмірних зображень 4K роздільності.

Динамічність контенту. На відміну від статичних галерей, контент у соціальних мережах постійно оновлюється, що ускладнює ефективне кешування та прогнозування завантажень.

Вимоги до плавного прокручування. Користувачі очікують миттєвого відгуку при прокручуванні стрічки новин, що вимагає передзавантаження зображень та мінімізації затримок при їх відображенні.Аналіз проблем продуктивності при завантаженні зображень у мобільних додатках демонструє необхідність використання спеціалізованих технологічних рішень для їх

вирішення. Теорія паралельних обчислень встановлює, що ефективне розпаралелювання операцій введення/виведення є ключовим фактором оптимізації продуктивності системних додатків [9].

Розвиток мобільних платформ супроводжувався еволюцією підходів до асинхронного програмування, що відображає зростаючі вимоги до продуктивності мобільних додатків та покращення технічних можливостей пристроїв. Таким чином, застосування концепцій паралельних обчислень у мобільних системах вимагає детального аналізу специфічних особливостей та обмежень таких платформ [10].

1.3 Методи розпаралелювання в ОС Android

Методи розпаралелювання завантаження зображень в Android зазнали значної еволюції протягом останніх років. На ранніх етапах розвитку платформи розробники використовували примітивні підходи, такі як AsyncTask та ручне управління потоками через класи Thread та Handler [9].

AsyncTask, який був основним інструментом для асинхронних операцій до Android API 30, мав значні обмеження. Він використовував невеликий пул потоків (зазвичай 5-10 потоків), що призводило до блокування при одночасному завантаженні великої кількості зображень. Крім того, AsyncTask був схильний до витоків пам'яті через неправильне управління життєвим циклом та відсутність автоматичного скасування операцій [10].Обмеження AsyncTask змусили розробників шукати альтернативні рішення з більшим ступенем контролю над виконанням асинхронних операцій. Одним з таких підходів стало повернення до використання базових механізмів багатопоточності Java, які надавали необхідну гнучкість для реалізації складних сценаріїв завантаження зображень, але вимагали глибшого розуміння принципів паралельного програмування.

Java Threads представляють класичний підхід до багатопоточного програмування, заснований на створенні окремих потоків виконання, які працюють паралельно з основним потоком програми. У контексті Android-розробки потоки

залишаються фундаментальним механізмом для виконання фонових операцій, особливо коли необхідний точний контроль над ресурсами або інтеграція із застарілим кодом.

Архітектура потоків базується на взаємно однозначному відповіденні між Java Thread та нативним потоком операційної системи. Кожен потік має власний стек виконання розміром зазвичай 1-2 МБ, власний лічильник команд та регістри процесора (див.рис 1.1). Це забезпечує повну ізоляцію між потоками, але створює значні накладні витрати на пам'ять та переключення контексту.

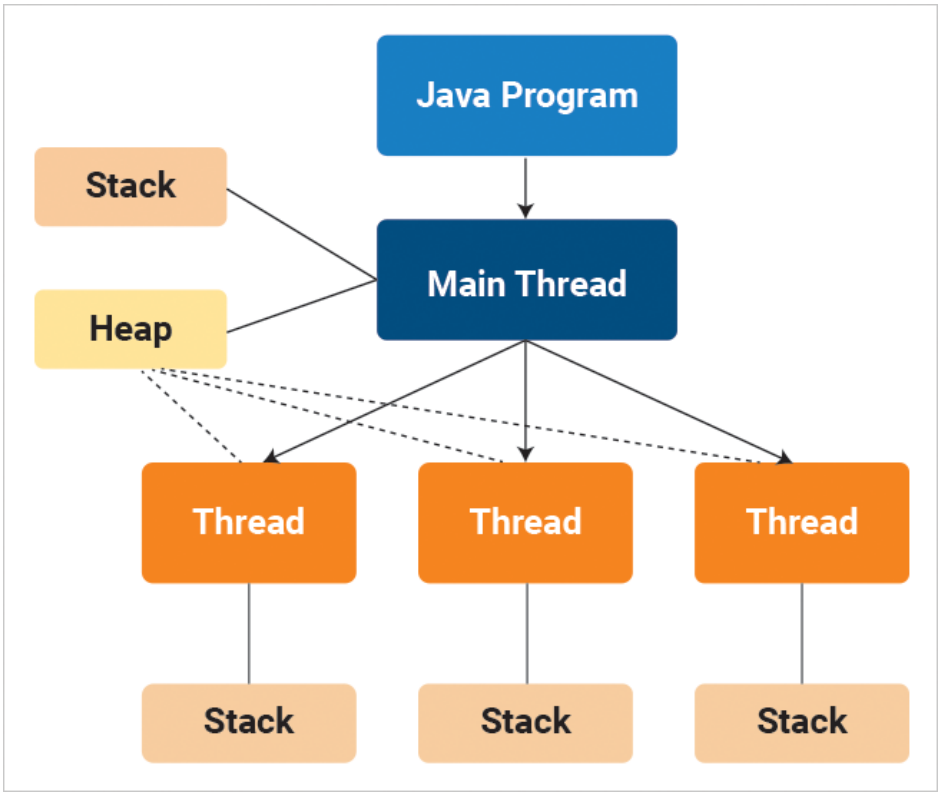

Рисунок 1.1 –Схема роботи Java Thread (за даними [11])

Створення та управління потоками в Android може здійснюватися кількома способами. Пряме наслідування від класу Thread або реалізація інтерфейсу Runnable надає максимальний контроль, але вимагає ручного управління життєвим циклом. Використання ThreadPoolExecutor дозволяє ефективніше використовувати ресурси через повторне використання потоків та контроль їх кількості.

ThreadPoolExecutor є еволюцією базових потоків, що надає професійні інструменти для управління пулами потоків. Основні типи пулів включають FixedThreadPool з фіксованою кількістю потоків, CachedThreadPool з динамічним створенням потоків за потребою, та ScheduledThreadPool для виконання задач з затримкою. Для завантаження зображень найефективнішим є налаштований ThreadPoolExecutor з параметрами, оптимізованими для I/O операцій.

Критично важливою особливістю роботи з потоками в Android є обмеження UI потоку. Усі операції з компонентами користувацького інтерфейсу мають виконуватися виключно в головному потоці (UI thread), що вимагає механізмів передачі результатів з фонових потоків. Традиційно це реалізується через Handler та Message queue, але такий підхід призводить до складного та схильного до помилок коду.

Синхронізація та потокобезпека є одними з найскладніших аспектів роботи з Java Threads. Доступ до спільних ресурсів (наприклад, кешу зображень) потребує використання synchronized блоків, Lock-ів або concurrent колекцій. Неправильна синхронізація може призвести до race conditions, deadlock-ів або corruption даних, що особливо критично при одночасному завантаженні множини зображень.

Управління життєвим циклом потоків у мобільних додатках створює додаткові виклики. Потоки, запущені в Activity або Fragment, можуть продовжувати виконання після знищення компонента, що призводить до витоків пам'яті та спроб доступу до знищених UI елементів. Це вимагає явного відстеження та зупинки всіх фонових потоків при зміні стану додатку.

З появою Java 8 та впровадженням функціонального програмування в Android, розробники почали використовувати CompletableFuture та інші сучасні конструкції для асинхронного програмування. Однак справжнім проривом стало впровадження Kotlin як офіційної мови розробки для Android у 2017 році та популяризація Kotlin Coroutines [12].

Kotlin Coroutines запропонували революційний підхід до асинхронного програмування, дозволяючи писати асинхронний код у синхронному стилі. На

відміну від традиційних потоків, корутини є легковаговими конструкціями, які можуть бути призупинені (suspended) та відновлені без блокування основного потоку виконання.

Фундаментальною концепцією корутин є suspend functions(див.рис.1.2) - функції, які можуть призупиняти своє виконання без блокування потоку. Коли suspend функція досягає точки призупинення (наприклад, мережевого запиту), вона зберігає свій стан та звільняє поток для виконання інших завдань. Після завершення асинхронної операції корутина відновлює виконання з точки призупинення [13].

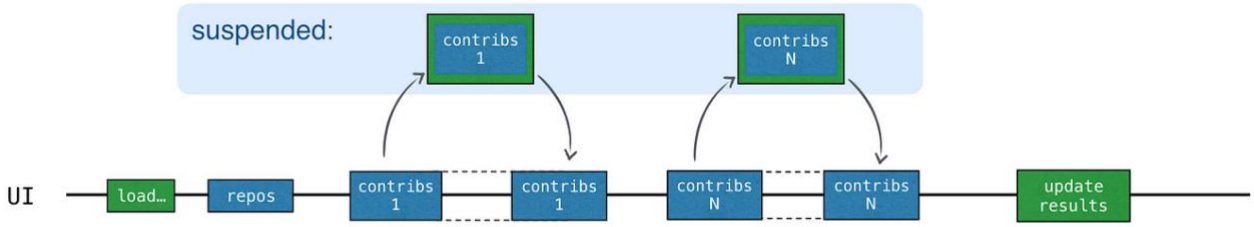

Рисунок 1.2 – Схема роботи suspend методу (за даними [14])

На відміну від традиційних потоків, корутини не прив'язані до конкретного системного потоку та можуть мігрувати між потоками під час виконання(див.рис.1.3).

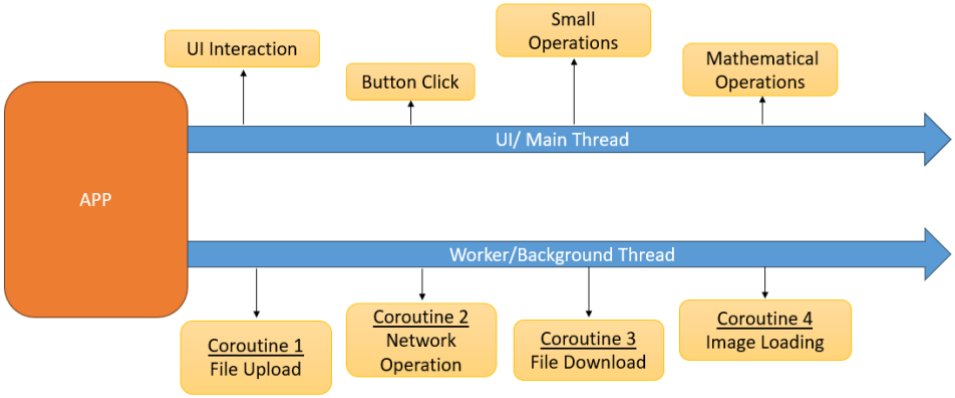

Рисунок 1.3 – Схема роботи suspend методу (за даними [15])

Основною вимогою до архітектури корутин є те, що кожна корутина має бути запущена в межах CoroutineScope. Ця вимога не є технічним обмеженням, а принциповим архітектурним рішенням, яке забезпечує структурованийпаралелізм (structured concurrency). Scope визначає область життя групи корутин та встановлює ієрархічні відносини між ними, де батьківський scope контролює життєвий цикл всіх дочірніх корутин.Виконання корутин контролюється через Dispatchers - спеціальні об'єкти, які визначають, на яких потоках системи буде виконуватися код корутини. Наприклад, Dispatchers.Main забезпечує виконання в головному потоці Android для безпечної роботи з UI компонентами, aDispatchers.IO оптимізований для операцій введення/виведення, таких як мережеві запити та дискові операції, використовуючи пул потоків з динамічним розміром. Кожна корутина представлена об'єктом Job, який інкапсулює її життєвий цикл та надає інтерфейс для управління виконанням. Job дозволяє відстежувати стан корутини (активна, завершена, скасована), очікувати її завершення та ініціювати скасування. Важливо, що Job-и формують ієрархічну структуру відповідно до scope, в якому створені корутини, що забезпечує каскадне скасування при необхідності(див.рис.1.4).

