МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Кваліфікаційна наукова
праця на правах рукопису

ШЕВЧЕНКО АНДРІЙ КОСТЯНТИНОВИЧ

УДК 004.932:004.272 (043.3)

ДИСЕРТАЦІЯ
ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОБРОБКИ ГРАФІЧНИХ ДАНИХ ДЛЯ SIMD-АРХІТЕКТУР ПРОЦЕСОРІВ

Спеціальність 05.13.06 – «Інформаційні технології»

Подається на здобуття наукового ступеня кандидата технічних наук

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело



_____ А.К. Шевченко

Науковий керівник:
Приставка Пилип Олександрович
доктор технічних наук, професор

Київ – 2026

АНОТАЦІЯ

Шевченко А. К. Інформаційна технологія обробки графічних даних для SIMD-архітектур процесорів. — Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.06 «Інформаційні технології». — Національний університет «Київський авіаційний інститут», Київ, 2026.

Дисертаційну роботу присвячено розробці інформаційної технології (ІТ) підвищення швидкодії обробки 8-бітних цифрових RGB-зображень на системах з архітектурою ARM64. Запропонована ІТ базується на використанні оптимізованої векторизованої операції згортки із застосуванням 16-бітних цілочисельних обчислень та SIMD-інструкцій NEON для ЦП архітектури ARM64. Актуальність роботи зумовлена поширенням ARM64-платформ у вбудованих системах та мобільних пристроях, зокрема в безпілотних літальних апаратах, що потребують високої швидкодії обробки зображень в умовах обмежених ресурсів та вимог реального часу.

У роботі вперше розроблено та теоретично обґрунтовано метод векторизованої операції згортки, що базується на 16-бітних цілочисельних операціях. Запропоновано формалізований підхід до побудови векторизованих згорткових операторів, розроблено обчислювальне ядро conv_RGB_NEON та реалізовано модульну ІТ у вигляді С-бібліотеки з Python-інтерфейсом. Проведено оптимізацію обчислень із врахуванням кеш-ефективних структур коду, обробки розріджених ядер згортки та розроблено нотацію для опису SIMD NEON64 операцій, яка є основою опису векторизованої операції згортки.

Експериментальні дослідження на одноплатних ARM64-комп'ютерах (Raspberry Pi 4, Orange Pi 5 Pro) показали значне прискорення виконання операції згортки (до 11,1 раза на Orange Pi 5 Pro та до 10,2 раза на Raspberry Pi 4, зокрема до 2,5 раза для малих ядер 2×2–7×7) порівняно з функцією cv::filter2D бібліотеки OpenCV. При цьому точність обробки залишається високою, з максимальною абсолютною похибкою не більше 1 (одиниці) на кожен колірний канал 8-бітного RGB цифрового зображення, що візуально не впливає на якість зображення. Розроблена ІТ адаптована для обробки відео в реальному часі (досягнуто стабільної швидкодії 26–29 кадрів/секунду при обробці відеопотоку з дрона на Orange Pi 5 Pro).

Практична значущість роботи полягає у створенні готового програмного інструментарію для розробників систем, що потребують високопродуктивної обробки зображень на платформах ARM64, зокрема в системах бортової обробки даних БПЛА, мобільних додатках, системах відеоспостереження та edge-computing. Результати дослідження впроваджено: при виконанні науково-дослідних робіт подвійного призначення НДР №1062-ДБ16, №247-ДБ19 та №479-ДБ24 на базі Національного університету «Київський авіаційний інститут»; у навчальному процесі Національного університету «Київський авіаційний інститут» при викладанні дисциплін «Архітектура обчислювальних систем та операційні системи», «Системне програмування» та «Програмування мовою асемблер»; у виробничій діяльності ТОВ «ЕЛІСАТ» (вбудоване програмне забезпечення пристроїв збору й обробки даних з датчиків) та ТОВ «ІНСПІРІТО ТІМ» (програмно-апаратні засоби обробки відеоданих з бортових камер БПЛА).

Ключові слова: SIMD, ARM64, NEON64, операція згортки, 16-бітні обчислення, обробка цифрових зображень, векторизація, швидкодія, інформаційна технологія, реальний час, одноплатні комп'ютери, БПЛА.

ABSTRACT

Shevchenko A. K. Information technology of graphic data processing for processor SIMD architectures. — Manuscript.

Thesis for a Candidate of Technical Science degree in specialty 05.13.06 «Information technologies». – National University «Kyiv Aviation Institute», Kyiv, 2026.

The dissertation is dedicated to the development of an information technology (IT) for increasing the speed of processing 8-bit digital RGB images on systems with the ARM64 architecture. The proposed IT is based on the use of an optimized vectorized convolution operation employing 16-bit integer calculations and SIMD NEON instructions for ARM64 architecture CPUs. The relevance of the work is driven by the widespread use of ARM64 platforms in embedded systems and mobile devices, particularly in unmanned aerial vehicles, which require high-performance image processing under conditions of limited resources and real-time demands.

The purpose of the work is to solve the scientific and technical problem of developing an information technology for processing digital images and video by means of a high-speed convolution operation for ARM64 architecture CPUs, which enables real-time filtering and scaling of streaming video and digital images on single-

board computers with limited computing resources.

The scientific novelty of the obtained results is as follows. For the first time, a method of the high-speed convolution operation for processing 8-bit RGB digital images using 16-bit integer operations has been developed and theoretically substantiated, which increases performance on ARM64 architecture CPUs compared with the analogous approach based on the cv::filter2D function of the OpenCV library. For the first time, a formalized notation for describing vectorized computations using SIMD NEON64 has been proposed, which allows the vectorized part of the code to be represented as structured expressions, thereby improving readability and lowering the entry threshold for mastering expert vectorization techniques. The technology of digital image processing on computers supporting computational parallelism has been improved by means of the proposed information technology of image and video processing based on the high-speed convolution operation for the ARM64 architecture, which enables real-time filtering and scaling of streaming video and digital images on single-board computers with ARM64 CPUs, such as Raspberry Pi 4 and Orange Pi 5 Pro. Linear operators based on linear combinations of B-splines that are close to interpolation in the mean have been further developed, namely: linear operators of 16-bit integer operations have been obtained that guarantee the absence of overflow in the 16-bit computation of the vectorized dot product, which allows their application to subband filtering, contrast enhancement, digital image stabilization and recurrent twofold enlargement of the linear dimensions of digital images.

A formalized approach to constructing vectorized convolution operators has been proposed, the conv_RGB_NEON computational core has been developed, and a modular IT has been implemented as a C-library with a Python interface. Optimization of computations has been carried out considering cache-efficient code structures and the processing of sparse convolution kernels.

Experimental studies on single-board ARM64 computers (Raspberry Pi 4, Orange Pi 5 Pro) showed a significant acceleration of the convolution operation (up to 11.1 times on Orange Pi 5 Pro and up to 10.2 times on Raspberry Pi 4, particularly up to 2.5 times for small 2×2–7×7 kernels) compared to the cv::filter2D function of the OpenCV library. The processing accuracy remains high, with a maximum absolute error not exceeding 1 (unit) per color channel of an 8-bit RGB digital image, which does not visually affect image quality. The developed IT is adapted for real-time video processing (stable performance of 26–29 frames/second achieved when processing

video stream from a drone on Orange Pi 5 Pro).

The practical significance of the work lies in the creation of a ready-to-use software toolkit for developers of systems requiring high-performance image processing on ARM64 platforms, particularly in UAV onboard data processing systems, mobile applications, video surveillance systems, and edge-computing. The research results have been implemented: within dual-purpose R&D projects No. 1062-DB16, No. 247-DB19, and No. 479-DB24 at the National University “Kyiv Aviation Institute”; in the educational process of the National University “Kyiv Aviation Institute” in teaching the courses “Computer Systems Architecture and Operating Systems”, “System Programming”, and “Assembly Language Programming”; and in the operational activities of ELISAT LLC (embedded software for sensor data acquisition and processing devices) and INSPIRITO TIM LLC (hardware and software systems for processing video data from UAV onboard cameras).

Keywords: SIMD, ARM64, NEON64, convolution operation, 16-bit computing, digital image processing, vectorization, high performance, information technology, real-time, single-board computers, UAV.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Статті у наукових фахових виданнях України:

1. Приставка П. О., Шевченко А. К. Дослідження реалізації лінійного оператора згортки цифрового зображення при 16-бітних обчисленнях. *Актуальні проблеми автоматизації та інформаційних технологій.* 2016. Т. 20: 1. С. 78–90. URL: https://doi.org/10.15421/431608
2. Шевченко А. К. Вибір програмно-апаратних засобів для реалізації операції згортки з підвищеною швидкістю. *Наукоємні технології.* 2018. Т. 37: 1. С. 49–54. URL: https://doi.org/10.18372/2310-5461.37.12369
3. Шевченко А. К. Підвищення швидкодії операції згортки RGB зображень з використанням алгоритму кластеризації ядер згортки для архітектури процесорів ARM64. *Problems of Informatization and Management.* 2025. Т. 1: 81. С. 140–149. URL: https://doi.org/10.18372/2073-4751.81.20144

Статті в іноземних виданнях:

4. Shevchenko A., Tymchyshyn V. A SIMD-based approach to the enhancement of convolution operation performance. *CEUR Workshop Proceedings.* 2019. Vol. 2588. P. 447–458. URL: https://ceur-ws.org/Vol-2588/paper37.pdf (Scopus) Q4, ISSN 1613-0073
5. Shevchenko A., Prystavka P., Tymchyshyn V. Research on possible convolution operation speed enhancement via AArch64 SIMD. *Advances in Computer Science for Engineering and Education. Lecture Notes on Data Engineering and Communications Technologies.* Cham : Springer, 2022. Vol. 134. P. 61–75. URL: https://doi.org/10.1007/978-3-031-04812-8_6 (Scopus, Web of Science) Q3, ISSN 2367-4512
6. Prystavka P., Shevchenko A., Rokitianska I. Comparative analysis of detector-tracker architecture for object tracking based on SBC for UAV. *2024 IEEE 7th International Conference on Actual Problems of Unmanned Aerial Vehicles Development (APUAVD)*, Kyiv, Ukraine, 22–24 October 2024. 2024. P. 175–178. URL: https://doi.org/10.1109/APUAVD64488.2024.10765890 (SCOPUS, IEEE)
7. Shevchenko A., Prystavka P. Enhancement of convolution operation performance using SIMD of AArch64 (short paper). *CEUR Workshop Proceedings.* 2024. Vol. 3826. P. 385–391. URL: https://ceur-ws.org/Vol-3826/short32.pdf (Scopus) Q4, ISSN 1613-0073

Публікації, які додатково відображають наукові результати дисертації:

8. Приставка П. О., Шевченко А. К. Особливості застосування технології ARM NEON при обробці зображень. *Політ. Сучасні проблеми науки : XVI Міжнар. наук.-практ. конф. здобувачів вищ. освіти і молодих учених*, м. Київ, 25 берез. 2016 р. НАУ, 2016. С. 135.
9. Шевченко А. К., Приставка П. О. Оптимізація обробки цифрового зображення з використанням технології ARM NEON. *АВІА-2017 : XIII Міжнар. наук.-техн. конф.*, м. Київ, 19–21 квіт. 2017 р. НАУ, 2017. С. 2.27–2.29.
10. Prystavka P. O., Shevchenko A. K. A brief overview of modern software optimization approaches. *Авіація у XXI столітті – Безпека в авіації та космічні технології : VIII Всесвіт. конгр.*, м. Київ, 10–12 жовт. 2018 р. НАУ, 2018. С. 2.3.22–2.3.24.
11. Шевченко А. К. Огляд програмних засобів для підвищення швидкодії операції згортки. *Політ. Сучасні проблеми науки : XIX Міжнар. наук.-практ. конф. здобувачів вищ. освіти і молодих учених*, м. Київ, 1–5 квіт. 2019 р. НАУ, 2019. С. 35–37.
12. Shevchenko A. K. Cache-friendly code for convolution operation. *Політ. Сучасні проблеми науки : XXI Міжнар. наук.-практ. конф. здобувачів вищ. освіти і молодих учених*, м. Київ, 5–9 квіт. 2021 р. НАУ, 2021. С. 18–19.
13. Приставка П. О., Шевченко А. К. Дослідження швидкодії операції згортки що було оптимізовано за допомогою SIMD NEON ARM64 в умовах обробки відео з файлу. *АВІА-2023 : XVI Міжнар. наук.-техн. конф.*, м. Київ, 18–20 квіт. 2023 р. НАУ, 2023. С. 22–24.
14. Шевченко А. К., Приставка П. О. Дослідження можливостей підвищення швидкодії операції згортки за допомогою кластеризації розріджених ядер матриць згортки на архітектурі AArch64. *Політ. Сучасні проблеми науки : XXIII Міжнар. наук.-практ. конф. здобувачів вищ. освіти і молодих учених*, м. Київ, 4–7 квіт. 2023 р. НАУ, 2023. С. 18–21.

ЗМІСТ

ВСТУП 12

РОЗДІЛ 1. АНАЛІЗ СТАНУ ТА ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ АЛГОРИТМУ ЛІНІЙНОЇ ОПЕРАЦІЇ ЗГОРТКИ З ВИКОРИСТАННЯМ SIMD ОПТИМІЗАЦІЙ ПРИ ОБРОБЦІ ЦЗ ЗА УМОВИ 16-БІТНИХ ЦІЛОЧИСЕЛЬНИХ ОБЧИСЛЕНЬ 20

1.1 Короткий огляд сучасної оптимізації програмного забезпечення 23

1.2 Прискорення за допомогою апаратного забезпечення 24

1.2.1 SIMD NEON ARM64 25

1.2.2 SIMD - AVX 26

1.2.3 GPU – SIMT (майже MIMD) 27

1.2.4 DSP. Новітній DSP Hexagon від фірми Qualcomm 28

1.2.5 Суперкомп'ютерні архітектури на базі ARM64: NVIDIA Grace 30

1.3 Оптимізація за допомогою програмного забезпечення 31

1.3.1 Компілятори 31

1.3.2 CUDA 33

1.3.3 OpenCL 34

1.3.4 Бібліотеки OpenCV та ACL (Arm Compute Library) . . . 35

1.3.5 Ігровий двигун id Tech 3 39

1.3.6 Розгортання циклів 41

1.3.7 Кеш-френдлі оптимізація коду 47

1.4 Узагальнений огляд SIMD-реалізації cv::filter2D() на основі OpenCV Universal Intrinsics 49

1.5 Висновки 50

РОЗДІЛ 2. ЛІНІЙНІ ОПЕРАТОРИ ДЛЯ ОБРОБКИ ЦИФРОВИХ ЗОБРАЖЕНЬ ПІД ЧАС 16-БІТНИХ ОБЧИСЛЕНЬ 52

2.1 Умова відповідності маски фільтра згортки для 16-бітних обчислень 52

2.2 Фільтрація та масштабування зображень 53

2.3 Вплив на швидкодію ОЗ знакових та беззнакових обчислень . . 58

2.3.1 Відсутність необхідності аналізу знаку 58
2.3.2 Спрощення логіки обчислень 59
2.3.3 Уникнення умовних переходів 59
2.3.4 Відмова від використання регістру стану ЦП для підвищення швидкодії СО: 60
2.3.5 Спрощення інструкцій SIMD: 61
2.3.6 Зменшення кількості інструкцій: 62
2.4 Адаптація МЗ для беззнакових обчислень з компенсаційною нормалізацією результатів ОЗ 65
2.5 Оптимізація обчислень, де необхідно перевіряти потрапляння в діапазон від 0 до 255 68
2.6 Висновки 70

РОЗДІЛ 3. ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОБРОБКИ ЦИФРОВИХ ЗОБРАЖЕНЬ ІЗ ЗАСТОСУВАННЯМ ВЕКТОРИЗОВАНОЇ ОПЕРАЦІЇ ЗГОРТКИ НА ПЛАТФОРМІ ARM64 72
3.1 Формалізована нотація як універсальний науковий метод опису векторизованих обчислень. 72
3.1.1 Загальний огляд формалізованої нотації. 73
3.1.2 Зіставлення із наявними засобами опису SIMD-обчислень. 75
3.1.3 Застосування нотації для опису обчислювального ядра ШОЗ. 75
3.1.4 Розширені можливості застосування. 76
3.1.5 Теоретична значущість і методологічний внесок. 77
3.2 Концепція інформаційної технології для високошвидкодійної обробки ЦЗ 78
3.3 Архітектура інформаційної технології 79
3.3.1 Структура Python-застосунку 82
3.3.2 Структура C-бібліотеки (libconv_lib.so) 84
3.3.3 Взаємодія між компонентами 86
3.4 Програмна реалізація ядра інформаційної технології 88
3.4.1 Реалізація бібліотеки libconv_lib 88
3.4.2 Інтерфейс функції conv_RGB_NEON 88
3.4.3 Особливості програмної реалізації 89
3.4.4 Компіляція бібліотеки для цільових платформ 90

3.4.5 Порівняння швидкодії з аналогами 90
3.5 Методика застосування ІТ для обробки різних типів ЦЗ 91
3.5.1 Обробка статичних зображень 91
3.5.2 Обробка відеопотоків 92
3.5.3 Масштабування зображень з використанням ядра ІТ . . 93
3.5.4 Використання різних типів ядер згортки. 93
3.5.5 Комплексна обробка ЦЗ. 94
3.6 Розробка програмного інтерфейсу для ІТ 94
3.6.1 Python-обгортка для взаємодії з бібліотекою libconv_lib. 95
3.6.2 Інтеграція з NumPy для ефективної обробки масивів даних. 96
3.6.3 Реалізація CLI-інтерфейсу в ІТ 96
3.6.4 Передача даних між Python та C-бібліотекою 97
3.6.5 Утилітарні функції програмного інтерфейсу 98
3.6.6 Особливості та переваги розробленого програмного інтерфейсу 98
3.7 Комплексне застосування ІТ для задач обробки ЦЗ 99
3.7.1 Використання ІТ для реалізації складних операцій з масштабуванням зображень 99
3.7.2 Послідовне застосування кількох ядер згортки для досягнення складних ефектів 100
3.7.3 Оптимізація обробки для різних розмірів ядер згортки . 101
3.7.4 Обмеження та рекомендації щодо використання ІТ . . . 102
3.7.5 Приклади комплексного застосування ІТ для типових задач ОЦЗ 103
3.8 Висновки 104

РОЗДІЛ 4. ДОСЛІДЖЕННЯ ШВИДКОДІЇ АЛГОРИТМУ ЛІНІЙНОЇ ОПЕРАЦІЇ ЗГОРТКИ З ВИКОРИСТАННЯМ SIMD-ОПТИМІЗАЦІЙ ПРИ ОБРОБЦІ ЦИФРОВОГО ЗОБРАЖЕННЯ ЗА УМОВИ ВИКОРИСТАННЯ 16-БІТНИХ ОБЧИСЛЕНЬ 106
4.1 Опис експерименту 106
4.2 Прилади та апаратне забезпечення 107
4.3 Програмне налаштування апаратного забезпечення 108
4.4 Загальний опис програмного забезпечення (ОС тощо) 109
4.4.1 Середовище розробки та управління проектом 109

4.4.2 Компілятори, інтерпретатор та середовище віртуалізації та емуляції. 109
4.4.3 Інструменти для оптимізації ядра інформаційної технології 110
4.5 Загальний опис тестування швидкодії інформаційної технології за допомогою Shell-скриптів. 111
4.5.1 Базовий алгоритм тестування швидкодії. 112
4.5.2 Етапи тестування ядра інформаційної технології у Shell. 115
4.5.3 Методика тестування швидкодії інформаційної технології 118
4.6 Аналіз показників швидкодії векторизованої операції згортки цифрового зображення. 125
4.6.1 Аналіз швидкодії векторизованої операції згортки цифрового зображення 126
4.6.2 Аналіз швидкодії Python-інтерфейсу ШОЗ ЦЗ. 131
4.6.3 Аналіз швидкодії операцій масштабування ЦЗ. 134
4.7 Висновки 137

ВИСНОВКИ 139

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 144

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Україномовні позначення

АВ – автоматична оптимізація/векторизація

ВОЗ – векторизована операція згортки

ВР – векторний регістр NEON64

ВСД – векторизований скалярний добуток

КН – коефіцієнт нормалізації

ОЗ – операція згортки

ОЦЗ – обробка цифрових зображень

ПЗ – програмне забезпечення

ПМ – предікт-матриця

ПП – програмні продукти

ППО – платформи для паралельних обчислень

СД – скалярний добуток

ЦЗ – цифрове зображення (якщо не зазначено інакше, мається на увазі RGB-зображення)

ЦП – центральний процесор

ШОЗ – швидкодіюча операція згортки

ЯЗ – ядро згортки

ЯІТ – ядро інформаційної технології

ІТ – інформаційна технологія

Англомовні позначення

`conv_RGB_NEON(…)` – розроблене ядро ІТ

CNN – convolutional neural network (згорткові нейронні мережі)

IDE – Integrated Development Environment (середовище розробки ПЗ)

NEON64 – SIMD-розширення NEON64 архітектури ARM64

SBC – single board computer (одноплатний комп’ютер)

SIMD – single instruction, multiple data

ВСТУП

Актуальність. Обробка цифрових зображень (ЦЗ) та відеоконтенту є невід'ємною частиною сучасного інформаційного світу і застосовується в найрізноманітніших сферах: від побутових додатків у смартфонах до критично важливих систем у промисловості, медицині, безпеці та обороні. Операція згортки (ОЗ) є основою багатьох алгоритмів обробки ЦЗ, впливаючи на їхню ефективність та швидкість. З огляду на експоненційне зростання обсягів візуальних даних і вимог до їх аналізу в реальному часі, дослідження, спрямовані на підвищення швидкодії обробки ЦЗ, набувають особливої актуальності. Світові тенденції демонструють активний перехід до використання пристроїв IoT для локальної обробки даних (edge computing). Це зумовлено потребою у зменшенні затримок, підвищенні надійності та забезпеченні конфіденційності даних. Однак такі платформи, зокрема на базі архітектури ARM64, що домінує в мобільних та вбудованих пристроях, зазвичай мають обмежені обчислювальні ресурси. Стандартні підходи до обробки ЦЗ не завжди оптимально використовують специфіку архітектури ARM64 та її SIMD-розширення NEON64, що призводить до низької швидкодії.