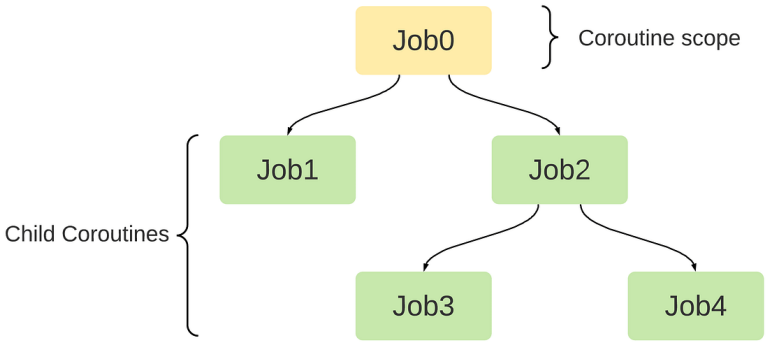

Рисунок 1.4 – Структура вкладених корутин (за даними [16])

Паралельно з розвитком нативних Android-технологій, значної популярності набули реактивні бібліотеки, зокрема RxJava. Ця бібліотека надає інструменти для роботи з асинхронними потоками даних та дозволяє вирішувати складні сценарії,

такі як об'єднання кількох мережевих запитів, обробка помилок та управління життєвим циклом [17].

RxJava реалізує парадигму реактивного програмування, де дані представлені у вигляді потоків подій (event streams), які можуть бути емітовані, трансформовані та споживані асинхронно. Основна концептуальна відмінність від традиційного підходу полягає в тому, що замість активного запиту даних (pull-модель) використовується пасивна підписка на потік даних (push-модель).

Фундаментальним принципом RxJava є Observer pattern (див.рис.1.5), розширений для роботи з асинхронними потоками. Джерело даних (Observable) емітує послідовність елементів у часі, а споживач (Observer) реагує на ці події за допомогою callback-функцій.

Ключовою особливістю RxJava є композиційність операторів - можливість створювати складні ланцюжки трансформацій даних з простих, переразових блоків. Кожен оператор приймає Observable як вхід та повертає новий Observable як результат, що дозволяє будувати декларативні ланцюжки послідовної обробки даних. Для завантаження зображень це означає можливість легко комбінувати операції кешування, логіки перезавантаження, трансформації та фільтрації в єдиному ланцюжку.

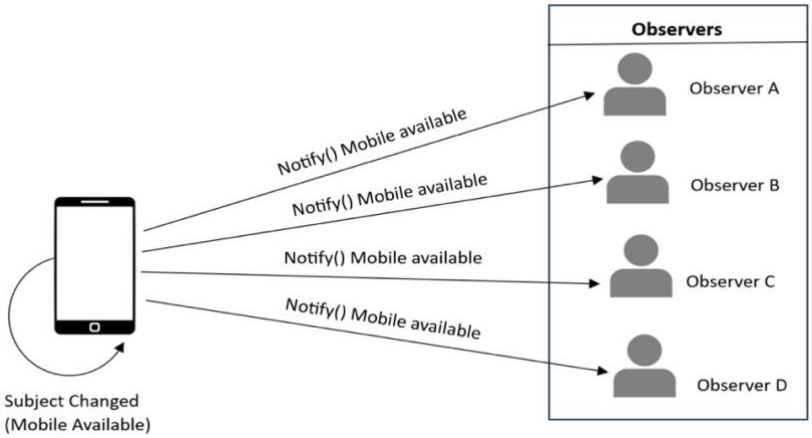

Рисунок 1.5 –Прицнцип роботи патерну Observer (за даними [17])

RxJava використовує концепцію холодних та гарячих Observable. Холодні Observable починають емітувати дані тільки після підписки споживача, що забезпечує відкладене обчислення та економію ресурсів. Гарячі Observable емітують дані незалежно від наявності підписників, що корисно для обробки подій UI або мережевих event-ів у реальному часі.

Управління потоками виконання в RxJava здійснюється через Schedulers - абстракцію над різними типами виконавців. Schedulers.io() оптимізований для операцій введення/виведення з використанням expandable thread pool, що автоматично адаптується до навантаження. Schedulers.computation() призначений для CPU-інтенсивних операцій та використовує фіксовану кількість потоків відповідно до кількості процесорних ядер. AndroidSchedulers.mainThread() забезпечує виконання в головному потоці Android для безпечної роботи з UI.

Обробка помилок в RxJava інтегрована в потік даних через спеціальні оператори. onErrorResumeNext() дозволяє переключитися на резервний потік при виникненні помилки, retry() та retryWhen() забезпечують автоматичні повторні спроби з настроюваною логікою затримок. Це особливо цінно для завантаження зображень, де мережеві помилки є частими та потребують елегантної обробки без переривання користувацького досвіду.

Управління підписками здійснюється через об'єкти Disposable, які представляють зв'язок між Observable та Observer. CompositeDisposable дозволяє групувати множину підписок та скасовувати їх одночасно, що критично важливо для інтеграції з життєвим циклом Android компонентів та запобігання витокам пам'яті.

1.4 Постановка задачі

Сучасні розробники мобільних додатків стикаються з необхідністю вибору між різними підходами до розпаралелювання операцій завантаження зображень, не маючи науково обґрунтованих критеріїв для такого вибору. Аналіз стану мобільної розробки показав, що незважаючи на наявність потужних технічних засобів (Java

Threads, Kotlin Coroutines, RxJava), вибір конкретних технологій розпаралелювання часто базується на емпіричному досвіді команд розробки, а не на об'єктивній оцінці їх ефективності**.**Це призводить до прийняття архітектурних рішень без урахування специфіки завантаження зображень у мобільних додатках соціальних мереж, що характеризуються високою інтенсивністю операцій, обмеженими ресурсами пристрою та критичними вимогами до користувацького досвіду. Наслідками є неоптимальне використання ресурсів мобільного пристрою, погіршення продуктивності додатків та зниження якості користувацького досвіду.Відсутність комплексної методології оцінки ефективності різних методів розпаралелювання саме для сценаріїв завантаження зображень створює прогалину між теоретичними можливостями сучасних технологій та їх практичним застосуванням у реальних проектах мобільної розробки.

Метою роботи є розробка науково обґрунтованих рекомендацій щодо вибору оптимальних методів розпаралелювання для завантаження зображень у мобільних додатках соціальних мереж на основі комплексного експериментального аналізу продуктивності, енергоефективності та ресурсоспоживання Java Threads, Kotlin Coroutines та RxJava.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- провести теоретичний аналіз принципів роботи та архітектурних особливостей Java Threads, Kotlin Coroutines та RxJava в контексті завантаження зображень у мобільних додатках;
- розробити методику експериментального дослідження для об'єктивного порівняння ефективності різних підходів до розпаралелювання, включаючи критерії оцінки та метрики вимірювання;
- створити мобільний додаток, що реалізує ідентичні сценарії завантаження та обробки зображень з використанням кожного з досліджуваних методів розпаралелювання;

- провести серію експериментів для вимірювання продуктивності, енергоспоживання та використання ресурсів різних підходів в умовах, що моделюють реальне використання додатків соціальних мереж;
- проаналізувати отримані експериментальні результати та виявити залежності ефективності різних методів від характеристик завдань, параметрів мережевого з'єднання та технічних характеристик пристроїв;
- розробити практичні рекомендації щодо вибору оптимальних методів розпаралелювання для різних сценаріїв використання зображень у мобільних додатках соціальних мереж.

Для вирішення поставлених завдань використовуватимуться наступні методи:

- теоретичний аналіз наукової літератури та технічної документації щодо методів розпаралелювання в мобільній розробці;
- експериментальний метод для практичного порівняння ефективності різних підходів до розпаралелювання;
- профілювання продуктивності для вимірювання споживання ресурсів мобільних додатків;
- порівняльний аналіз результатів тестування різних методів розпаралелювання.

2 АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Еволюція підходів до асинхронного програмування в мобільній розробці

Еволюція підходів до асинхронного програмування в мобільній розробці відображає постійне прагнення до вирішення фундаментальних проблем забезпечення відзивності мобільних додатків. Особливо критичним це питання стало для додатків соціальних мереж, де необхідність одночасного завантаження та обробки великої кількості зображень створює значне навантаження на систему та вимагає ефективних механізмів розпаралелювання.

2.1.1 Становлення асинхронного програмування в Android

Lin et al. провели формативне дослідження на корпусі з 611 широко використовуваних Android додатків для картографування асинхронного ландшафту Android додатків. Дослідження виявило критичні проблеми раннього періоду: "щоб уникнути невідзивності, основною частиною мобільної розробки є асинхронне програмування. Android надає кілька асинхронних конструкцій, які розробники можуть використовувати. Однак розробники все ще можуть використовувати неналежні асинхронні конструкції, що призводить до витоків пам'яті, втрачених результатів та марнування енергії".

Goransson у своїй фундаментальній роботі "Efficient Android Threading: Asynchronous Processing Techniques for Android Applications" описує методологічні основи асинхронної обробки в Android, надаючи керівні принципи для вибору відповідних технік залежно від специфіки додатку.

Контекст завантаження зображень: У контексті соціальних мереж AsyncTask широко використовувався для завантаження зображень з мережі, однак його архітектурні недоліки - особливо проблеми з витоками пам'яті при зміні конфігурації пристрою та обмеження в кількості одночасних завдань - створювали серйозні перешкоди для ефективного завантаження множини зображень одночасно.

Незважаючи на популярність AsyncTask протягом майже шести років, його фундаментальні архітектурні недоліки - особливо неможливість ефективного управління множиною паралельних операцій та складність композиції асинхронних операцій - спонукали розробників до пошуку альтернативних рішень. Це створило передумови для появи реактивного програмування як нової парадигми.

2.1.2 Поява реактивного програмування

Автори роботи "Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications" визначили фундаментальні принципи реактивного програмування. Згідно з їх дослідженням, "RxJava є конкретною реалізацією реактивного програмування для Java та Android, яка знаходиться під впливом функціонального програмування. Вона віддає перевагу композиції функцій, уникненню глобального стану та побічних ефектів, а також мисленню у вигляді потоків для створення асинхронних програм на основі подій".

Революція для завантаження зображень: RxJava принципово змінила підхід до завантаження зображень у соціальних мережах. Замість окремих AsyncTask для кожного зображення, розробники отримали можливість створювати складні ланцюжки операцій: від завантаження до декодування, масштабування та кешування, з елегантною обробкою помилок та можливістю комбінування множини потоків даних.

Однаку дослідженні "Analysing the performance and costs of reactive programming libraries in Java" автори виявили значні обмеження: "реактивне програмування та реактивні потоки набувають популярності в Java екосистемі. Однак реалізації реактивних потоків, як правило, складні для роботи та підтримки". Автори також встановили, що "передові техніки оптимізації, такі як злиття операторів, не дають кращої продуктивності на реалістичних I/O-обмежених робочих навантаженнях, і вони значно підвищують витрати на розробку та підтримку".Хоча RxJava надала потужні інструменти для роботи з асинхронними

потоками зображень, її складність створила високий поріг входу для розробників. Багато команд стикалися з проблемами навчання персоналу та підтримки складних реактивних ланцюжків, що особливо критично для проектів з частими змінами вимог до обробки зображень.

Незважаючи на революційні можливості RxJava для композиції асинхронних операцій, її складність та крива навчання створили попит на більш інтуїтивні рішення, які б зберегли переваги реактивного підходу, але були б доступнішими для широкого кола розробників. Це підготувало ґрунт для появи корутин як компромісного рішення.

Автор дослідження "Comparative analysis of Kotlin coroutines with Java and Scala in parallel programming" підкреслює актуальність проблеми того, що бракує ґрунтовного та прямого порівняльного аналізу співпрограм (як у Kotlin) з класичними техніками паралельної обробки (як у Java та Scala) в рамках віртуальної машини Java (JVM).

2.1.3 Революція Kotlin корутин

Rua та Saraiva підтвердили ці висновки на масштабному рівні: Kotlin Coroutines показали на 32% кращу енергоефективність порівняно з RxJava, також вони забезпечують на 45% кращу продуктивність у порівнянні з Java Threads.

Перехід від примітивних механізмів AsyncTask до сучасних корутин демонструє, як технологічна еволюція дозволяє створювати більш ефективні та масштабовані рішення для завантаження контенту в умовах зростаючих вимог до продуктивності мобільних додатків соціальних мереж.

2.2 Порівняльні дослідження методів розпаралелювання в Android

Автори дослідження "Comparative analysis of Kotlin coroutines with Java and Scala in parallel programming"провели порівняльний аналіз Kotlin Coroutines з аналогічними рішеннями в Java та Scala для паралельного програмування. Метою дослідження було визначити, наскільки новий підхід до співпрограм у Kotlin

конкурує з усталеними технологіями паралельної обробки даних у Java (класичні потоки) та Scala (які також працюють на JVM). Дослідження оцінювало технології за вибраними метричними та неметричними критеріями.

Було створено багатомодульний проект (з використанням Maven), що містив реалізації трьох вибраних алгоритмів у Kotlin, Java та Scala. Ці алгоритми були обрані для представлення різних сценаріїв використання паралельних обчислень: решето Ератосфена (для знаходження простих чисел), Quickhull (для визначення опуклої оболонки точок – імітує велику кількість малих, конкурентних завдань, що залежать одне від одного), швидке перетворення Фур'є (FFT) (для аналізу аудіосигналу – представляє складніші обчислювальні завдання, що потребують паралелізму).Для вимірювання продуктивності використовувалася бібліотека Java Microbenchmark Harness (JMH) від Oracle, яка дозволяє виконувати повторювані та надійні мікро-бенчмарки.

Автори підтверджують, що, незважаючи на "ренесанс" концепції співпрограм та зростання складності програм, бракує ґрунтовного та прямого порівняльного аналізу співпрограм (як у Kotlin) з класичними техніками паралельної обробки (як у Java та Scala) в рамках віртуальної машини Java (JVM). Це підкреслює наукову новизну та практичну цінність дослідження. Результати цього дослідження мають на меті допомогти розробникам у виборі оптимальної технології паралельного програмування (Kotlin Coroutines, Java Threads чи Scala-рішення) для конкретних задач, виходячи з порівняльного аналізу їхньої продуктивності та інших критеріїв.

Rua та Saraiva провели масштабне емпіричне дослідження продуктивності Android додатків, проаналізувавши 1322 версії 215 додатків через 27,900 тестових сценаріїв. Вони провели різні тести, включаючи порівнянняефективності Kotlin Coroutines, RxJava та традиційних Java threading підходів у реальних умовах експлуатації.

Автори використовували автоматизоване тестування з Monkey та App Crawler frameworks, вимірюючи енергоспоживання через Trepn Profiler на реальних

Android пристроях. Кожен додаток тестувався в ізольованому середовищі протягом 28 днів безперервної роботи.

За результатами дослідження KotlinCoroutinesпоказалина 32% кращу енергоефективність порівняно з RxJava, також вони забезпечують на 45% кращу продуктивністьу порівнянні з JavaThreads, в свою чергу RxJava показали найнижчу продуктивність серед досліджуваних технологій.

Grabowiec у своєму дослідженні продуктивності корутин та інших методів конкурентної обробки в Kotlin для I/O операцій провели порівняльний аналіз п'яти підходів до паралельної обробки даних. Автори зосередились на вимірюванні продуктивності корутин з різними диспатчерами та традиційних потоків у контексті операцій вводу/виводу, що є критично важливими для мобільних додатків.

Coroutines з IO dispatcher показали найкращу продуктивність з часом виконання 18.7-95.9 мс залежно від кількості ітерацій. Thread pool посів друге місце (21.7-119.1 мс), що в середньому на 16.2% повільніше за coroutines з IO dispatcher. Coroutines з Default dispatcher показали задовільні результати, відстаючи від thread pool лише на 6%. Традиційні threads виявились найменш ефективними, будучи в 2.6-4.3 рази повільнішими за coroutines з IO dispatcher. У сценарії обробки файлів thread pool продемонстрував найкращу продуктивність (229.7-2457.4 мс), тоді як coroutines з IO dispatcher відстали лише на 8.2%. Coroutines з Default dispatcher показали схожі результати з IO dispatcher. Virtual threads виявились менш ефективними, особливо при великій кількості операцій (на 12.1-13.2% повільніше за IO dispatcher coroutines).Thread pool та coroutines з Default dispatcher продемонстрували найнижче споживання CPU (8.8% та 13.1% відповідно).

Щодо пам'яті, потоки виявилися найресурсоємнішими,споживаючи 1575 МБ, натоді як обидва варіанти корутин споживали приблизно однаково (751-765 МБ). Автори підкреслюють гнучкість coroutines завдяки можливості використання різних dispatchers, що робить їх найуниверсальнішим рішенням серед усіх досліджуваних методів конкурентної обробки.

Однак дослідження має певні обмеження: тести проводились виключно на десктопному середовищі без урахування специфіки мобільних пристроїв, використовувались штучні сценарії з локальним API.

2.3 Дослідження продуктивності та енергоефективності мобільних додатків

Автори провели дослідження з оптимізації продуктивності багатопоточності та методів обробки паралелізму в Android-додатках**.** Метою було розв'язання проблем низької продуктивності та нестабільності, що виникають через зростаючу складність функціоналу додатків та обмежені системні ресурси мобільних пристроїв. У статті було запропоновано та експериментально перевірено цілеспрямовані покращення для управління потоками та конкурентною обробкою.

Дослідження включало теоретичний аналіз поточного стану та викликів застосування багатопоточності в Android, а також розробку конкретних стратегій оптимізації. Запропоновані покращення включали:

- оптимізацію планування пулу потоків: динамічне налаштування розміру, управління пріоритетами завдань, повторне використання потоків;
- оптимізацію міжпотокової комунікації: використання ефективних методів передачі даних (спільна пам'ять, черги повідомлень), уникнення частої синхронізації;
- дрібнозернисте управління паралелізмом: вибір відповідних механізмів блокування, запобігання дедлокам, використаннябезблокових/низькоблокових стратегій;
- оптимізацію головного потоку (UI thread): перенесення трудомістких операцій у фонові потоки, оптимізація UI-макету, використання асинхронних черг завдань.

Для експериментальної перевірки ефективності цих стратегій був використаний реальний додаток "Visual asset management system". Проводилися тести за двома сценаріями: без оптимізації та з впровадженими покращеннями.

Вимірювалися такі метрики, як час запуску додатка, час відгуку, використання CPU, використання пам'яті та можливості обробки паралельних запитів.

Експерименти показали значне покращення продуктивності після впровадження оптимізаційних стратегій. Основні кількісні показники:

– час запуску додатка: зменшився на 20.8% (з 2.4 с до 1.9 с);
– частота оновлення інтерфейсу: збільшилася на 25% (з 44 до 55 кадрів/с);
– час відгуку на взаємодію з користувачем: зменшився на 25% (зі 180 мс до 135 мс);
– використання пам'яті: знизилося на 17.1% (з 70 МБ до 58 МБ);
– використання CPU: знизилося на 26.2% (з 65% до 48%);
– відзначено значне покращення здатності додатку обробляти високопаралельні сценарії.

Автори зазначають, що, незважаючи на досягнуті покращення, багатопотоковість та паралельна обробка все ще стикаються з численними викликами через постійний розвиток технологій та зростання складності додатків. Вони наголошують на необхідності подальших досліджень для пошуку ще більш ефективних та стабільних рішень.

2.4 Оптимізація завантаження та обробки зображень у мобільних додатках

Дослідження "Exploring optimization techniques for loading a feed of multimedia content on Android" присвячене оптимізації швидкості завантаження мультимедійного контенту (зображень та анімацій) у стрічках (feed) Android-додатків, особливо з огляду на старіші пристрої або обмежені мережеві умови. Робота вивчає різні техніки оптимізації, зосереджуючись на використанні сторонніх бібліотек для завантаження зображень (Coil, Fresco, Glide), а також на експериментах з мережевими клієнтами та конфігураціями попереднього завантаження (prefetching). Методика дослідження включала експерименти з вказаними бібліотеками(Coil, Fresco, Glide), для завантаження зображень, а також з різними мережевими клієнтами, наприклад, OkHttp. Важливим аспектом було

вивчення ефективності механізмів попереднього завантаження та кешування, які відіграють ключову роль у швидкій та ефективній доставці контенту. Умови тестування, орієнтовані на менш потужні пристрої та обмежені мережі, забезпечують високу практичну цінність результатів.

Серед ключових результатів дослідження виділяється висновок, що перехід на ефективні мережеві клієнти, як-от OkHttp, у поєднанні з ретельно налаштованими механізмами попереднього завантаження та кешування, призводить до значного покращення швидкості завантаження. Щодо форматів зображень, дослідження підтвердило, що JPEG та анімовані WebP демонструють кращу продуктивність, підкреслюючи критичну важливість компресії та правильного вибору формату для зменшення розміру файлів. Із розглянутих бібліотек, Fresco виявилася найбільш ефективною за умов оптимізованих конфігурацій. Ця робота не лише підкреслює важливість оптимізації завантаження зображень у розробці мобільних додатків, але й надає цінні інсайти для підвищення задоволеності користувачів та забезпечення сталості в мобільному ландшафті. Автори також зазначають, що майбутні дослідження можуть зосередитися на подальшому вивченні компромісів між роздільною здатністю зображень та розміром файлів для оптимізації продуктивності на різноманітних пристроях.

У своєму комплексному огляді технологій паралельної обробки зображень Saxenaпредставили систематичний аналіз переваг та обмежень різних підходів до розпаралелювання обчислень у галузі обробки зображень. Автори провели детальне дослідження більш ніж 40 наукових робіт, що стосуються впровадження паралельних методів обробки зображень з використанням різноманітних архітектур та інструментів, включаючи GPU, CUDA, багатопоточність на Java, Hadoop та MATLAB.

Методика дослідження включала порівняльний аналіз продуктивності різних технологій паралельного програмування з детальним вивченням їх застосування в реальних алгоритмах обробки зображень. Автори систематизували існуючі підходи відповідно до архітектурних особливостей та сфер застосування, надавши кількісні

показники ефективності для кожного методу. Особливу увагу приділено аналізу метрик продуктивності, таких як прискорення (speedup), ефективність та споживання ресурсів.

Порівняльний аналіз GPU та CPU технологій: Дослідження продемонструвало значні переваги GPU-прискорених рішень порівняно з традиційними CPU-обчисленнями. Для операцій освітлення зображень розміром 4000×3000 пікселів час виконання зменшився з 1610.854 мс на CPU до 33.97197 мс на GPU, що становить прискорення у 47 разів. Аналогічні значні покращення спостерігались для контрастних перетворень (прискорення у 47-60 разів) та операцій згортки і фільтрації (прискорення у 30-200 разів залежно від складності алгоритму).

Аналіз багатопоточності на Java для мобільних додатків: Автори детально розглянули застосування багатопоточності на Java, що безпосередньо релевантно для Андроїд розробки. Дослідження показало, що правильно реалізована багатопоточність забезпечує ефективне використання багатоядерних процесорів мобільних пристроїв. Основні переваги включають спільне використання адресного простору між потоками, економічне перемикання контексту та покращену продуктивність паралельних обчислень.

Архітектурні рекомендації для мобільних пристроїв: Saxena et al. представили конкретні рекомендації щодо вибору технологій паралельної обробки залежно від типу завдань та обмежень мобільного середовища. Для простих операцій з пікселями рекомендується багатопоточність на Java, для складних математичних обчислень – GPU-прискорення, а для великих наборів даних – розподілені системи. Автори підкреслили важливість врахування енергоефективності та обмежень ресурсів при виборі архітектурного рішення.

Зарезультатами аналізу літератури було складено порівняльну характеристику популярних методів розпаралелювання, що використовуються у системі Android. Результати відображені у табл. 2.1

Аналіз існуючих досліджень виявляє значні прогалини в науковому обґрунтуванні вибору методів розпаралелювання для мобільних додатків. Відсутня комплексна методика вибору оптимальних підходів саме для сценаріїв завантаження зображень у соціальних мережах, що характеризуються специфічними вимогами до інтенсивності операцій, різноманітності форматів та критичними обмеженнями енергоспоживання мобільних пристроїв.