Запропонована в дисертації інформаційна технологія (ІТ) підвищення швидкодії обробки цифрових RGB-зображень шляхом оптимізації ОЗ з використанням 16-бітних цілочисельних операцій та SIMD-інструкцій NEON64 для архітектури ARM64 є унікальною, оскільки поєднує ці підходи, що досі не було широко досліджено в контексті ОЗ ЦЗ.

Актуальність цього дослідження для України сьогодні є надзвичайно високою. В умовах сучасних викликів ефективна обробка візуальної інформації на пристроях з процесорами архітектури ARM64 має критичне значення для сфери безпеки та оборони. Це стосується систем відеоспостереження, аналізу супутникових знімків і, особливо, обробки даних з безпілотних літальних апаратів (БПЛА). Запропонована ІТ дозволяє підвищити продуктивність обробки ЦЗ на бортових комп'ютерах БПЛА, які часто базуються на енергоефективних ARM-процесорах. Це сприятиме зменшенню затримок під час аналізу даних у реальному часі, підвищенню автономності БПЛА в завданнях виявлення та розпізнавання об'єктів (у БПЛА/дронах із машинним зором). Така оптимізація є важливою для створення інтелектуальніших та енергоефективніших безпілотних

систем для потреб Збройних Сил України.

Зв'язок роботи з науковими програмами, темами та планами.

Розвиток цієї технології узгоджується зі стратегічними цілями України щодо цифрової трансформації та інноваційного розвитку до 2030 року, зокрема зі Стратегією цифрового розвитку інноваційної діяльності України (WINWIN) до 2030 року, яка визначає розвиток безпілотних технологій та штучного інтелекту пріоритетними напрямами. Це також відповідає державним програмам підтримки виробництва дронів та розвитку оборонно–промислового комплексу (ОПК). Розробка вітчизняного програмного забезпечення, оптимізованого під поширені апаратні платформи, підвищує технологічну незалежність країни та сприяє розвитку високотехнологічного сектору. Технології, розроблені для оборонних потреб, можуть знайти застосування і в цивільному секторі: в агродронах, системах моніторингу, промисловій автоматизації та телемедицині.

Дисертаційне дослідження безпосередньо пов'язане з виконанням науково–дослідних робіт на базі Національного університету «Київський авіаційний інститут», зокрема:

- НДР №1062-ДБ16 "Автоматизація розпізнавання та класифікації цільових об'єктів за відеоданими з камери безпілотного повітряного судна"(номер державної реєстрації 0116U004638);
- НДР №247-ДБ19 "Розроблення та виготовлення програмно–апаратних засобів цільового навантаження для повітряного спостереження та альтернативної навігації літального апарату"(номер державної реєстрації 0119U100553);
- НДР №479-ДБ24 "Розроблення багатокритеріальної завадозахищеної навігаційної системи для безпілотних апаратів"(номер державної реєстрації 0124U000291).

Робота відповідає статті 3 Закону України «Про пріоритетні напрями розвитку науки і техніки» в частині напряму «інформаційні та комунікаційні технології» у тематичних піднапрямах:

- «Високопродуктивні обчислення та суперкомп'ютерні технології»;
- «Інформаційні технології обробки та аналізу зображень»;
- «Інформаційні технології для національної безпеки і оборони».

Мета і задачі дослідження. Метою роботи є розв'язання науково-технічної задачі, що полягає в розробці інформаційної технології обробки ЦЗ та відео з

використанням швидкодіючої операції згортки (ШОЗ) для центрального процесора (ЦП) архітектури ARM64, яка б дозволила в режимі реального часу проводити фільтрацію та масштабування потокового відео та ЦЗ на одноплатних комп'ютерах з обмеженими обчислювальними ресурсами.

Для досягнення поставленої мети необхідно розв'язати такі **основні задачі:**

1. Провести аналіз існуючих програмно–апаратних та архітектурно–інформаційних засобів для забезпечення, підвищення та реалізації швидкодіючих обчислень, для ОЗ ЦЗ і програмного забезпечення в цілому. Визначити потенційні напрями вдосконалення ПЗ для цілочисельних обчислень.

2. Розробити математичну модель ОЗ та формалізовану нотацію для опису векторизованих обчислень з використанням SIMD NEON64.

3. Розробити метод ШОЗ для обробки 8-бітних RGB ЦЗ з використанням 16-бітних цілочисельних операцій, для забезпечення підвищення швидкодії на центральних процесорах архітектури ARM64.

4. Розробити ІТ підвищення швидкодії обробки RGB ЦЗ, для фільтрації та масштабування потокового відео та ЦЗ на одноплатних комп'ютерах з ЦП ARM64 в режимі реального часу.

5. Провести експериментальне дослідження швидкодії розробленого алгоритму на різних платформах SBC (Raspberry Pi 4, Orange Pi 5 Pro) для різних розмірів ЦЗ (256x144, 426x240, 640x360, 854x480, 1280x720, 1920x1080, 2560x1440, 3840x2160).

6. Проаналізувати отримані результати та визначити показники ефективності запропонованого підходу для різних розмірів ЦЗ та апаратних платформ.

7. Розробити методику експериментальної оцінки та порівняння швидкодії різних програмних реалізацій ОЗ на платформах з архітектурою центрального процесора ARM64 та рекомендації щодо застосування оптимізованого алгоритму ОЗ в системах реального часу та вбудованих пристроях.

Об'єктом дослідження є процеси обробки цифрових RGB-зображень операцією згортки на платформах з архітектурою ARM64.

Предметом дослідження є методи, алгоритми та інформаційна технологія векторизованої операції згортки (ВОЗ) з використанням SIMD-інструкцій ARM64 NEON для підвищення швидкодії обробки ЦЗ.

Методи дослідження. Для вирішення поставлених завдань використано

такі методи:

1. Теоретичні методи: аналіз та узагальнення наукової літератури з питань оптимізації ОЗ та обробки ЦЗ; математичне моделювання процесів обробки ЦЗ із використанням операторів згортки.

2. Емпіричні методи: експериментальне дослідження швидкодії розробленого алгоритму на різних платформах ARM64; порівняльний аналіз швидкодії розробленого алгоритму з існуючими рішеннями.

3. Методи програмної інженерії: розробка та оптимізація програмного коду з використанням мов C/C++ та асемблерних вставок (Inline Assembly); використання інструментів профілювання для аналізу швидкодії коду.

4. Методи комп'ютерного моделювання: симуляція роботи алгоритму на різних типах ЦЗ та з різними параметрами згортки.

5. Статистичні методи: обробка та аналіз експериментальних даних; візуалізація результатів досліджень з використанням графіків та діаграм.

6. Методи оптимізації: застосування технологій SIMD, зокрема інструкцій NEON64; оптимізація алгоритмів для роботи з цілочисельними даними.

7. Методи системного аналізу: аналіз ефективності розробленого алгоритму в контексті різних сценаріїв застосування та апаратних платформ.

Наукова новизна одержаних результатів полягає у такому:

Вперше:

1. Розроблено та теоретично обґрунтовано метод ШОЗ для обробки 8-бітних RGB ЦЗ з використанням 16-бітних цілочисельних операцій, що забезпечує підвищення швидкодії на центральних процесорах архітектури ARM64 порівняно з аналогічним підходом на основі функції cv::filter2D бібліотеки OpenCV:

– для ядер згортки середнього розміру (8x8–11x11) досягнуто прискорення в діапазоні 3,1–11,1 раза для Orange Pi 5 Pro та 2,7–10,2 раза для Raspberry Pi 4;

– для великих ядер згортки (12x12–15x15), де бібліотека OpenCV може використовувати FFT-алгоритми, запропонований метод демонструє прискорення 1,8–5,8 раза для Orange Pi 5 Pro та 1,7–5,2 раза для Raspberry Pi 4;

– для операцій масштабування ЦЗ, що включають чотириразове застосування ОЗ, розроблений підхід (zoomUP_2__NEON64) показує прискорення до 6,3 раза (Orange Pi 5 Pro) та до 6,2 раза (Raspberry Pi 4) для ядер середнього розміру порівняно з аналогічною операцією на базі cv::filter2D.

2. Запропоновано формалізовану нотацію для опису векторизованих об-

числень з використанням SIMD NEON64, яка дозволяє представляти опис векторизованої частини коду у вигляді структурованих виразів, що сприяє кращій читабельності та знижує поріг входу при засвоєнні технік експертної векторизації.

Удосконалено технологію обробки ЦЗ на комп'ютерах, що підтримують паралелізм обчислень, шляхом запропонованої в роботі ІТ обробки зображень та відео з використанням ШОЗ для архітектури ARM64, яка дозволяє в режимі реального часу проводити фільтрацію та масштабування потокового відео та ЦЗ на одноплатних комп'ютерах з ЦП ARM64, як-от: Raspberry Pi 4 та Orange Pi 5 Pro.

Набули подальшого розвитку лінійні оператори на основі лінійних комбінацій *B*-сплайнів, що є близькими до інтерполяційних у середньому, а саме: отримано лінійні оператори 16-бітних цілочисельних операцій, які гарантують відсутність переповнення при 16-бітному обчисленні ВСД, що дозволяє їх застосування для обробки ЦЗ на основі розробленого методу, а саме для субсмугової фільтрації, контрастування, цифрової стабілізації ЦЗ та для рекурентного двократного збільшення лінійних розмірів ЦЗ.

Практичне значення отриманих результатів:

1. Розроблено алгоритм методу ШОЗ для обробки RGB ЦЗ з використанням 16-бітних цілочисельних обчислень (Розділ 3), що включає формалізацію векторизованих обчислень, адаптацію ядер згортки та його (алгоритму) оптимізацію для SIMD-архітектур ЦП.

2. На основі розробленої алгоритмізації створено програмну реалізацію ОЗ шляхом використання інструкцій SIMD NEON ARM64, що забезпечує прискорення обробки ЦЗ до 11,1 раза (Orange Pi 5 Pro) та до 10,2 раза (Raspberry Pi 4) порівняно з аналогічним підходом на основі функції cv::filter2D бібліотеки OpenCV на SBC та подібних системах.

3. Запропоновано та апробовано методику експериментальної оцінки та порівняння швидкодії різних програмних реалізацій ОЗ на платформах з архітектурою центрального процесора ARM64, яка дозволяє об'єктивно визначати переваги запропонованих та оптимізованих у роботі алгоритмів.

4. Розроблена технологія теоретично має кращу енергоефективність порівняно з реалізаціями на основі операцій з плаваючою комою завдяки використанню цілочисельних обчислень та скороченню часу виконання операцій

завдяки використанню SIMD ARM64 NEON, що є перспективним напрямом для подальших досліджень енергоспоживання на мобільних та вбудованих пристроях.

5. Вперше проведено комплексне дослідження ефективності запропонованої ОЗ для різних розмірів ЦЗ (від 256x144 до 3840x2160) та розмірів ядер згортки (від 2x2 до 15x15) на SBC з архітектурою ARM64 (Orange Pi 5 Pro, Raspberry Pi 4), що дозволило виявити найбільш швидкодіючі сценарії застосування розробленого методу, зокрема при використанні ЯЗ розміром 8x8 було отримано найбільші показники підвищення швидкодії, а саме, від 5,2 до 11,1 раза (Orange Pi 5 Pro) та від 4,5 до 10,2 раза (Raspberry Pi 4) порівняно з cv::filter2D OpenCV.

6. Розроблений у роботі алгоритм ОЗ та рекомендації щодо його застосування можуть бути використані в системах комп'ютерного зору на базі архітектури ЦП ARM64, що дозволяє підвищити швидкодію обробки ЦЗ без необхідності оновлення апаратного забезпечення, зокрема в автономних UAV-системах, що відповідає актуальним пріоритетним напрямам досліджень та розвитку обороноздатності України.

7. Результати дослідження можуть бути застосовані для покращення швидкодії систем на основі архітектури ЦП ARM64, зокрема систем відеоспостереження, аналізу супутникових знімків та обробки даних з камер БПЛА, що є актуальним при вирішенні завдань національної безпеки та оборони.

8. Розроблений метод ШОЗ може бути інтегрований в існуючі бібліотеки обробки ЦЗ (наприклад, OpenCV) для підвищення їх ефективності на платформах ARM64, що є перспективним напрямом розвитку високопродуктивних бібліотек обробки зображень.

9. Результати дослідження можуть бути покладені в основу розробки спеціалізованих SIMD-методів обробки ЦЗ, які будуть використовувати описану в (Додаток А) методику підвищення ШОЗ.

10. Запропоновані алгоритми ШОЗ можуть бути застосовані для прискорення ОЗ в згорткових нейронних мережах на мобільних та вбудованих пристроях, що сприятиме розвитку мобільних додатків з елементами штучного інтелекту.

11. Розроблена ІТ та програмне забезпечення (бібліотека libconv_lib.so на C++ з SIMD ARM64 NEON-оптимізаціями та Python-інтерфейс) надають готовий інструментарій для розробників систем, що потребують високої швидкодії

обробки ЦЗ на платформах ARM64, зокрема в системах бортової обробки даних БПЛА, мобільних робототехнічних комплексах та інших критичних застосунках, де поєднання швидкодії та енергоефективності є ключовим.

12. Результати досліджень впроваджено в освітній процес при розробці курсів: «Архітектура обчислювальних систем та операційні системи», «Системне програмування» та «Програмування мовою асемблер» (див. Додаток В).

13. Результати роботи впроваджено у виробничу діяльність ТОВ «ЕЛІ-САТ» (вбудоване програмне забезпечення пристроїв збору й обробки даних з датчиків) та ТОВ «ІНСПІРІТО ТІМ» (програмно-апаратні засоби обробки відеоданих з бортових камер БПЛА), що підтверджено відповідними актами впровадження (див. Додаток Г та Додаток Д).

Особистий внесок здобувача. Дисертаційна робота є результатом самостійного дослідження. У роботах, виконаних разом зі співавторами, автору належить: [3, 22, 23, 62, 63, 81] – розробка концепції та архітектури інформаційної технології підвищення швидкодії обробки ЦЗ на основі векторизованої ОЗ з використанням 16-бітних цілочисельних обчислень та SIMD ARM64 NEON; [23, 63, 81] – теоретичне обґрунтування та розробка математичних моделей ОЗ для 16-бітних цілочисельних обчислень при обробці 8-бітних RGB ЦЗ, у тому числі побудова умови коректності, методики адаптації знакових ЯЗ до беззнакових цілочисельних операцій та оптимізації для розріджених ЯЗ; [22, 23] – формулювання та обґрунтування формалізованої нотації для опису векторизованих обчислень з використанням SIMD NEON64; [3, 22, 23, 81] – безпосередня програмна реалізація ключових компонентів інформаційної технології, у тому числі розробка бібліотеки на C/C++ з NEON64-оптимізаціями та Python-інтерфейсу для прикладного використання; [3, 8, 22, 23] – планування, організація та проведення експериментальних досліджень оцінки швидкодії розробленої технології, у тому числі порівняльний аналіз з функцією cv::filter2D бібліотеки OpenCV на платформах Raspberry Pi 4 та Orange Pi 5 Pro; [22, 23, 62, 63] – аналіз отриманих експериментальних даних та формулювання висновків і рекомендацій за результатами дослідження.

Апробація результатів дисертації. Результати дисертаційної роботи доповідались та обговорювались на: XVI, XIX, XXI, XXIII Міжнародній науково–практичній конференції здобувачів вищої освіти і молодих учених «ПОЛІТ. Сучасні проблеми науки» (м. Київ, 2016, 2019, 2021, 2023), VIII, XXI Всесвітньо-

му конгресі «Авіація у XXI столітті» – «Безпека в авіації та космічні технології» (м. Київ, 2018, 2022), XIII, XVI Міжнародній науково–технічній конференції «АВІА» (м. Київ, 2017, 2023).

Публікації. Основні положення дисертаційної роботи опубліковано в 7 статтях, з яких 3 у виданнях, що входять до переліку фахових ВАК (нині – категорія Б МОН України), 4 у виданнях, що індексуються в наукометричних базах Scopus та Web of Science, а також у 7 тезах доповідей на конференціях.

Структура роботи та її обсяг. Дисертація складається з анотації, вступу, чотирьох розділів, висновків, додатків, списку використаних джерел і має 155 сторінок основного тексту, 11 рисунків, 18 таблиць, 91 сторінку додатків. Список використаних джерел містить 100 найменувань і займає 11 сторінок. Загальний обсяг роботи 246 сторінок.

РОЗДІЛ 1. АНАЛІЗ СТАНУ ТА ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ АЛГОРИТМУ ЛІНІЙНОЇ ОПЕРАЦІЇ ЗГОРТКИ З ВИКОРИСТАННЯМ SIMD ОПТИМІЗАЦІЙ ПРИ ОБРОБЦІ ЦЗ ЗА УМОВИ 16-БІТНИХ ЦІЛОЧИСЕЛЬНИХ ОБЧИСЛЕНЬ

Розділ присвячений дослідженню швидкодії алгоритму лінійного ОЗ та огляду сучасних технік оптимізації ПЗ в цілому. Представлено короткий огляд сучасної оптимізації програмного забезпечення (підрозд. 1.1). Розглянуто різні типи оптимізацій, зокрема апаратні (SIMD NEON ARM64 (підрозд. 1.2.1), AVX (підрозд. 1.2.2), GPU SIMT (підрозд. 1.2.3), DSP Hexagon (підрозд. 1.2.4), суперкомп'ютерні архітектури (підрозд. 1.2.5)) та програмні (компілятори (підрозд. 1.3.1), CUDA (підрозд. 1.3.2), OpenCL (підрозд. 1.3.3), бібліотеки OpenCV та ACL (підрозд. 1.3.4), ігровий двигун id Tech 3 (підрозд. 1.3.5), розгортання циклів (підрозд. 1.3.6), кеш-френдлі оптимізація (підрозд. 1.3.7)). Розділ завершується узагальненим оглядом SIMD-реалізації cv::filter2D() на основі OpenCV Universal Intrinsics (підрозд. 1.4). Висновки підкреслюють важливість низькорівневих технік (підрозд. 1.5).

Процес автовекторизації (АВ) програмного коду здебільшого базується на використанні SIMD (Single Instruction, Multiple Data) інструкцій центрального процесора (ЦП). Ця техніка широко застосовується у сучасних компіляторах, таких як GCC [1] та Clang/LLVM [2]. АВ вмикається шляхом компіляції програми з прапором -O3 або з більш «агресивними» прапорами, такими як -O4/-Ofast, які, однак, не рекомендуються для компілювання фінальних/релізних версій програмних продуктів через потенційну нестабільність роботи результуючого продукту/бібліотеки/програми. Слід зазначити, що назви прапорів можуть відрізнятися залежно від платформи, компілятора та мови програмування, але їх призначення залишається незмінним — задіяти АВ для оптимізації коду проєкту.

Проте, як показано в [3], для ефективної обробки ЦЗ часто необхідна вища швидкодія, ніж та, що досягається за допомогою АВ. Крім того, вирішення різноманітних задач обробки ЦЗ має велике значення і може суттєво підвищити швидкодію широкого спектра застосунків, таких як обробка відеопотоку (стабілізація, фільтрація, шумопригнічення тощо), застосування різноманітних ефектів до одного і того ж ЦЗ, а також загальна обробка ЦЗ (фільтрація, шумопригнічення). Слід завжди брати до уваги такі ключові фактори, з урахуванням вищезгаданого:

1. обчислювальна складність обраного методу/функції/процедури;
2. можливість оптимізації/векторизації методу/функції/процедури (взагалі чи це можливо з позиції векторизації);
3. наявні апаратні ресурси цільової процесорної архітектури (неможливість використання SIMD-оптимізації, якщо немає векторного співпроцесора (навіть якщо ЦП є представниками схожих сімейств)).

Одним із можливих шляхів зменшення обчислювальної складності є розробка нових спеціалізованих методів для конкретних задач, як-от у [4, 5]. Крім того, серед операцій обробки ЦЗ можна виділити найбільш ресурсоємні, але водночас і найбільш значущі з точки зору часу виконання: згортку, масштабування (яке досягається за рахунок згортки) та аналіз кольору (яскравості, контрасту тощо). ОЗ (1.1) є одночасно найпростішою та найскладнішою (з наведених вище) у контексті ресурсоємності та оптимізації:

$$p_{i,j} = \sum_{k=0}^{r} \sum_{l=0}^{c} \Gamma_{k,l} P_{k+i-a,l+j-a'}, \tag{1.1}$$

Де $i = a, \ldots, W - (r - a) - 1$, $j = a', \ldots, H - (c - a') - 1$ є індексами вихідного ЦЗ p; W та H — ширина та висота вхідного ЦЗ P та вихідного ЦЗ p (на цьому етапі дослідження ми нехтуємо ефектом країв вхідного/вихідного ЦЗ), Γ — ядро ОЗ (матриця $r \times c$), і, нарешті, a, a' — так звані якорі, які визначають відносне положення відфільтрованої точки ЦЗ.