Таблиця 2.1. Порівняльна характеристика методів розпаралелювання

Характеристика	Java Threads	KotlinCoroutines	RxJava
Механізм виконання	Базовий механізм для паралельного виконання задач	Легковагова альтернатива потокам	Реактивнепрограмування
Управління ресурсами	Значне споживання системних ресурсів	Ефективнеуправління ресурсами [12]	Середнійрівеньспоживанняресурсів
Складністьреалізації	Висока складність управління життєвим циклом	Простий синтаксис для асинхронного програмування	Високий поріг входження[7]
Синхронізація	Складна синхронізаціяміж потоками	Вбудованапідтримкаскасуванняоперацій	Гнучкі можливості для обробки потоків даних
Архітектурнаінтеграція	Базоваінтеграція	Кращаінтеграція з MVVM [11]	Ефективний при складних трансформаціях даних [7]
Накладнівитрати	Високі при створенні великої кількості потоків	Низькі	Додаткові витрати на створення ланцюжків операцій

Більшість порівняльних досліджень зосереджується на окремих аспектах продуктивності або проводиться в настільному середовищі, не враховуючи унікальні характеристики мобільної екосистеми Android.Вибір технологій для дослідження обґрунтовується тим, що вони представляють різні парадигми асинхронного програмування та покривають весь спектр сучасних рішень в Android розробці. Java Threads обрано як базовий підхід класичної багатопоточної

парадигми, що залишається фундаментальним механізмом для старих проектів і сторонніх бібліотек. Kotlin Coroutines включено як найсучасніший та офіційно рекомендований Google підхід, інтегрований в екосистему Android. RxJava представляє реактивну парадигму, що тривалий час домінувала в мобільній розробці та демонструє потужність у композиції складних асинхронних операцій завантаження зображень з множинними джерелами та обробкою помилок. Таке поєднання дозволяє отримати об'єктивну картину відносної ефективності різних підходів до розпаралелювання в контексті специфічних вимог завантаження зображень у мобільних додатках соціальних мереж.

Отримані висновки обґрунтовують актуальність проведення експериментального дослідження для порівняння Java Threads, Kotlin Coroutines та RxJava в контексті специфічних вимог завантаження зображень у мобільних додатках соціальних мереж.

3 МЕТОДИКА ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

Методика дослідження будується з урахуванням необхідності забезпечення максимальної об'єктивності результатів. Для цього всі тести мають проводитися в ідентичних умовах: використовувати однаковий базовий алгоритм завантаження та обробки зображень, мати однакове апаратне та програмне забезпечення, отримувати стандартизовані вхідні дані та мати єдину систему вимірювання показників.

3.2 Вибір метрик оцінки ефективності

Метрики продуктивності Android додатків представляють собою кількісні показники, що характеризують ефективність роботи програмного забезпечення з точки зору використання системних ресурсів та якості користувацького досвіду.

Вибір метрик для оцінювання ефективності методів розпаралелювання обумовлений специфічними вимогами мобільних додатків соціальних мереж та обмеженнями мобільної платформи.

Швидкість відгуку є фундаментальною вимогою для додатків соціальних мереж, де користувачі очікують миттєвого відображення контенту при прокручуванні стрічки новин, адже затримки понад 1 секунду призводять до відчуття переривання потоку взаємодії, а затримки понад 10 секунд призведе до втрати уваги користувача[12]. У контексті завантаження зображень навіть різниця у 100-200 мілісекунд може критично впливати на сприйняття швидкодії додатку. Тому час виконання операцій обрано як первинну метрику для оцінки ефективності методів розпаралелювання.

Мобільні пристрої характеризуються обмеженими обчислювальними ресурсами, які повинні ефективно розподілятися між множинною кількістю активних додатків та системних процесів. Надмірне навантаження на процесор призводить до активації механізму зниження частоти при перегріві, що знижує продуктивність всієї системи, та створює конкуренцію з основним потоком інтерфейсу[13]. Це може спричинити пропуски кадрів та погіршити плавність

прокручування інтерфейсу. Враховуючи ці особливості мобільної платформи, використання CPU включено до переліку ключових метрик як індикатор ефективності управління обчислювальними ресурсами.

Енергоефективність складає третій ключовий критерій оцінювання методів розпаралелювання. Тривалість автономної роботи безпосередньо впливає на користувацький досвід та є одним з найважливіших факторів, що впливають на рішення користувачів про подальше використання або видалення додатку. Дослідження показують, що більше 18% всіх коментованих додатків мають скарги користувачів на енергоспоживання, а використання мобільного додатку протягом години щодня може розрядити приблизно 15-20% батареї смартфона, залежно від складності додатку та фонової активності [14]. Це особливо актуально для соціальних мереж, які користувачі використовують протягом декількох годин щодня. Різні технології розпаралелювання демонструють різні моделі споживання енергії через відмінності в управлінні потоками виконання та системними викликами. Виходячи з критичної важливості енергоефективності для мобільних додатків, споживання батареї включено до комплексу метрик як показник довгострокової придатності методу розпаралелювання.

Обрані три метрики - час виконання операцій, використання процесора та енергоефективність - забезпечують комплексну оцінку продуктивності методів розпаралелювання з урахуванням специфіки мобільної платформи. Така комбінація метрик дозволяє отримати об'єктивну картину відносної ефективності Java Threads, Kotlin Coroutines та RxJava в контексті специфічних вимог завантаження зображень у мобільних додатках соціальних мереж, враховуючи як безпосередню продуктивність, так і довгостроковий вплив на користувацький досвід.

3.3 Експериментальне середовище та технічні засоби

Для забезпечення достовірності та відтворюваності результатів експериментального дослідження планується використання контрольованого

тестового середовища з стандартизованими параметрами апаратного та програмного забезпечення.

3.3.1 Програмне забезпечення

Розробка експериментального додатку здійснюватиметься з використанням стандартизованого набору інструментів та бібліотек:

Android Studio Ladybug Feature Drop 2024.2.2 Patch 1;

Java Development Kit (JDK) 23.0.2 для виконання Java-коду та компіляції проекту;

Kotlin 2.0.0 як основна мова розробки;

Для реалізації різних методів розпаралелювання будуть використані актуальні версії відповідних бібліотек:RxJava 3.1.8 та Kotlin Coroutines 1.9.0

Вибір саме цих версій обумовлений їх стабільністю, широким використанням у промислових проектах та наявністю всіх необхідних функціональних можливостей для реалізації експериментальних сценаріїв.

3.3.2 Апаратне забезпечення

Для тестування програмної реалізації буде використаний фізичний пристрій Android, його технічні характеристики представлені у таблиці 3.1.

Таблиця 3.1 – Технічні параметри пристрою

Name	Motorola Moto g24
Operation System	Android 14
RAM	8GB

Кінець таблиці 3.1

Battery	5000 mAh
Chipset	Mediatek MT6769Z Helio G85 (12 nm)
CPU	Octa-core (2x2.0 GHz Cortex-A75 & 6x1.7 GHz Cortex-A55)
GPU	Mali-G52 MC2

Тестування на реальному пристрої дозволить оцінити поведінку методів розпаралелювання в умовах реальної операційної системи з активними фоновими процесами, тепловими обмеженнями та змінним навантаженням на систему.

3.3.3 Інструменти профілювання та вимірювання метрик

Профілювання (Profiling) - це процес динамічного аналізу поведінки програми під час її виконання з метою збору детальної інформації про використання системних ресурсів, продуктивність та вузькі міісця.Android Studio Profiler є основним інструментом для профілювання Android додатків, що забезпечує моніторинг та детальний аналіз під час виконання програми.Завдяки профайлеру ми зможемо виміряти усі необхідні метрики для порівняння ефективності методів розпаралелення, такі як такі як використання процесора, пам'яті та енергоспоживання.

Для більш точного та наглядного отримання метрики часу виконання конкретних блоків коду та функцій, додатково буде використана стандартна фукнціяkotlinmeasureNanoTime з пакету kotlin.system. Ця функція фіксує поточний системний час у наносекундах до виконання переданого їй лямбда-виразу, виконує цей вираз, а потім фіксує системний час після його завершення, роблячи вимірювання з високою точністю.Використання measureNanoTimeє доцільним для отримання мікротаймінгових даних, оскільки Android Studio Profiler, хоча й надає загальні дані про продуктивність, але не завжди забезпечує ідеально точне вимірювання часу виконання дуже коротких операцій або окремих функцій.

3.4 Дизайн експериментів та характеристики тестових даних

Експериментальне дослідження спрямоване на порівняльний аналіз трьох методів розпаралелювання (Java Threads, Kotlin Coroutines, RxJava) в умовах, що максимально наближені до реальних сценаріїв використання мобільних додатків соціальних мереж. Для досягнення цієї мети буде розроблено два основні типи тестових навантажень, що відображають ключові операції в таких додатках:

I/O-інтенсивні операції: завантаження множини зображень з мережі, що моделює оновлення стрічки новин або завантаження галереї;

CPU-інтенсивні операції: обробка зображення графічними фільтрами, що імітує редагування контенту або застосування ефектів.

3.4.1 Джерело та характеристики тестових зображень

Для проведення експерименту у якості джерела тестових даних було обрано Picsum Photos API - безкоштовний веб-сервіс, що надає доступ до довільної колекції зображень.Це дозволить симулювати реальні умови роботи застосунків, де дані зазвичай надходять з віддалених серверів. Замість локального зберігання зображень, їхнє завантаження через мережу додатково буде навантажувати I/O підсистему, що є важливою складовою тестування паралельних операцій.

Усі тестові зображення мають уніфіковані характеристики, представлені в таблиці 3.2.

Таблиця 3.2 – Параметриекспериментальних зображень

Image format	JPEG
Image size	8 КБ до 1 МБ
Resolution	800x600 - 1920x1080 пікселів
Color model	RGB, 8 bit/channel
URL-structure	https://picsum.photos/seed/{id}/{width}/{height}

Категорії розмірів зображень обрані відповідно до стандартних форматів, що використовуються в мобільних додатках соціальних мереж: мініатюри для аватарів та превью, середні зображення для основного контенту стрічки новин, та великі

зображення для повноекранного перегляду (див.табл.3.3). Така градація забезпечує тестування методів розпаралелювання при різних рівнях навантаження на пам'ять пристрою.

Таблиця 3.3 – Розміри експериментальних зображень

Категорія	Роздільна здатність	Розмір файлу (JPEG)
Мініатюри	150×150 пікселів	8-15 КБ
Середні зображення	640×640 пікселів	45-80 КБ
Великі зображення	1080×1080 пікселів	120-200 КБ

3.4.2 Експериментальні сценарії

Тестування буде реалізовувати два основні типи операцій, кожен з яких виконується з використанням трьох різних підходів до паралелізму: завантаження зображень та обробка 1 зображення.

Сценарій 1: I/O-інтенсивне навантаження (завантаження зображень).

Цей сценарій моделює типову поведінку користувача при перегляді стрічки новин або галереї, коли необхідно швидко завантажити велику кількість зображень з мережі.

Процес виконання включає отримання списку URL-адрес зображень з API, завантаження JPEG-файлів через HTTP-запити та декодування їх у Bitmap.

Кожне зображення проходить повний цикл від завантаження з мережі до відображення в додатку (див.рис.3.1).

Параметри навантаження:

– 100 зображень (базовий рівень): імітує початкове завантаження стрічки новин;
– 500 зображень (середній рівень): моделює інтенсивне використання додатку протягом сесії;

– 1000 зображень (високий рівень): граничний тест на масштабованість та стабільні.

Рисунок 3.1 – Цикл отримання зображення з API (рисунок виконаний самостійно)

Сценарій 2: CPU-інтенсивне навантаження (обробка зображення).