Рівняння (1.1) є цілком сумісним із функцією cv::filter2D(...) бібліотеки OpenCV [6] та NEConvolution{N}x{N}::run() бібліотеки ARM Compute Library [7], але в цій роботі ми зосередимося на квадратних ядрах згортки (ЯЗ), тобто $c = r$, і надалі вважатимемо, що ЯЗ має квадратну форму за замовчуванням.

Щодо бібліотеки ACL, слід зазначити, що після коміту (***7dcb9fadb9***) ця бібліотека була «оптимізована» виключно для використання у режимі роботи додатків штучного інтелекту (AI) постоброблення, а частину класів/коду, відповідальних за обробку ЦЗ, було відредаговано/відформатовано/зведено до необхідного мінімуму або й узагалі видалено. Це означає, що ОЗ (наприклад, NEConvolution5x5::run()), яка була конкурентним рішенням для даного дослідження, була видалена. Незважаючи на це, коміт з кодом, який надає/представляє конкурентне рішення, було зафіксовано (***7dcb9fadb9***), і в разі потреби можна повернутися до розгляду його нюансів/коду. Але, так чи інакше, можна відзначити,

що кеш-френдлі стиль коду, що ззовні інколи здається комплексним/громіздким, насправді надає суттєвий приріст швидкодії (порівнюючи його з класичним написанням коду; прикладом може слугувати код, що множить дві матриці та кладе результат розрахунків у третю) [8].

Позитивним аспектом є те, що кожен піксель вихідного ЦЗ може бути обчислений одночасно, і, по суті, задача може бути розпаралелена. Розпаралелювання настільки важливе, що розробники апаратного та програмного забезпечення створили набір ППО: Nvidia CUDA, OpenCL, ATI Stream Technology (ATI-ST) (доступна через OpenCL API), Intel Threading Building Blocks (TBB), Apple Metal Compute, Heterogeneous System Architecture (HSA), ArrayFire тощо. Кожен розробник ППО надає програмний інструментарій: мову програмування з С-подібним синтаксисом + API, додаткові модулі, фреймворки, набір бібліотек, набір спеціалізованих спец-компіляторів тощо. Основою для обчислень ППО переважно є сучасні графічні прискорювачі, здатні виконувати будь-яку задачу паралельно за умови використання згаданого вище інструментарію. Наприклад, на даний момент GeForce GTX 3080/4080 є дуже популярними для навчання CNN, значно прискорюючи його. Варто зазначити, що шейдерні блоки графічного процесора схожі на мобільні процесори, які використовують архітектуру RISC, що застосовується виробниками сучасних смартфонів. Таким чином, можна стверджувати, що покращення швидкодії ОЗ для однієї архітектури центрального процесора позитивно вплине майже на будь-яке програмне забезпечення на будь-якій операційній системі через широке розмаїття подібних завдань: фільтрація ЦЗ (підвищення різкості, виявлення країв, розмиття тощо), масштабування ЦЗ, навчання CNN, прискорення обробки мультимедіа тощо. Варто зазначити, що ОЗ (1.1) є основою для функціонування CNN, тому, прискорюючи ОЗ, ми досягаємо вищої швидкодії CNN і, таким чином, зменшуємо час навчання (у перспективі). Більше того, в певному сенсі підвищення швидкодії для подібних задач (як ОЗ) сформувало сучасні підходи до архітектури GPU.

У цій роботі ми пропонуємо новий метод оптимізації ОЗ, який використовує SIMD. Оскільки SIMD може застосовуватися не лише до цілочисельних ядер, ми продемонструємо метод редукції/конвертування/оптимізації для ядер з дійсними значеннями, зведених до цілочисельного діапазону, що дозволяє застосувати цю техніку. Крім того, ми надамо експериментальне порівняння коду, створеного людиною на основі цього підходу, з іншими визнаними рішеннями

(бібліотеками OpenCV, ARM Compute Library). Решта дослідження побудована так. Спочатку ми розглянемо переваги та недоліки SIMD, а потім у главі 2 представимо спеціалізовані ЯЗ, а у главі 3 — самі методи оптимізації ОЗ із використанням цих ЯЗ; у главі 4 буде описано дослідження швидкодії ОЗ, що використовує у своїй основі SIMD-операції (головний метод дослідження).

1.1 Короткий огляд сучасної оптимізації програмного забезпечення

Проведемо огляд у стилі ”знизу догори розглядаючи спочатку апаратне забезпечення, потім програмне, і, нарешті, алгоритмічні методи підвищення швидкодії ПЗ.

Слід розглянути дві стратегії написання програм: перша стратегія передбачає, що у програміста є весь необхідний інструментарій для реалізації програми — ЕОМ та набір компіляторів, інтерпретаторів, текстових редакторів; друга передбачає, що він (програміст) повинен створити такий інструментарій самостійно, тобто має знання машинних кодів. Перша стратегія вимагає дуже широкого спектру знань як мов програмування, так і програмних бібліотек та фреймворків для досягнення поставленої мети. Друга передбачає, що програміст володіє мовами програмування (як-от C та асемблер), які є дуже наближеними до команд ЦП (або ЕОМ) таким чином, що далі абстрактний рівень закінчується і починається рівень програмування за допомогою машинних кодів. Також можна припустити, що ці дві стратегії умовно розділяють програмістів на дві категорії — високорівневих та низькорівневих програмістів. Основною проблемою низькорівневих програмістів є те, що дуже складно знати всі діалекти наявних асемблерів. Більше того — сучасні корпорації не дійшли згоди щодо питання діалектів асемблерів (щодо створення/розробки єдиного діалекту асемблера, як такого), як і до єдиної ефективної архітектури ЦП. Скоріше за все, це не буде вирішено взагалі, бо дух конкуренції, а не консолідації, є рушієм сучасної економіки. Отже, немає необхідності дискутувати щодо цього. Є сенс просто наголосити на тому, що передумовою для таких рішень є створення мови програмування C (і далі її похідних, як-от C++, Java тощо), яка об’єднала всі діалекти наявних на той час асемблерів, спонукавши виробників створювати компілятори самостійно, через попит на C. Це зняло тягар потреби у знаннях усіх тонкощів різних ЦП

та їх діалектів асемблерів, наявних на той час, з низькорівневих програмістів. І цей процес триває й понині, наслідком чого може слугувати такий компілятор, як Clang, та його бекенд LLVM, і той факт, що низькорівневою мовою тепер вважають С [2]. Але інколи без знань специфічних команд асемблера даного ЦП неможливо отримати від компілятора оптимізації програмного продукту, навіть якщо буде використано АВ -O3 (або - Ofast). Саме тому вендори використовують/надають два типи інструментарію: Inline Assembly та більш популярні серед високорівневих програмістів Intrinsics. Ці два підходи надають можливість обом типам програмістів використовувати вбудований потенціал сучасних ЦП майже на 100% [9].

Але здебільшого Intrinsic-функції (хоча вони просто виглядають як функції, насправді це вказівки компілятору, що в частині коду, де йде їх використання, треба зробити прямі підстановки SIMD-інструкцій) надають у результаті більше SIMD (Single Instruction, Multiple Data) асемблерного векторизованого коду, ніж того очікує програміст. І навпаки, Inline Assembly надає рівно стільки SIMD асемблерного векторизованого коду, скільки було записано, бо Inline Assembly є прямим наказом вбудувати його команди у дану частину коду компіляторові. Але це є прямий доступ до команд, що виконує процесор, тобто регістри ЦП, що зайняті програмою, яка використовує SIMD-команди ЦП, обрані завчасно. Отже, два підходи для написання векторизованого коду — використання Intrinsic-функцій або Inline Assembly — надають інструменти обом типам програмістів для векторизації ПП [5, 9].

Отже, коли було оглянуто (хоч і коротко) ці два підходи (Inline Assembly та використання Intrinsics), можна перейти до подальшого огляду, наголосивши на тому, що автор дослідження зупинився на використанні Inline Assembly задля досягнення більш швидкодіючого результату. Мінусом цього рішення є підвищення часу на написання коду оптимізації функцій (або частин функцій). Але тепер відомо про два підходи, котрі по суті не є конкурентами, але надають обом типам програмістів можливості для підвищення швидкодії коду.

1.2 Прискорення за допомогою апаратного забезпечення

Апаратне забезпечення може впливати на швидкодію ПЗ як позитивно, так і негативно, залежно від якості компіляторів, за допомогою яких будуються

ПП, та залежно від самої ОС (планувальник тощо — складові частини ОС, котрі впливають на швидкодію ПЗ в цілому). Таксономія Флінна, яка класифікує архітектури ЕОМ на основі їхньої здатності до паралелізму, підкреслює важливість розуміння нюансів архітектур сучасних ЦП, які будуть експлуатуватися для конкретних обчислювальних задач [10, 11]. SIMD є однією з таких ознак паралелізму на апаратному рівні, що дозволяє обробляти великі масиви даних паралельно, використовуючи одну інструкцію/операцію і, що не менш важливо, в одному апаратному потоці [12]. Це особливо ефективно для задач, які вимагають однотипної обробки великих обсягів даних, наприклад, обробки ЦЗ або відео [13].

1.2.1 SIMD NEON ARM64

Перше питання, на яке потрібно дати відповідь — «Чи можливе використання паралелізації обробки масиву даних ЦЗ для елементів ЯЗ за допомогою технології SIMD?». Наразі такі операції, як SIMD, реалізовані як у RISC-ЦП (наприклад, процесори ARM Cortex-A8-23 [14–17], Cortex-A53-72 та більш сучасних версіях Cortex-X1/2/3/4 [12]) та RISC-V, так і в CISC-ЦП (наприклад, серія ЦП Intel x86/64) [18, 19]. Доступ до цієї функціональності можна отримати за допомогою спеціальних розширень Inline Assembly C/C++ — NEON64 (існує варіація NEON32, але у даному дослідженні було проведено серії експериментів, використовуючи NEON64 оптимізацію коду, тож надалі будемо мати на увазі саме NEON64-оптимізацію коду) фірми ARM (для RISC-архітектури) [14, 20, 21]. NEON64 є набором інструкцій, який дозволяє виконувати SIMD-операції, тобто застосовувати одну й ту саму команду до масиву даних, наче це робиться в циклі для кожного елемента, але водночас. Це особливо важливо для мобільних пристроїв, де енергоефективність є критичною, а наявність SIMD зменшує навантаження на обчислювальні задачі. NEON64 підтримує широкий спектр операцій, включаючи арифметичні, перетворення даних та обробку сигналів, що робить його ідеальним для різноманітних застосувань — від мультимедіа до машинного навчання. Отже, відповідь на поставлене питання така: «Так, однозначно це можливо» [8, 22, 23].