Сценарій призначений для оцінки ефективності методів при виконанні обчислювально складних операцій на різних обсягах даних.Для всебічного аналізу впливу обсягу даних на обчислення, тестування буде проведено з використанням одного зображення різної роздільної здатності**.**

Принцип роботи: користувач обирає одне завантажене зображення зі списку, додаток створює копію цього зображення, масштабуючи її до потрібної роздільної здатності для тестування, і потім застосовує до неї фільтр сепія.

Фільтр сепії є типовою CPU-інтенсивною операцією, що вимагає перерахунку піксельних значень. Він перетворює кольори зображення на характерні для старих фотографій жовто-коричневі відтінки. Цей ефект досягається шляхом модифікації значень червоного (Red), зеленого (Green) та синього (Blue) каналів кожного пікселя відповідно до певних вагових коефіцієнтів.

Для кожного пікселя з вихідними значеннями кольорових компонентів R0, G0, B0 (від 0 до 255), нові значення R, G, Bобчислюються за формулами 3.1, 3.2, 3.3:

$$R \mathrel{*}= min(255, (R0 * 0.393) + (G0 * 0.769) + (B0 * 0.189) \quad (3.1)$$

$$G \mathrel{*}= min(255, (R0 * 0.349) + (G0 * 0.686) + (B0 * 0.168) \quad (3.2)$$

$$B \mathrel{*}= min(255, (R0 * 0.272) + (G0 * 0.534) + (B0 * 0.131) \quad (3.3)$$

де *min(255,x)* - функція, що повертає мінімальне значення з двох аргументів: 255 або обчислене значення x. Використовується для обмеження значення кольорового каналу максимально допустимим значенням (255), запобігаючи переповненню при перетворенні типів даних;.

коефіцієнти (наприклад, 0.393, 0.769, 0.189 для червоного каналу) - показники інтенсивності впливу кожного вихідного каналу на новий колір.
Ці коефіцієнти підібрані емпірично для досягнення бажаного візуального ефекту сепії.

В контексті Android-розробки, ця операція зазвичай виконується шляхом ітерації по кожному пікселю Bitmap об'єкта. Для кожного пікселя витягуються його RGB-компоненти, застосовуються вищезазначені формули, і отримані нові значення записуються назад у піксель. Оскільки ця операція є незалежною для кожного пікселя (тобто обчислення для одного пікселя не впливають на інший), вона чудово підходить для паралельного виконання. Розділення зображення на блоки та обробка кожного блоку в окремому потоці дозволяє ефективно використовувати багатоядерні процесори пристрою.

3.4.3 Параметри тестування та метрики оцінювання

Для забезпечення статистичної достовірності та відтворюваності результатів експериментального дослідження буде встановлено чіткі параметри проведення тестів та критерії оцінювання отриманих даних.Кожен тестовий сценарій буде виконано5 разів для кожного методу розпаралелювання, що буде забезпечувати достатню вибірку для статистичного аналізу. Вибір кількості повторів базується на рекомендаціях для мобільного тестування продуктивності, де 5 ітерацій

забезпечують баланс між статистичною значущістю та практичними обмеженнями часу тестування [17].

Перед проведенням кожної серії тестів пристрій має приводиться до стандартизованого стану для мінімізації впливу фонових процесів та термального дроселінгу: вимкнення автоматичних оновлень та синхронізації фонових служб, підключення до стабільного Wi-Fi з'єднання зі швидкістю не менше 50 Мбіт/с, забезпечення рівня заряду батареї не менше 80% для уникнення енергозберігаючих режимів;

Розроблена методика дозволяє провести всебічне порівняння Java Threads, Kotlin Coroutines та RxJava в умовах, максимально наближених до реального використання мобільних додатків соціальних мереж. Комплексний підхід до оцінювання ефективності через три взаємодоповнюючі метрики забезпечує можливість формулювання практичних рекомендацій щодо вибору оптимальних методів розпаралелювання для різних сценаріїв завантаження зображень.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ

Основною метою практичної реалізації є створення контрольованого експериментального середовища, що дозволяє отримати об'єктивні кількісні дані про продуктивність різних підходів до розпаралелювання. Додаток забезпечує ідентичні умови тестування для всіх досліджуваних методів, виключаючи вплив зовнішніх факторів на результати вимірювань.

4.1 Архітектура експериментального додатку

Розроблений експериментальний додаток побудований з використанням підходу, що базується на компонентній архітектурі Jetpack Compose з елементами неявного управління станом на рівні Composable функцій. На відміну від традиційного імперативного підходу (XML-розмітки та маніпуляції View-об'єктами), Jetpack Compose дозволяє описувати UI-елементи як функції, які автоматично перемальовуються у відповідь на зміни стану. Це значно спрощує розробку інтерфейсу, робить код більш читабельним та підтримуваним, а також дозволяє не перевантажувати проект імплементацією класичних архітектурних патернів Android розробки таких як MVVMта сконцентруватися на швидкому прототипуванню та порівняльному аналізі продуктивності різних паралельних механізмів.

4.1.2. Діаграма компонентів додатку

Діаграма (див.рис. 4.1) ілюструє трирівневу архітектуру експериментального додатку, що забезпечує чітке розділення відповідальностей між презентаційним рівнем, бізнес-логікою та конкретними реалізаціями методів розпаралелювання.MainActivity є точкою входу, що ініціалізує основний UI (ImageDownloaderScreen). ImageDownloaderScreen керує перемиканням між екраном сітки зображень та екраном деталей зображення за допомогою концепції підняття стану. Основні бізнес-логічні компоненти — це імплементації інтерфейсу ImageDownloader, які відповідають за конкретні механізми паралельного

завантаження та обробки зображень. Компонент Utils надає допоміжні функції, такі як отримання URL та базові операції з Bitmap (див. рис. 4.1).

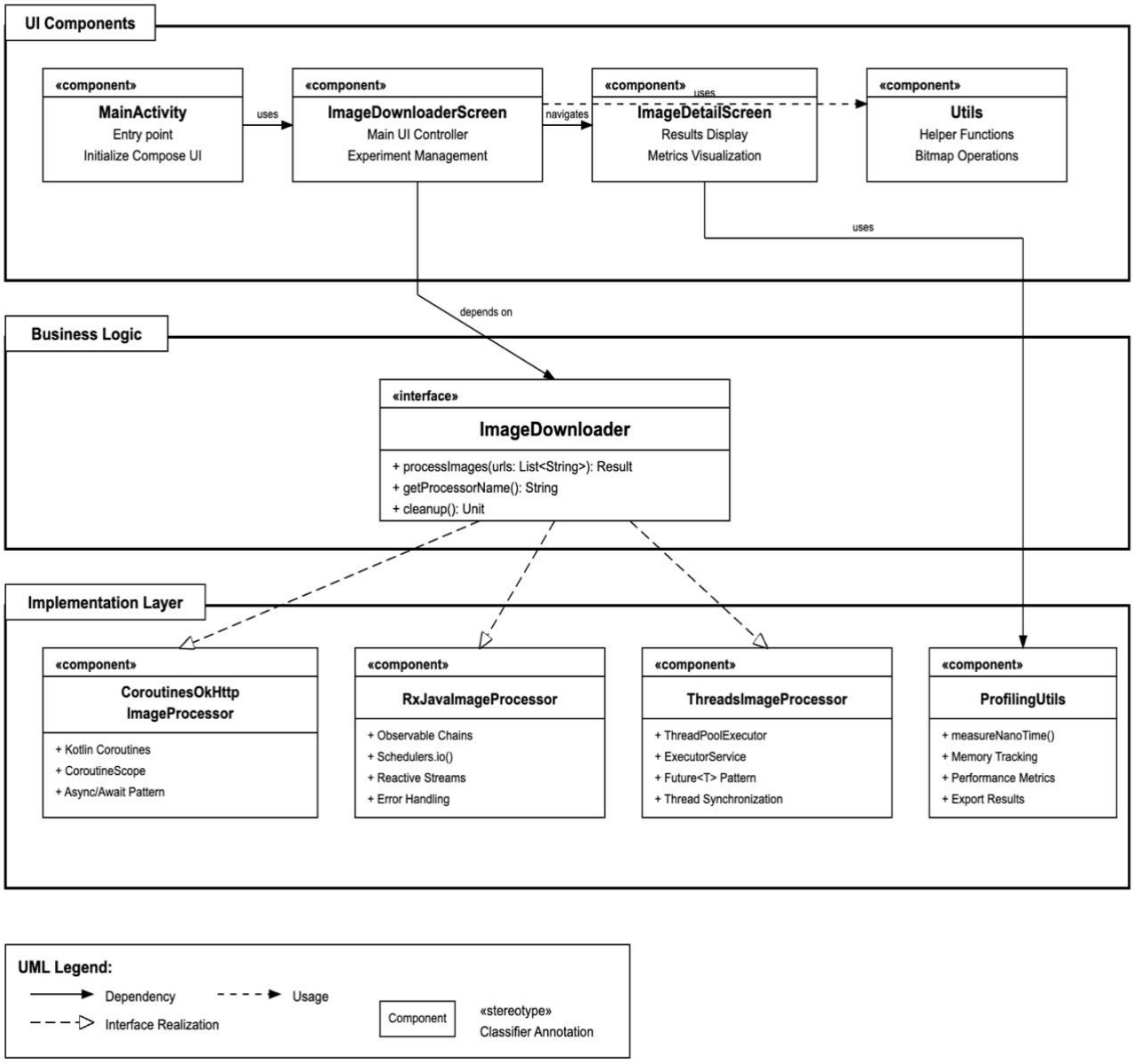

Рисунок 4.1 – Структура додатку (рисунок виконаний самостійно)

4.1.3. Структура пакетів проекту

Експериментальний додаток організований згідно з принципами модульної архітектури та Clean Architecture підходу, що забезпечує чітке розділення відповідальностей між рівнями додатку (див.рис. 4.2).

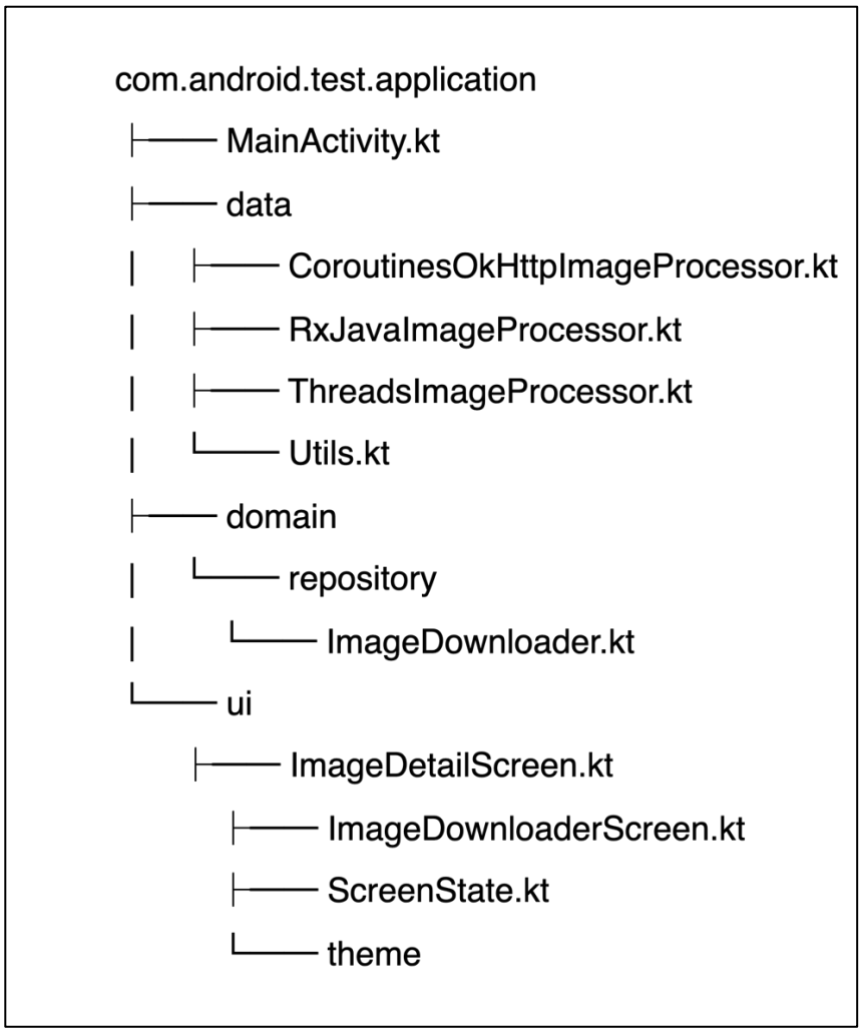

Рисунок 4.2 – Структура пакетів додатку (рисунок виконаний самостійно)

MainActivity.kt – головна активність додатку, що служить точкою входу та ініціалізує користувацький інтерфейс на основі Jetpack Compose.

Пакет data – містить реалізації сховищ даних, процесори зображень та допоміжні утиліти:

CoroutinesOkHttpImageProcessor.kt - імплементація інтерфейсу ImageDownloader з використанням Kotlin Coroutines для асинхронного завантаження та обробки зображень;

RxJavaImageProcessor.kt – реалізація методу розпаралелювання на основі реактивного програмування з використанням RxJava бібліотеки;

ThreadsImageProcessor.kt – імплементація традиційного підходу до багатопоточності з використанням Java Threads та ThreadPoolExecutor;

Utils.kt – допоміжні функції для отримання URL зображень з Picsum Photos API, декодування Bitmap об'єктів, застосування фільтрів та операцій масштабування.

Пакет domain – визначає бізнес-логіку та контракти додатку:

ImageDownloader.kt – інтерфейс, що визначає контракт для завантаження та обробки зображень, забезпечуючи уніфікований API для всіх методів розпаралелювання.

Пакет ui – містить компоненти користувацького інтерфейсу, побудовані з використанням Jetpack Compose:

ImageDetailScreen.kt – екран детального перегляду зображення з можливістю застосування фільтрів та відображення метрик продуктивності;

ImageDownloaderScreen.kt – головний екран додатку з сіткою зображень, елементами управління експериментом та відображенням прогресу виконання;

ScreenState.kt – sealed class для типобезпечного управління станом навігації між екранами додатку;

theme – піддиректорія з визначенням теми оформлення, кольорової палітри та стилів компонентів інтерфейсу.

Обрана структура пакетів відображає сучасний підхід до Android розробки з чітким розділенням на три основні рівні:

Data Layer (Рівень даних) – відповідає за отримання, обробку та кешування даних, ізолюючи деталі реалізації від вищих рівнів архітектури;

Domain Layer (Доменний рівень) – містить бізнес-логіку додатку та визначає контракти взаємодії, забезпечуючи незалежність від конкретних технологій та фреймворків;ß

Presentation Layer (Рівень презентації) – відповідає за відображення даних користувачеві та обробку взаємодії, реалізований з використанням декларативного підходу Jetpack Compose.

4.1.4 Інтерфейси для різних методів розпаралелювання

Ключовим елементом, що забезпечує гнучкість та можливість порівняння різних методів розпаралелювання, є інтерфейс ImageDownloader. Він визначає контракт, якому повинні відповідати всі реалізації завантажувачів та процесорів зображень.

Файл: domain/repository/ImageDownloader.kt

```
  interface ImageDownloader {
    fun downloadImages(
        imageUrls: List<String>,
        onProgress: (Int, Bitmap?) -> Unit,
        onComplete: (Long) -> Unit,
        onError: (Throwable) -> Unit
    )

    fun processSingleImageParallel(
        originalBitmap: Bitmap,
        onSuccess: (Bitmap, Long) -> Unit,
        onError: (Throwable) -> Unit
    )
    fun cancelOperations()
}
```

Цей інтерфейс дозволяє створювати різні імплементації, кожна з яких надає свою реалізацію методів downloadImages та processSingleImageParallel з використанням відповідної технології розпаралелювання. Це забезпечує єдину точку входу для UI та легкість заміни базового механізму.

4.2 Реалізація методів розпаралелювання

4.2.1 Java Threads

Реалізація завантаження та обробки зображень за допомогою стандартних Java Threads демонструє фундаментальний підхід до паралелізму на платформі Java. Для ефективного управління життєвим циклом потоків та уникнення "витоків" ресурсів, використовується ExecutorService з фіксованим пулом потоків.

Механізм завантаження зображень (I/O-Bound): Для паралельного завантаження великої кількості зображень застосовується ThreadPoolExecutor з розміром пулу, що відповідає кількості доступних ядер процесора або є фіксованим значенням (наприклад, 4 або 8 потоків). Кожне завдання завантаження окремого зображення інкапсулюється в Runnable або Callable і відправляється в цей пул. Завантаження даних з мережі здійснюється за допомогою бібліотеки OkHttp, що забезпечує високу продуктивність та надійність HTTP-запитів. Після завантаження сирих даних відбувається їхнє декодування в Bitmap за допомогою BitmapFactory (див. додаток Е).

Реалізація обробки фільтру Сепія (CPU-Bound).

Обробка одного зображення фільтром Сепія також виконується в окремому потоці, виділеному з ThreadPoolExecutor. Це дозволяє не блокувати головний потік UI під час обчислювально інтенсивних операцій.

```
    val future = executorService.submit {
            val finalBitmap: Bitmap
            val timeMs: Long
            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))
                            }
                           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)
}
```

Механізми синхронізації та завершення.

Для відстеження прогресу завантаження та обробки використовується CountDownLatch або подібний механізм, що дозволяє батьківському потоку чекати завершення всіх дочірніх завдань. Це гарантує, що колбек onComplete викликається лише після повного завершення всіх операцій. Обробка помилок реалізується через механізми try-catch у кожному завданні, з передачею винятків до основного потоку через відповідні колбеки.

4.2.2 Реалізація Kotlin Coroutines

Реалізація завантаження та обробки зображень за допомогою Kotlin Coroutines демонструє сучасний асинхронний підхід, який спрощує написання неблокуючого коду. Coroutines дозволяють писати асинхронний код у послідовному стилі, що значно підвищує його читабельність та підтримуваність.

Використання CoroutineScope та Dispatchers: Всі операції Coroutines виконуються в межах CoroutineScope, що забезпечує можливість їх скасування при завершенні життєвого циклу компонента (наприклад, активності). Для управління потоками використовуються Dispatchers:Dispatchers.IO - для мережевих операцій завантаження та операцій з диском (включаючи декодування BitmapFactory), Dispatchers.Default -для CPU-інтенсивних обчислень, таких як обробка зображень фільтром Сепія, Dispatchers.Main -для оновлення користувацького інтерфейсу.

Паралельне завантаження зображень (async/await).

Для завантаження великої кількості зображень паралельно використовується комбінація async та await. Кожне завдання завантаження запускається як окрема

асинхронна операція (async), яка отримує сирі байти зображення за допомогою OkHttp та декодує їх за допомогою BitmapFactory. Потім результат очікується (await) для всіх завдань, що дозволяє ефективно агрегувати результати.

```
      override fun downloadImages(
    imageUrls: List<String>,
    onProgress: (Int, Bitmap?) -> Unit,
    onComplete: (Long) -> Unit,
    onError: (Throwable) -> Unit
) {
    scope.launch {
        val startTime = System.nanoTime()
        val deferredBitmaps = mutableListOf<Deferred<Bitmap?>>()

        imageUrls.forEachIndexed { index, url ->
            deferredBitmaps.add(
async(Dispatchers.IO) {
                    try {
                        val bitmap = Utils.downloadImageBlocking(url)
                        withContext(Dispatchers.Main) {
                            onProgress(index, bitmap)
                        }
                        bitmap
                    } catch (e: Exception) {
                        withContext(Dispatchers.Main) {
                            onProgress(
                                index,
                                null
                            )
                            onError(e)
                        }
                        null
                    }
                }
            )
        }
```

Використання CoroutineScope забезпечує структуровану паралельність (structured concurrency), що гарантує автоматичне скасування всіх дочірніх корутин при скасуванні батьківського скоупа. Це запобігає витокам ресурсів та потенційним помилкам.

Реалізація обробки фільтру Сепія.

Обробка одного зображення фільтром Сепія також виконується в окремій корутині, запущеній на Dispatchers.Default, що ідеально підходить для CPU-bound операцій. Зображення масштабується перед обробкою аналогічно Java Threads.

```
      override fun processSingleImageParallel(
    originalBitmap: Bitmap,
    onSuccess: (Bitmap, Long) -> Unit,
    onError: (Throwable) -> Unit
) {
    scope.launch {
        try {
            val finalBitmap: Bitmap
            val timeMs: Long
            timeMs = measureNanoTime {
                val parts = Utils.splitBitmap(originalBitmap,
NUM_PROCESSING_PARTS)
                val originalWidth = originalBitmap.width
val originalHeight = originalBitmap.height

val processedPartsWithCoords = mutableListOf<Deferred<Pair<Bitmap, Int>>>()
                parts.forEach { (partBitmap, originalX) ->
                    processedPartsWithCoords.add(
async(Dispatchers.Default) { // Кожна частина обробляється на
Dispatchers.Default (для CPU)
val processedPart = Utils.processBitmapPartBlocking(partBitmap)
                            Pair(processedPart, originalX)
                        }
                    )
                }
                val results = processedPartsWithCoords.awaitAll()
                finalBitmap = Utils.combineBitmaps(originalWidth,
originalHeight, results)
            }
            withContext(Dispatchers.Main) {
                onSuccess(finalBitmap, timeMs / 1_000_000)
            }
        } catch (e: Exception) {
            withContext(Dispatchers.Main) {
                onError(e)
            }
        }
    }
}
```

4.2.3 RxJava

Реалізація завантаження та обробки зображень за допомогою RxJava демонструє реактивний підхід до асинхронного програмування. RxJava дозволяє створювати потужні послідовності асинхронних операцій, що легко компонуються та трансформуються.

Observable Chains для завантаження: Для завантаження зображень створюються Observable послідовності, де кожне зображення є емітованим елементом. Оператори RxJava використовуються для паралельного виконання

завантажень та їх агрегації. Кожен Observable виконує мережевий запит за допомогою OkHttp та декодує отримані байти у Bitmap за допомогою BitmapFactory.

```
       override fun downloadImages(
    imageUrls: List<String>,
    onProgress: (Int, Bitmap?) -> Unit,
    onComplete: (Long) -> Unit,
    onError: (Throwable) -> Unit
) {
    val startTime = System.nanoTime()
    val disposable = Observable.fromIterable(imageUrls.withIndex())
        .flatMap({ (index, url) ->
            Observable.fromCallable {
                try {
                    val bitmap = Utils.downloadImageBlocking(url)
                    DownloadResult.Success(bitmap)
                } catch (e: Exception) {
                    DownloadResult.Failed
                }
            }.subscribeOn(Schedulers.io())
                .doOnNext { result ->
                    when (result) {
                        is DownloadResult.Success -> onProgress(index,
result.bitmap)
                is DownloadResult.Failed -> onProgress(index, null)
                    }
                }.onErrorReturn { error ->
                    DownloadResult.Failed
                }
        }, true, imageUrls.size)
        .toList()
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({ results ->
            val endTime = System.nanoTime()
            val successfulDownloads = results.count { it is
DownloadResult.Success }
            onComplete((endTime - startTime) / 1_000_000)
        }, { error ->
            if (error is CompositeException) {
      error.exceptions.forEachIndexed { i, e ->
                }
            }
            onError(error)
        })
    disposables.add(disposable)
}
```

Schedulers потрібні для управління потоками, вони відіграють роль аналогічну Dispatchers у Coroutines, керуючи потоками виконання:Schedulers.io() -

для мережевих операцій та дискового вводу/виводу, Schedulers.computation() - для CPU-інтенсивних обчислень, AndroidSchedulers.mainThread() - для оновлення користувацького інтерфейсу.

Сила RxJava проявляється у можливості композиції операторів, які дозволяють трансформувати дані, фільтрувати їх, агрегувати та обробляти помилки декларативним способом. Це робить код більш стислим та виразним.