1.2.2 SIMD - AVX

З іншого боку, CISC-архітектура, представлена серією процесорів Intel x86/64, використовує SIMD через розширення SSE_n (Streaming SIMD Extensions) та $AVX_{1|2|512}$ (Advanced Vector Extensions) [24, 25]. Ці розширення дозволяють процесорам Intel ефективно виконувати складні обчислення на великих наборах даних, що є критично важливим для наукових обчислень, фінансового моделювання та інших високошвидкодіючих обчислювальних задач.

По суті, $AVX_{1|2|512}$ є розширенням набору інструкцій x86 для ЦП Intel і AMD. AVX є наступником SSE (Streaming SIMD Extensions) і пропонує значні покращення у порівнянні з попередником [24–26]. Ось основні відмінності між SSE та AVX:

1. Ширина регістрів: SSE використовує 128-бітні регістри, тоді як AVX підтримує 256-бітні регістри, що дозволяє обробляти вдвічі більше даних за одну операцію.

2. Кодування інструкцій: AVX використовує новий формат кодування VEX, який дозволяє більш ефективно представляти інструкції.

3. Гнучкість операцій: AVX вводить трьохоперандний формат інструкцій, що дозволяє виконувати операції виду ”a = b op c”за одну інструкцію, на відміну від двохоперандного формату SSE.

4. Розширені можливості: AVX2 додає підтримку цілочисельних операцій з 256-бітними векторами, а AVX-512 розширює можливості до 512-бітних операцій.

Вплив на швидкодію:

1. Обробка зображень: AVX дозволяє обробляти вдвічі більше пікселів за одну операцію порівняно з SSE, що особливо корисно для фільтрів згортки та інших операцій обробки зображень.

2. Наукові обчислення: У задачах, що вимагають операцій з рухомою комою, AVX може забезпечити прискорення до 2 разів порівняно з SSE.

3. Криптографія: AVX2 вводить інструкції, корисні для криптографічних алгоритмів, що може значно прискорити шифрування та дешифрування.

4. Машинне навчання: AVX-512 особливо ефективний для ОЗ в нейронних мережах, що може прискорити тренування та інференс моделей.

Важливо зазначити, що ефективне використання AVX вимагає оптимізації на рівні компілятора або ручного написання інтринсиків [24–27]. Проте, по-

тенційний приріст швидкодії робить цю технологію привабливою для багатьох швидкодіючих застосувань.

Таким чином, розуміння та використання можливостей SIMD в різних архітектурах є важливим аспектом для розробників програмного забезпечення, які прагнуть максимізувати швидкодію своїх додатків. Вибір між RISC та CISC архітектурами залежить від конкретних вимог до швидкодії та енергоефективності [28]. На відміну від SIMD, принципи MIMD (Multiple Instruction, Multiple Data) не реалізовані в сучасних процесорах, але частково підтримуються графічними процесорами.

1.2.3 GPU – SIMT (майже MIMD)

Сучасні графічні процесори (GPU) забезпечують можливості паралельних обчислень за допомогою шейдерних блоків — SCPU (Shader Control Processing Unit), заснованих на RISC-архітектурі, які можуть використовуватися паралельно за допомогою таких фреймворків, як Nvidia CUDA [29] та OpenCL [30]. Принцип MIMD (Multiple Instruction, Multiple Data) досягається за допомогою SIMT (Single Instruction, Multiple Threads) інструкцій у векторних інструкціях SCPU для регістрів довжиною 128/256/...-біт [31]. Зазвичай GPU мають 32 або більше SIMD регістрів, що перевищує кількість, яку мають сучасні ЦП ARM або Intel. Однак, отримати доступ до SIMD/MIMD інструкцій SCPU безпосередньо дещо ускладнено, оскільки вони огорнуті C-подібними функціями/макросами, і лише фіксована кількість внутрішніх та попередньо реалізованих операцій є безпосередньо доступною, наприклад, бітові зсуви та двійкова логіка. Тому в більшості випадків програмісти використовують спеціальні фреймворки для доступу до цих можливостей, такі як CUDA та OpenCL для GPU, або просто OpenCL для роботи з CPU/DSP/FPGA. Поширеність цих фреймворків призвела до того, що більшість виробників графічних карт отримали сертифікати сумісності з ними. Під ”їхніми”слід розуміти таких техно-гігантів, як Khronos Group Inc (сертифікування виробників та розширення функціоналу OpenCL та інших Open-подібних фреймворків) та Intel/NVidia (сертифікування та розширення CUDA фреймворку/бібліотеки).

Для ілюстрації ефективності GPU порівняно з CPU розглянемо ОЗ зображення — фундаментальну операцію в обробці ЦЗ та комп’ютерному зорі. Візьмемо за приклад згортку зображення розміром 4096x4096 пікселів з ядром

3x3.

CPU реалізація (використовуючи OpenMP для паралелізації на 8 ядрах): Intel Core i7-10700K @ 3.80GHz: ∼120 мс

GPU реалізація (використовуючи CUDA): NVIDIA GeForce RTX 3080: ∼2,5 мс

У цьому конкретному випадку GPU демонструє прискорення приблизно в 48 разів порівняно з багатоядерним CPU. Це пояснюється здатністю GPU обробляти тисячі потоків одночасно, що ідеально підходить для паралельної природи ОЗ.

Більш прикладне дослідження для сучасних спеціалістів з AI та data science представлено у роботах NVIDIA [32, 33], де показано значне підвищення швидкодії навчання моделей на однакових наборах даних при використанні GPU порівняно з CPU. Згідно з наведеними результатами, навіть обмеження пропускної здатності шини даних, що зазвичай є основним фактором, який стримує використання зовнішніх пристроїв у обчисленнях, не завадило досягти суттєвого прискорення.

Важливо зазначити, що реальна швидкодія може варіюватися залежно від специфіки завдання, розміру даних та оптимізації коду. Крім того, необхідно враховувати час передачі даних між CPU та GPU, який може бути значним для менших обсягів обчислень.

Ці приклади демонструють потенціал SIMT архітектури GPU для задач, які можуть бути ефективно розпаралелені, особливо в області обробки ЦЗ та машинного навчання.

1.2.4 DSP. Новітній DSP Hexagon від фірми Qualcomm

Крім використання GPU, можна використовувати співпроцесори, наприклад, цифровий сигнальний процесор (DSP). DSP відіграють важливу роль у сучасних обчислювальних системах, зокрема у мобільних пристроях, де вони використовуються для оптимізації обробки аудіо, відео та інших мультимедійних даних. Qualcomm, один з провідних виробників мобільних процесорів, розробив DSP Hexagon, який інтегрований у їхні процесори Snapdragon. DSP Hexagon використовує процесорну архітектуру VLIW (Very Long Instruction Word), яка дозволяє одночасно виконувати кілька інструкцій, збільшуючи загальну швидкодію обробки (за умови відсутності співпадінь аргументів цих інструкцій).

Особливістю DSP Hexagon є його здатність обробляти до трьох інструкцій за один такт ЦП, кожна з яких має мати різні вхідні дані. Це забезпечує високий ступінь паралелізму на рівні асемблера, що є важливим для задач, де потрібна швидка обробка великих обсягів даних. У порівнянні з традиційними SIMD-інструкціями, такими як NEON (у ARM-процесорах), DSP Hexagon пропонує значно вищу швидкодію, особливо у сферах, де потрібна інтенсивна обробка сигналів, як-от кодування та декодування аудіо та відео. До того ж це окремий апаратний модуль сучасного SoC, що, у свою чергу, робить його використання ще й максимально енергоефективним.

Порівняно з простим SIMD (NEON32 або NEON64), швидкодія DSP Hexagon у 4 рази вища для кодування/декодування аудіо/відео і може сягати до $\sim$18 разів [34]. Однак використання DSP Hexagon має свої особливості. Через високу складність обробки даних, переривання на рівні DSP може бути більш часово витратним порівняно з перериваннями в ARM SIMD. Це означає, що для задач, де потрібна часта передача даних між процесором та DSP, може спостерігатися зниження загальної швидкодії. Таким чином, ефективне використання DSP Hexagon вимагає ретельного планування та оптимізації обчислювальних процесів, щоб максимізувати його переваги та мінімізувати потенційні недоліки. Тому його використання є дуже схожим за стилем на використання сучасних GPU – якщо постійно передавати та приймати дані, то швидкість їх обробки буде значно знижена. У кожній технології є свої недоліки та переваги.

Для використання можливостей DSP Hexagon розробникові потрібно використовувати наступне ПЗ/фреймворки/бібліотеки:

1. Qualcomm Hexagon SDK: Це основний інструмент для розробки під Hexagon DSP. SDK містить компілятори, симулятори, документацію та приклади коду. Він дозволяє розробникам писати та оптимізувати код безпосередньо для Hexagon DSP;
2. Hexagon Neural Network (NN) library: Ця бібліотека оптимізована для виконання операцій глибокого навчання на DSP Hexagon. Вона інтегрується з популярними фреймворками машинного навчання, такими як TensorFlow Lite та SNPE (Snapdragon Neural Processing Engine) [35];
3. FastCV: Бібліотека комп'ютерного зору, оптимізована для роботи на Hexagon DSP. Вона надає ряд функцій для обробки ЦЗ та відео, що автоматично використовують можливості DSP [36];

4. Qualcomm Math Library: Набір оптимізованих математичних функцій, які можуть виконуватися на DSP для підвищення швидкодії обчислень;

5. HVX (Hexagon Vector eXtensions): Програмісти можуть використовувати ці векторні розширення для написання високоефективного коду, що працює на DSP [35].

Важливо зазначити, що розробка для DSP Hexagon вимагає специфічних знань та навичок. Код має бути спеціально оптимізований для архітектури DSP, щоб повністю використати її переваги. Крім того, не всі додатки можуть отримати значну вигоду від використання DSP, тому розробники повинні ретельно оцінювати доцільність перенесення обчислень на DSP для кожного конкретного випадку.

1.2.5 Суперкомп'ютерні архітектури на базі ARM64: NVIDIA Grace

Розвиток архітектури ARM64 вийшов далеко за межі мобільних пристроїв, впевнено завойовуючи позиції в сегменті високошвидкодіючих обчислень (HPC) та центрів обробки даних. Яскравим прикладом цієї тенденції є розробка компанією NVIDIA суперчипів, що базуються на ARM-архітектурі, зокрема процесора NVIDIA Grace та інтегрованої системи Grace Hopper.

NVIDIA Grace CPU є процесором, розробленим спеціально для великомасштабних завдань штучного інтелекту та HPC. Він містить 72 ядра ARM Neoverse V2, оптимізованих для однопотокової швидкодії та підтримки векторних розширень SVE2 (Scalable Vector Extension 2). Це дозволяє досягти надзвичайної енергоефективності та продуктивності на Ват, що є критичним для сучасних суперкомп'ютерів.

Ще більш значущим є суперчип NVIDIA Grace Hopper (GH200), який об'єднує на одному модулі ARM-процесор Grace та графічний прискорювач Hopper H100. Ключовою інновацією є використання високошвидкісного інтерконекту NVLink-C2C, що забезпечує когерентний доступ до пам'яті між CPU та GPU з пропускною здатністю до 900 ГБ/с. Це усуває традиційне вузьке місце — шину PCIe — і дозволяє GPU напряму звертатися до системної пам'яті CPU, а CPU — до пам'яті GPU, значно спрощуючи програмування та прискорюючи виконання задач, що потребують інтенсивного обміну даними.

Такі системи, як NVIDIA DGX GH200, що об'єднують до 256 суперчипів Grace Hopper, демонструють потенціал архітектури ARM64 у створенні екзафло-

псних суперкомп'ютерів для найскладніших наукових та промислових завдань. Для розробників програмного забезпечення це означає, що методи оптимізації, орієнтовані на ARM64 та її SIMD-можливості (NEON/SVE), стають актуальними не лише для мобільних пристроїв, але й для найпотужніших обчислювальних систем у світі. Це підкреслює важливість даного дослідження, оскільки розроблені методи оптимізації для ARM64 можуть бути масштабовані від вбудованих систем до суперкомп'ютерного рівня.

1.3 Оптимізація за допомогою програмного забезпечення

1.3.1 Компілятори

Програмне забезпечення (ПЗ), яке ми використовуємо для створення виконуваних бінарних файлів (наприклад, сам компілятор, додаткові бібліотеки, фреймворки), суттєво впливає на швидкодію програми за рахунок використання різних оптимізацій та можливостей апаратної платформи. Однією з ключових оптимізацій є АВ, яка дозволяє компіляторам автоматично перетворювати деякі частини коду на векторні інструкції, що значно підвищує швидкодію обчислень, особливо при роботі з великими масивами даних, та без значної втрати обчислювальної точності [4]. Розглянемо три відомі компілятори: GNU Compiler Collection (GCC/G++) [1], Clang [2, 37] та nvcc (компілює cu-файли для CUDA).

GCC, розроблений спільнотою FSF і вперше створений Річардом Столлманом, є не просто компілятором, а цілою колекцією компіляторів для різних мов програмування та архітектур ЦП. Його здатність до АВ дозволяє ефективно використовувати можливості сучасних процесорів, зокрема SIMD інструкції, для оптимізації обчислювальних процесів [38].

Clang, який активно використовується компаніями, такими як Apple та Google, є фронтендом для різних мов програмування і використовує LLVM як свій бекенд. LLVM, що лежить в основі Clang, забезпечує потужні можливості оптимізації, включаючи АВ для архітектур, таких як ARM. Це дозволяє Clang ефективно конкурувати з GCC у плані швидкодії, надаючи розробникам гнучкі інструменти для створення високошвидкодійного коду [39]. Clang є фронтендом для C, C++, Objective-C, Objective-C++ та OpenCL. Власне генерацію двійкового коду та векторизацію виконує фреймворк/бекенд LLVM (для архітектури ЦП ARM64). Як GCC, так і Clang орієнтовані на швидкодію, але все ж вони посту-

паються в порівнянні з вручну написаним асемблерним кодом, що використовує SIMD-оптимізацію ресурсомісткого коду програми [40].

nvcc, компілятор від NVIDIA, спеціалізується на компіляції коду для CUDA, платформи паралельних обчислень, розробленої NVIDIA. nvcc здатний оптимізувати код для виконання на GPU, використовуючи техніки, подібні до AB, щоб максимально використати потенціал паралельних обчислень.

Хоча сучасні компілятори, такі як GCC, Clang та nvcc, здатні виконувати вражаючі оптимізації, включаючи AB, вони все ще не завжди можуть зрівнятися з вручну написаним асемблерним кодом з точки зору максимальної швидкодії. Однак, враховуючи складність та часові витрати на написання асемблерного коду, використання сучасних компіляторів та їх можливостей щодо AB є більш практичним рішенням для більшості програмістів, але при цьому швидкодія/ресурсомісткої частини коду буде посередньою (під питанням).

GCC та Clang пропонують широкий спектр оптимізацій, які можна активувати за допомогою різних рівнів оптимізації (-O1, -O2, -O3, -Ofast) та спеціальних прапорців [32]. Ось деякі ключові оптимізації:

1. Вбудовування функцій (Function Inlining): GCC та Clang можуть автоматично вбудовувати невеликі функції, що зменшує накладні витрати на виклики функцій. Це особливо ефективно для коротких, часто викликаних функцій.

2. Розгортання циклів (Loop Unrolling): Компілятори можуть розгорнути цикли, зменшуючи кількість ітерацій та збільшуючи можливості для паралелізму на рівні інструкцій.

3. Константне згортання (Constant Folding), або обчислення, які можна виконати під час компіляції (наприклад, 2+3).

4. Видалення ”мертвого”коду (Dead Code Elimination).

5. AB: GCC та Clang можуть автоматично перетворювати скалярні операції на векторні SIMD інструкції. Наприклад, цикл, що обробляє елементи масиву, може бути трансформований для використання SSE або AVX інструкцій [41–44].

6. Оптимізація графу потоку керування, що зменшує кількість умовних переходів завдяки аналізу та оптимізує структуру розгалужень і циклів у програмі компіляторами.

7. Повторне використання регістрів (Register Reallocation), або оптимізація використання регістрів процесора для зменшення доступів до пам’яті.

Clang, завдяки використанню LLVM, має додаткові можливості:

1. Міжпроцедурна оптимізація (Interprocedural Optimization, IPO): LLVM може оптимізувати код на рівні всієї програми, а не тільки окремих функцій.
2. Оптимізація на основі профілю (Profile-Guided Optimization, PGO): Використання інформації про реальне виконання програми для подальшої оптимізації.

Важливо зазначити, що ефективність цих оптимізацій може варіюватися залежно від конкретного коду та архітектури.

1.3.2 CUDA

Останній компілятор, який заслуговує на увагу в контексті сучасних технологій оптимізації, - nvcc, який є ключовим елементом екосистеми CUDA від NVIDIA. Технологія CUDA відкриває нові горизонти для паралельних обчислень на графічних процесорах (GPU), що є критично важливим для задач, які можуть бути ефективно розподілені на дрібніші, незалежні операції [31].

Архітектура CUDA базується на концепції Streaming Multiprocessors (SM), які є основними обчислювальними блоками GPU. Кожен SM містить велику кількість ядер CUDA, здатних виконувати тисячі потоків паралельно. Ця масивна паралельність є ключовою перевагою CUDA над традиційними CPU для певних типів обчислень.

Особливістю CUDA є використання моделі SIMT (Single Instruction, Multiple Thread), яка, хоч і має схожості з SIMD у традиційних ЦП, проте працює за іншим принципом. У SIMT одна інструкція виконується над кількома потоками даних, але вона застосовується до варпів - груп потоків, зазвичай по 32 на кожен шейдерний конвеєр GPU від NVIDIA. Варп - це фундаментальна одиниця виконання в архітектурі CUDA. Розмір варпу в 32 потоки був обраний як оптимальний баланс між гнучкістю та ефективністю використання ресурсів GPU [29].

Кожен потік у варпі виконує одну й ту ж інструкцію одночасно, що дозволяє досягти високого рівня паралелізму та швидкодії. Однак, якщо потоки в межах одного варпу виконують різні гілки коду (наприклад, через умовні оператори), це може призвести до розбіжності варпу і зниження швидкодії.

CUDA також пропонує ієрархічну модель пам’яті, що включає глобальну, розділену, локальну та константну пам’ять. Ефективне використання цієї ієрархії є ключовим для оптимізації швидкодії CUDA програм. Наприклад, розділена пам’ять, яка є набагато швидшою за глобальну, може бути використана для

зменшення латентності доступу до даних в межах одного SM.

Ця архітектура створює унікальні виклики для програмістів, які повинні забезпечити ефективне управління пам'яттю та обробку помилок для досягнення оптимальної швидкодії в реальних умовах використання. Програмісти повинні враховувати розбіжність варпу при написанні коду для CUDA і намагатися мінімізувати її вплив на швидкодію, наприклад, шляхом реорганізації коду або використання спеціальних технік, таких як предикація.

Останні версії CUDA вводять нові можливості, такі як динамічний паралелізм, що дозволяє GPU запускати нові потоки без втручання CPU, та уніфікована віртуальна адресація, яка спрощує управління пам'яттю між CPU та GPU. Ці інновації дозволяють ще більше розширити спектр застосувань CUDA, особливо в сферах машинного навчання та обробки великих даних.

1.3.3 OpenCL

Відповіддю на розмаїття компіляторів та технологій у сфері оптимізації програмного забезпечення став стандарт OpenCL, розроблений The Khronos Group Inc [30]. OpenCL підтримується майже всіма основними виробниками апаратного забезпечення та надає уніфікований інтерфейс для паралельних обчислень на різноманітних платформах, включаючи GPU, DSP та CPU.

На відміну від CUDA, яка обмежена графічними процесорами NVIDIA, OpenCL пропонує кросплатформенність, дозволяючи розробникам створювати код, який може виконуватися на різних пристроях без значних змін. Це особливо цінно для проектів, які вимагають гнучкості та переносимості.

Порівняння OpenCL з CUDA виявляє кілька ключових відмінностей:

1. Підтримка пристроїв: OpenCL працює на широкому спектрі пристроїв, тоді як CUDA обмежена GPU NVIDIA.
2. Швидкодія: CUDA часто демонструє вищу швидкодію на GPU NVIDIA завдяки тісній інтеграції з апаратним забезпеченням.
3. Екосистема: CUDA має більш розвинену екосистему інструментів та бібліотек, особливо для задач машинного навчання.
4. Складність розробки: OpenCL може бути складнішим у використанні через свою універсальність, тоді як CUDA пропонує більш інтуїтивний API для GPU-обчислень.

Останні версії OpenCL (наприклад, OpenCL 3.0) впроваджують нові мо-

жливості, такі як підтримка графів обчислень та покращена інтеграція з мовами програмування високого рівня, що наближає його функціональність до CUDA.