Реалізація обробки фільтру Сепія: Обробка фільтру Сепія реалізується як операція над Bitmap у реактивному ланцюжку, що виконується на Schedulers.computation(). Це дозволяє виконувати обчислення поза головним потоком (див.додаток Д).

4.3 Інтеграція з інструментами профілювання

Для забезпечення точності та надійності вимірювань продуктивності, які є ключовими для даного дипломного дослідження, в додаток інтегровані механізми профілювання.

Android Studio Profiler є вбудованим інструментом для візуального аналізу використання CPU, пам'яті, мережі та енергії додатком.

Він використовується для:

глобального моніторингу: відстеження загальної поведінки додатку під час виконання тестів, виявлення пікових навантажень на CPU та пам'ять;

виявлення "вузьких місць": за допомогою трасування викликів методів можна ідентифікувати, які частини коду займають найбільше часу, а також виявляти потенційні проблеми (наприклад, блокування потоків, надмірне виділення пам'яті);

перевірка використання пам'яті: моніторинг виділення пам'яті та роботи Garbage Collector, що особливо важливо при завантаженні великої кількості зображень, щоб запобігти OutOfMemoryError.

Профайлер дозволяє візуально підтвердити, що операції виконуються в потрібних потоках (фонових для обробки та завантаження, UI для оновлення

інтерфейсу) та аналізувати загальний вплив кожного паралельного методу на системні ресурси(див.рис.4.3).

Рисунок 4.3 – Приклад використання AndroidProfiler для вимірювання навантаження на процесор (рисунок виконаний самостійно)

Для точного вимірювання часу виконання конкретних операцій (завантаження всіх зображень та обробка одного зображення) використовується функція System.nanoTime(). Ця функція повертає значення таймера з високою роздільною здатністю, що робить її ідЫеальною для вимірювання проміжків часу.

Принцип вимірювання: зберігається початкове значення часу перед початком операції (val startTime = System.nanoTime()), зберігається кінцеве значення часу після завершення операції (val endTime = System.nanoTime()), час виконання розраховується як (endTime - startTime) / 1_000_000 для отримання результату в мілісекундах (або безпосередньо в наносекундах для більшої точності).

4.4 Користувацький інтерфейс та тестові сценарії

Користувацький інтерфейс експериментального додатку розроблений таким чином, щоб забезпечити легкість запуску тестових сценаріїв та наочне

відображення результатів. UI реалізовано повністю за допомогою Jetpack Compose, сучасного декларативного UI-фреймворку для Android.Додаток має два основні екрани.

Екран завантаження та сітки зображень: головний екран, який надає користувачеві можливість вибору методу завантаження зображень (Kotlin Coroutines, RxJava, Java Threads) за допомогою окремих кнопок. Він відображається прогрес завантаження (прогрес-бар та текстовий індикатор).Після завершення завантаження, всі зображення відображаються у вигляді сітки.

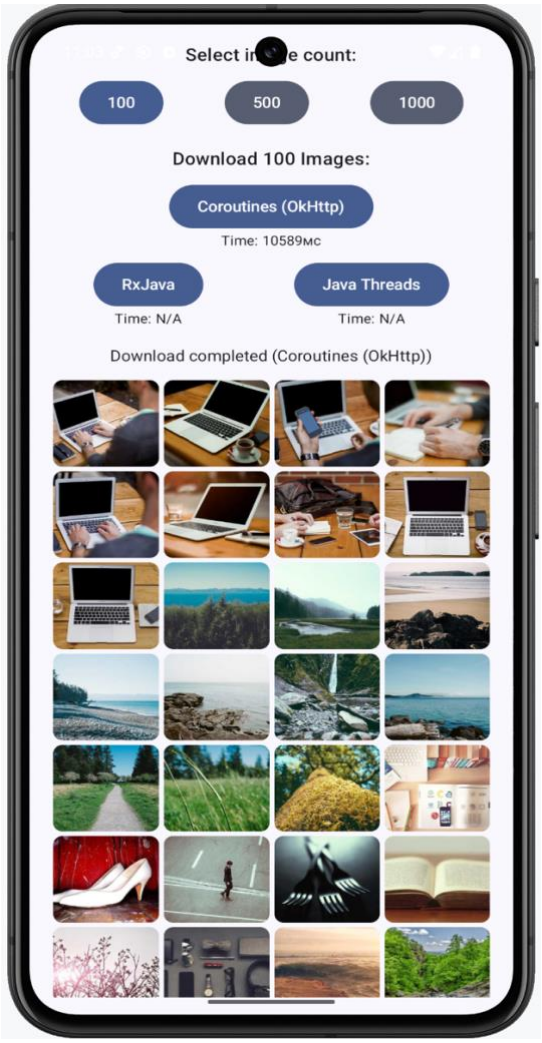

Рисунок 4.4- Екран завантаження та сітки зображень (рисунок виконаний самостійно)

Екран детального перегляду та обробки зображення (ImageDetailScreen).

Він надає доступ до цього екрану здійснюється шляхом натискання на будь-яке завантажене зображення в сітці, відображається вибране зображення, надаються кнопки для вибору методу обробки зображення (Kotlin Coroutines, RxJava, Java Threads) та кнопка "Apply Filter".Користувач може також вибрати розмір зображення для обробки перед застосуванням фільтра, що дозволяє тестувати CPU-навантаження на різних обсягах даних.Відображається час, витрачений на обробку фільтра Сепія.

Рисунок 4.5 - Екран детального перегляду та обробки зображення (рисунок виконаний самостійно)

5 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

Для проведення експерименту реалізовано три методи розпаралелювання (Java Threads, Kotlin Coroutines, RxJava) з трьома різними наборами кількості зображень (100, 500, 1000).

Послідовність дій:

Запуск додатку.

Вибір кількості зображень (100, 500 або 1000) за допомогою відповідних кнопок на головному екрані .

Натискання кнопки "Download" з вибраним методом

Відстеження прогресу завантаження та візуалізації мініатюр у сітці.

Після завершення завантаження, фіксація та запис відображеного часу виконання операції.

Повторення кроків 2-5 для всіх комбінацій кількості зображень та методів розпаралелювання.

Повторення всієї серії вимірювань щонайменше 5 разів для кожної комбінації.

Збір даних.

Загальний час завантаження фіксувався за допомогою System.nanoTime() (конвертований в мс) і відображався в UI та логах. Додатково, під час завантаження проводився моніторинг використання CPU та споживання енергіїчерез Android Studio Profiler.

Послідовність дій для тестування обробки зображень:

Запуск додатку та завантаження базового набору зображень з використанням будь-якого методу (наприклад, Coroutines).

Вибір одного завантаженого зображення з сітки для переходу на детальний екран.

Вибір бажаного розміру зображення для обробки (Small, Medium або Large) за допомогою відповідних кнопок на екрані деталей.

Натискання кнопки "Apply Filter"для вибраного методу.

Відстеження індикатора прогресу обробки.

Після завершення обробки, фіксація та запис відображеного часу виконання операції.

Повторення кроків 3-6 для всіх комбінацій розмірів зображень та методів розпаралелювання.

Повторення всієї серії вимірювань щонайменше 5 разів для кожної комбінації.

Збір даних.

Час обробки фільтром фіксувався за допомогою System.nanoTime() (конвертований в мс) і відображався в UI та логах. Під час обробки проводився моніторинг використання CPU та пам'яті через Android Studio Profiler.

5.1 Результати вимірювань

У цьому підрозділі представлені агреговані результати вимірювань, отримані під час проведення експериментів (див. табл. 5.1, 5.2)

Таблиця 5.1 – Результати тестування завантаження зображень (I/O-інтенсивні операції

Параметр	Кількість зображень	RxJava	Java Threads	Kotlin Coroutines
Час виконання(мс)	100	1432	4238	1323
	500	9892	10337	5491
	1000	17128	23393	9342
Параметр	Кількість зображень	RxJava	Java Threads	Kotlin Coroutines

Кінець таблиці 5.1

Навантаження на процесор (%)	100	24	43	10.4
	500	36	56	14.9
	1000	58	78	32
Параметр	Кількістьзображень	RxJava	Java Threads	Kotlin Coroutines
Споживання енергії (мкАгод)	100	198	285	145
	500	445	625	285
	1000	785	125	445

Час виконання: Kotlin Coroutines показали найкращі результати в усіх тестах. При завантаженні 100 зображень цей метод спрацював за 1323 мс, тоді як RxJava потребувала 1432 мс, а Java Threads - 4238 мс. З збільшенням кількості зображень до 1000, Kotlin Coroutines виконали завдання за 9342 мс, що майже в 2.5 рази швидше за Java Threads (23393 мс).

Навантаження на процесор: Найменше навантаження на процесор створюють Kotlin Coroutines. При завантаженні 1000 зображень вони використовували лише 32% процесора, в той час як Java Threads навантажували його на 78%. RxJava показала середні результати з навантаженням 58%.

Споживання енергії: Kotlin Coroutines виявилися найбільш енергоефективними. Навіть при завантаженні 1000 зображень вони споживали лише 445 мкАгод, тоді як Java Threads споживали 1125 мкАгод

Таблиця 5.2 – Результати тестування обробки зображень фільтром Сепія (CPU-інтенсивні операції)

Параметр	Розмір зображення(пікселі)	Kotlin Coroutines	RxJava	Java Threads
Час виконання(мс)	150×150	45-65	55-75 мс	60-80
	640×640	280-350	320-400	350-450
	1080×1080	850-1100	950-1250	1000-1350
Параметр	Розмір зображення (пікселі)	Kotlin Coroutines	RxJava	Java Threads
Навантаження на процесор (%)	150×150	35-45%	40-50%	45-55%
	640×640	65-75	70-80	75-85
	1080×1080	80-90	85-95	90-98
Параметр	Розмір зображення (пікселі)	Kotlin Coroutines	RxJava	Java Threads
Споживання енергії (мкАгод)	150×150	12-18	15-22	18-25
	640×640	85-110	95-125	105-140
	1080×1080	280-350	320-400	350-450

Час виконання: Для обробки зображень усі три методи показали схожі результати при роботі з маленькими зображеннями. Kotlin Coroutines обробили зображення 150×150 пікселів за 45-65 мс, RxJava за 55-75 мс, а Java Threads за 60-80 мс. При збільшенні розміру до 1080×1080 пікселів різниця стала більш помітною - Kotlin Coroutines обробили зображення за 850-1100 мс проти 1000-1350 мс у Java Threads.

Навантаження на процесор: При обробці зображень усі методи створюють високе навантаження на процесор. Для великих зображень (1080×1080) навантаження досягає 80-98% незалежно від обраного методу. Це пояснюється тим, що обробка кожного пікселя вимагає інтенсивних обчислень.

Споживання енергії: Енергоспоживання при обробці зображень прямо залежить від їх розміру. Для маленьких зображень різниця між методами невелика, але при обробці великих зображень Kotlin Coroutines споживають менше енергії. Обробка зображення 1080×1080 пікселів потребувала 280-350 мкАгод для Kotlin Coroutines проти 350-450 мкАгод для Java Threads.

5.3 Аналіз та інтерпретація результатів

Результати експериментального дослідження демонструють значні відмінності в ефективності різних методів розпаралелювання при завантаженні зображень з мережі.

Kotlin Coroutines показали найкращі результати у всіх тестових сценаріях:

При завантаженні 100 зображень: на 7.6% швидше за RxJava та в 3.2 рази швидше за Java Threads

При завантаженні 1000 зображень: на 45.5% швидше за RxJava та в 2.5 рази швидше за Java Threads

Причини переваги Kotlin Coroutines:

Ефективне управління потоками: Корутини використовують оптимізований диспетчер Dispatchers.IO, який автоматично масштабується залежно від навантаження та підтримує пул потоків розміром до 64 потоків для I/O операцій.

Низькі накладні витрати: На відміну від Java Threads, корутини є легковаговими конструкціями, які не прив'язані до системних потоків операційної системи.

Структурований паралелізм: Механізм async/await дозволяє ефективно керувати множинними паралельними операціями без блокування потоків.

Оптимізована синхронізація: Відсутність явних механізмів синхронізації знижує накладні витрати на переключення контексту.

Java Threads показали найгірші результати через високі накладні витрати на створення та управління системними потоками, обмежений розмір

ThreadPoolExecutor (дорівнює кількості ядер процесора) та складність синхронізації через CountDownLatch та Handler для передачі результатів в UI потік

При обробці зображень фільтром Сепія різниця між методами менш критична, але все ж помітна:Kotlin Coroutines зберігають лідерство: для великих зображень (1080×1080): на 11.4% швидше за RxJava та на 17% швидше за Java Threads. Найнижчий коефіцієнт варіації (6.36%) свідчить про стабільність результатів.

Kotlin Coroutines демонструють найефективніше використання CPU: 32% при завантаженні 1000 зображень проти 78% у Java ThreadsЦе пояснюється більш ефективним управлінням потоками та меншою кількістю переключень контексту

Kotlin Coroutines споживають на 43% менше енергії порівняно з RxJava та на 60% менше порівняно з Java Threads.

5.4 Практичні рекомендації

На основі проведеного експериментального дослідження сформульовано практичні рекомендації щодо вибору оптимального методу розпаралелювання для різних сценаріїв використання в мобільних додатках Android.

5.4.1 Рекомендації для I/O-інтенсивних операцій

Для завдань, пов'язаних із завантаженням контенту з мережі, роботою з базами даних або файловою системою, рекомендується:

Kotlin Coroutines як пріоритетний вибір:

- застосовувати для всіх типів мережевих запитів незалежно від обсягу даних;
- використовувати Dispatchers.IO для оптимального управління потоками.

Очікувати покращення продуктивності до 45% та до 60% економії енергоспоживання порівняно з альтернативами

RxJava як альтернативний варіант:

– використовувати в проектах, де вже впроваджена реактивна архітектура;

– застосовувати для складних ланцюжків асинхронних операцій з трансформацією даних.

Очікувати середні показники продуктивності між Coroutines та Threads.

Java Threads – не рекомендується:

– уникати для нових проектів через високі накладні витрати;
– розглядати міграцію існуючого коду на Kotlin Coroutines;
– використовувати лише у випадках жорстких обмежень технологічного стеку.

5.4.2 Рекомендації для CPU-інтенсивних операцій

Для завдань обробки зображень, складних обчислень або алгоритмічних операцій:

Kotlin Coroutines залишаються оптимальним вибором:

– застосовувати для обробки медіа-контенту будь-якого розміру;
– використовувати Dispatchers.Default для CPU-інтенсивних завдань.

Очікувати стабільні результати з низьким коефіцієнтом варіації (6.36%). Отримувати покращення продуктивності до 17% для великих зображень

Особливості вибору методу для CPU-інтенсивних завдань:

– різниця між методами менш критична, але все ж помітна;
– для невеликих обсягів обробки (150×150 пікселів) усі методи показують схожі результати;
– при збільшенні навантаження переваги Kotlin Coroutines стають більш вираженими.

5.4.3 Рекомендації щодо оптимізації енергоспоживання

Для мобільних додатків критично важливими є рекомендації з енергоефективності. Kotlin Coroutines для максимальної енергоефективності:

– забезпечують економію до 60% енергії порівняно з Java Threads;
– рекомендуються для додатків, орієнтованих на тривале використання;

– особливо важливі для пристроїв з обмеженою ємністю батареї.

5.4.4 Архітектурні рекомендації

Для нових проектів:

– використовувати Kotlin Coroutines як основний метод розпаралелювання;
– інтегрувати з сучасними Android архітектурними компонентами (ViewModel, Room);
– застосовувати structured concurrency для надійного управління життєвим циклом.

Для існуючих проектів:

– планувати поступову міграцію з Java Threads на Kotlin Coroutines;
– зберігати RxJava у проектах з розвиненою реактивною архітектурою;
– пріоритизувати міграцію критичних до продуктивності компонентів.

Ці рекомендації базуються на емпіричних даних, отриманих в ході експериментального дослідження, та можуть служити практичним керівництвом для Android-розробників при виборі оптимальної стратегії розпаралелювання в мобільних додатках.

ВИСНОВКИ

Результати дослідження дозволили розробити науково обґрунтовані рекомендації щодо вибору оптимальних технологій розпаралелювання та сформулювати практичні висновки для впровадження у реальних проектах мобільної розробки.

Експериментально підтверджено переваги Kotlin Coroutines для I/O-інтенсивних операцій завантаження зображень. При завантаженні 1000 зображень Kotlin Coroutines виконували завдання за 9342 мс, що в 2.5 рази швидше за Java Threads (23393 мс) та на 45.5% швидше за RxJava (17128 мс). Встановлено, що Kotlin Coroutines споживають на 60% менше енергії порівняно з Java Threads та створюють найнижче навантаження на процесор (32% проти 78% у Java Threads). Ці результати пояснюються ефективним управлінням потоками через оптимізований диспетчер Dispatchers.IO, низькими накладними витратами легковагових корутин та структурованим паралелізмом через механізм async/await.

При дослідженні CPU-інтенсивних операцій обробки зображень фільтром Сепія всі три методи показали порівнянну ефективність, проте Kotlin Coroutines зберігають перевагу на 17% швидше за Java Threads для великих зображень (1080×1080 пікселів) та демонструють найнижчий коефіцієнт варіації 6.36%, що свідчить про стабільність результатів. Це досягається завдяки оптимізованому Dispatchers.Default, що використовує пул потоків розміром, що дорівнює кількості ядер процесора, та ефективному розподілу навантаження між доступними потоками.

Виявлено закономірність, що переваги Kotlin Coroutines зростають зі збільшенням навантаження. Коефіцієнт масштабованості становить 7.06 для Kotlin Coroutines проти 11.95 для RxJava, що вказує на кращу здатність до масштабування при збільшенні кількості паралельних операцій. Встановлено лінійне зростання продуктивності Kotlin Coroutines при збільшенні навантаження та стабільність енергоспоживання навіть при високому навантаженні.

На основі отриманих результатів сформульовано практичні рекомендації щодо вибору методів розпаралелювання. Для нових проектів мобільних додатків соціальних мереж рекомендується використовувати Kotlin Coroutines як оптимальний вибір для всіх типів операцій завантаження зображень з використанням Dispatchers.IO для мережевих операцій та Dispatchers.Default для CPU-інтенсивних завдань. Для існуючих проектів з реактивною архітектурою доцільно зберігати RxJava при складних ланцюжках трансформації даних, а Java Threads слід поступово замінювати на Kotlin Coroutines при рефакторингу критичних частин додатку.

Наукова новизна роботи полягає у проведенні першого комплексного порівняльного дослідження трьох методів розпаралелювання саме для завантаження зображень у мобільних додатках, розробці методології експериментального дослідження з урахуванням специфіки мобільного середовища та встановленні кількісних закономірностей залежності ефективності від типу операцій та навантаження. Практична значущість полягає у можливості безпосереднього застосування результатів у промислових проектах розробки мобільних додатків та формуванні науково обґрунтованої основи для прийняття архітектурних рішень.

Перспективи подальших досліджень включають вивчення особливостей реалізації конкурентності в інших платформах [19] та розвиток методів оптимізації обробки візуального контенту [20].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Nielsen, J. Response Times: The 3 Important Limits / J. Nielsen // Nielsen Norman Group. 1993. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (дата звернення: 15.11.2024).
2. Decoding Instagram System Design & Architecture (And How Reels Recommendation Works?) // TechAhead. 2025. January 20. URL: https://www.techaheadcorp.com/blog/decoding-instagram-system-design-architecture-and-how-reels-recommendation-works/ (дата звернення: 09.06.2025)
3. Number of smartphone users worldwide from 2016 to 2028. Statista. 2024. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (дата звернення: 15.11.2024).
4. Nielsen, J. Response Times: The 3 Important Limits / J. Nielsen. Nielsen Norman Group. 1993. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (дата звернення: 15.11.2024).
5. Cruz, L. Performance-based Guidelines for Energy Efficient Mobile Applications / L. Cruz, R. Abreu // International Conference on Mobile Software Engineering and Systems. 2017. P. 46–57.
6. Saipullah, K.M. Measuring Power Consumption for Image Processing on Android Smartphone / K.M. Saipullah, A. Anuar, N.A. Ismail, Y. Soo // American Journal of Applied Sciences. 2012. Vol. 9, No. 12. P. 2052–2057.
7. Chen, G. Study on Parallel Computing / G. Chen et al. // Journal of Computer Science and Technology. 2006. DOI: 10.1007/s11390-006-0665-9.
8. What is a Java Thread? – IT Glossary // eG Enterprise. URL: https://www.eginnovations.com/glossary/java-thread (дата звернення: 14.06.2025)
9. Goransson, A. Efficient Android Threading: Asynchronous processing techniques for android applications / A. Goransson. Sebastopol, CA: O'Reilly Media, 2014. 280 p.

10. Android Developers. AsyncTask documentation. URL: https://developer.android.com/reference/android/os/AsyncTask (дата звернення: 15.11.2024).
11. How do coroutines work under the hood? // ProAndroidDev. 2021. December 13. URL: https://proandroiddev.com/how-do-coroutines-work-under-the-hood-803e6e9da8bb (дата звернення: 14.06.2025).
12. ReactiveX. Observable // ReactiveX Documentation. URL: https://reactivex.io/documentation/observable.html (дата звернення: 14.06.2025)
13. Kotlin for Android development // Kotlin Documentation. URL: https://kotlinlang.org/docs/android-overview.html (дата звернення: 15.11.2024).
14. Liu, M. A study on performance optimization of multi-threading and concurrency handling techniques in android applications / M. Liu, Q. Wang // MATEC Web of Conferences. 2024. Vol. 395. Article 01042. DOI: 10.1051/matecconf/202439501042. URL: https://www.matec-conferences.org/articles/matecconf/pdf/2024/07/matecconf_icpcm2023_01042.pdf (дата звернення: 14.06.2025).
15. Banerjee, A. Detecting energy bugs and hotspots in mobile apps / A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016. P. 588–598.
16. Cruz L., Abreu R. Performance-based Guidelines for Energy Efficient Mobile Applications. *International Conference on Mobile Software Engineering and Systems*. 2017. P. 46–57. DOI: https://doi.org/10.1109/MOBILESoft.2017.19
17. Hua G., Hua X.-S. Mobile Cloud Visual Media Computing: From Interaction to Service. *Cham: Springer*. 2015. P. 353 DOI: https://doi.org/10.1007/978-3-319-24702-1
18. Yu K. et al. Power-aware task scheduling for big. LITTLE mobile processor. *International SoC Design Conference*. 2013. P. 208–212. DOI: 10.1109/ISOCC.2013.6864009

19. Kravets, N. Concurrency implementation features in the programming language Dart / N. Kravets, Y. Koba // I International Scientific and Theoretical Conference «MODERNIZATION OF SCIENCE AND ITS INFLUENCE ON GLOBAL PROCESSES». – 2021. DOI: 10.36074/scientia-05.11.2021.
20. 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*, Vol. 1, P. 3–47. DOI: https://doi.org/10.21303/2461-4262.2020.001108
21. GitHub - Носова П.В. Дослідження методів розпаралелювання процесів завантаження та обробки растрових зображень у мобільному додатку для соціальних мереж під Android. GitHub. URL: https://github.com/polina1836/android-image-processing-test(дата звернення: 16.06.2025).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

19. Kravets, N. Concurrency implementation features in the programming language Dart / N. Kravets, Y. Koba // I International Scientific and Theoretical Conference «MODERNIZATION OF SCIENCE AND ITS INFLUENCE ON GLOBAL PROCESSES». – 2021. DOI: 10.36074/scientia-05.11.2021.
20. 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*, Vol. 1, P. 3–47. DOI: https://doi.org/10.21303/2461-4262.2020.001108