Однак, незалежно від вибору між OpenCL та CUDA, обидві технології стикаються з проблемою накладних витрат на передачу даних між CPU та прискорювачами (GPU, DSP). Це може стати значною перешкодою для ефективності, особливо у задачах, де часта передача даних є критичною. Щоб мінімізувати цю проблему, програмісти часто вдаються до агрегації даних у великі пули, розміром 100-200 МБ, що може забезпечити до 20-кратного збільшення швидкодії порівняно з CPU.

Проте такий підхід не завжди оптимальний. Він ефективний для задач з обробкою великих обсягів даних, таких як навчання згорткових нейронних мереж (CNN), але може бути менш швидкодійним для обробки потокового відео з камери, де важлива швидкість реакції та мінімальна затримка.

Вибір між OpenCL та CUDA, а також стратегія оптимізації передачі даних, залежить від конкретних вимог проекту, цільових пристроїв та характеру обчислювальних задач. Це підкреслює необхідність глибокого розуміння специфіки задачі та вибору відповідних технологій та підходів для її оптимізації.

1.3.4 Бібліотеки OpenCV та ACL (Arm Compute Library)

Важливим аспектом підвищення швидкодії програмного забезпечення є правильний вибір компілятора, але це лише частина загальної картини. Інша суттєва частина полягає у використанні оптимізованого коду для часто використовуваних функцій та операцій, таких як ОЗ, масштабування та інші. Ці операції, важливі для обчислювальної швидкодії, часто реалізуються за допомогою бібліотек, які включають SIMD-оптимізований код для архітектур, таких як armeabi-v7a та arm64-v8a. Це особливо важливо у проектах, де ключовими є енергоефективність та швидкість обробки ЦЗ та відео.

OpenCV (Open Source Computer Vision Library) є однією з найпопулярніших бібліотек у галузі комп’ютерного зору та обробки ЦЗ [45]. Започаткована Intel у 1999 році, вона стала стандартом de facto для багатьох застосувань у сфері комп’ютерного зору [46]. OpenCV пропонує широкий спектр алгоритмів, включаючи:

1. Базові операції з зображеннями (фільтрація, перетворення кольорових просторів).

2. Аналіз ЦЗ (виявлення країв, кутів, ліній).
3. Обробка відео (відстеження об’єктів, вилучення фону).
4. Калібрування камери та 3D реконструкція.
5. Розпізнавання об’єктів.
6. Машинне навчання (SVM, k-NN, дерева рішень).

OpenCV оптимізована для різних архітектур ЦП та їх SIMD особливостей (SSE, AVX для x86 архітектури, NEON32 та NEON64 для ARM), що робить її ефективною на різноманітних платформах - від потужних серверів до мобільних пристроїв. Ця кросплатформенність є однією з ключових переваг OpenCV.

Arm Compute Library (ACL) [7, 8], з іншого боку, є спеціалізованою бібліотекою, розробленою компанією Arm для оптимізації обчислень на своїх процесорах. ACL фокусується на забезпеченні високошвидкодійних примітивів для машинного навчання та обробки ЦЗ, особливо для мобільних та вбудованих систем.

Порівняння OpenCV та ACL виявляє цікаві відмінності:

1. Сфера застосування: OpenCV є більш універсальною і охоплює ширший спектр задач комп’ютерного зору, тоді як ACL більше спеціалізується на оптимізації для конкретних архітектур Arm.
2. Оптимізація: ACL часто демонструє вищу швидкодію на Arm-процесорах завдяки глибокій оптимізації під конкретну архітектуру. Наприклад, вона у 14 разів швидша порівняно з OpenCV у завданнях CNN у одному потоці для 32-бітних операцій (була - станом на 2019 рік) [8].
3. Розширюваність: ACL вирізняється кращою розширюваністю завдяки своїй модульній архітектурі, що дозволяє легше додавати нові функції та оптимізації.
4. Підтримка спільноти: OpenCV має величезну спільноту розробників та велику кількість ресурсів, тоді як ACL має меншу, але більш спеціалізовану спільноту.

Механізм підвищення швидкодії включає не лише використання оптимізованих бібліотек, але й грамотне конфігурування проекту та використання вбудованих (інлайн) функцій. Цей підхід базується на практичному досвіді, де ми досліджували ефекти підвищення швидкодії ACL, використовуючи такі техніки:

1. Cache-friendly стиль написання коду, що є підхідом, котрий передбачає організацію даних та обчислень таким чином, щоб максимально ефективно

використовувати кеш-пам’ять процесора. Це може включати:

- Лінійне розміщення даних у пам’яті для зменшення кеш-промахів.
- Розбиття великих масивів даних на менші блоки, що поміщаються в кеш.
- Переупорядкування обчислень для максимізації повторного використання даних у кеші.

2. Використання лямбда-функцій і виразів з примусовою їх інтеграцією у код декілька разів (8 разів за раз): Ця техніка дозволяє компілятору краще оптимізувати код, вбудовуючи функції безпосередньо в місце виклику. Це зменшує накладні витрати на виклики функцій та покращує локальність коду. Але тим не менш підвищує поріг входження.
3. Векторизація обчислень: Використання SIMD інструкцій для паралельної обробки даних. Це особливо ефективно для операцій, які можна застосувати до багатьох елементів даних одночасно, як-от фільтрація ЦЗ або матричні операції.
4. Розгортання циклів: Техніка, яка зменшує кількість ітерацій циклу, виконуючи більше роботи в кожній ітерації. Це може покращити використання конвеєра процесора та зменшити накладні витрати на управління циклом.

Важливо відзначити, що оптимізації можуть по-різному впливати на різні архітектури. Наприклад, ми спостерігали загальне підвищення швидкодії для CPU ARM32, але її зменшення для ARM64 ЦП. Це пов’язано з оптимізацією циклів, що у свою чергу пов’язано з оптимізацією часових затримок на безумовний/умовний перехід ARM64, порівняно з ARM32. Такі відмінності підкреслюють важливість тестування оптимізацій на цільових платформах.

Для досягнення максимальної швидкодії при роботі з такими бібліотеками рекомендується:

1. Використовувати останні версії бібліотек, які включають найновіші оптимізації (але треба завжди фіксувати код на тег комітах, що у свою чергу є ознакою фіксування версії ПП). Регулярні оновлення можуть принести значні покращення швидкодії.
2. Враховувати особливості цільової архітектури при виборі алгоритмів та налаштуванні параметрів. Наприклад, для ARM-процесорів важливо враховувати особливості NEON інструкцій.
3. Профілювати код для виявлення вузьких місць та можливостей оптимізації.

Інструменти профілювання можуть виявити неочевидні проблеми швидкодії. Однак, вибір інструменту профілювання є критичним, особливо для специфічних архітектур. Саме тому у цьому дослідженні використовувався самописний інструмент профілювання, а не Valgrind (Callgrind).

4. Розглянути можливість використання GPU-прискорення для особливо ресурсомістких операцій, якщо це доступно на цільовій платформі. Наприклад, OpenCV має модуль cuda для роботи з NVIDIA GPU.
5. Експериментувати з різними налаштуваннями компілятора та прапорцями оптимізації. Іноді незначні зміни в налаштуваннях можуть дати суттєвий приріст швидкодії.
6. Використовувати техніки паралельного програмування, такі як OpenMP або TBB, для ефективного використання багатоядерних процесорів.

Дещо розглянемо профілювання коду для цього дослідження. Valgrind [47, 48], хоча і є потужним інструментом для x86 архітектури, має обмеження при роботі з ARM. Він працює у ”віртуальному”середовищі, що для ARM архітектури не є показовим (досі не визначено, коли це станеться). Це може призвести до неточних результатів, особливо коли йдеться про оптимізації низького рівня та специфічні для ARM інструкції, такі як NEON.

Самописний інструмент профілювання дозволяє:

1. Точно вимірювати швидкодію безпосередньо на цільовій ARM платформі.
2. Враховувати специфічні особливості ARM архітектури, такі як різниця між ARM32 та ARM64.
3. Мінімізувати накладні витрати на профілювання, що особливо важливо для вбудованих систем з обмеженими ресурсами.
4. Налаштовувати процес профілювання під конкретні потреби дослідження, фокусуючись на релевантних метриках.

Цей підхід, хоча і вимагає більше зусиль на розробку, забезпечує більш точні та релевантні дані для оптимізації на ARM платформах.

Отже тепер продовжимо огляд і вкажемо, що було використано бібліотеку - OpenCV - як еталони для порівняння та аналізу швидкодії нашого коду. Це дозволить нам зробити обґрунтовані висновки щодо ефективності різних підходів до оптимізації. Порівняння з цією бібліотекою надасть цінну інформацію про те, наскільки наші власні оптимізації ефективні порівняно з індустріальними стандартами.

У майбутньому, з розвитком нових архітектур процесорів та спеціалізованих прискорювачів, можна очікувати подальшого вдосконалення цих бібліотек. Зокрема, інтеграція з нейронними прискорювачами та більш ефективне використання гетерогенних обчислювальних систем може відкрити нові можливості для оптимізації обробки ЦЗ та задач комп'ютерного зору.

1.3.5 Ігровий двигун id Tech 3

Ігровий двигун id Tech 3 є важливим прикладом використання оптимізацій, як алгоритмічних, так і SIMD, у відкритому програмному забезпеченні. Цей двигун, розроблений компанією id Software, став ключовою віхою в історії ігрової індустрії, демонструючи потенціал передових технологій оптимізації для створення більш реалістичних та динамічних ігрових світів.

Історія розробки id Tech 3 бере свій початок від його попередників - id Tech 1 (використаного в Doom) та id Tech 2 (Quake II). Кожна ітерація двигуна приносила значні покращення в графіки та швидкодії. id Tech 3, випущений у 1999 році для гри Quake III Arena, став революційним кроком вперед, особливо в аспекті оптимізації для тогочасного апаратного забезпечення [49, 50].

Переходячи до аналізу сторонніх бібліотек та підходів до оптимізації, варто звернути увагу на те, як SIMD-оптимізації впроваджуються у різноманітні програмні продукти, від пропрієтарних до відкритих. Наприклад, ядро Windows 10/11 використовує AVX(1|2) для підвищення загальної швидкодії системи, тоді як Oracle Java VM застосовує AVX(1|2)/3DNow/ARM $\text{NEON}_{(32|64)}$, що сприяє прискоренню виконання Java-додатків за рахунок швидкої роботи Java-машини [17, 21, 25]. Це підкреслює важливість SIMD-оптимізацій у сучасному програмному забезпеченні.

id Tech 3 став одним з перших ігрових двигунів, який широко застосовував SIMD (Single Instruction, Multiple Data) інструкції, зокрема SSE (Streaming SIMD Extensions) від Intel [49, 50]. Ці оптимізації дозволили значно підвищити швидкодію у ключових областях рендерингу та фізичних розрахунків. SIMD інструкції дозволяли обробляти кілька елементів даних одночасно, що було особливо корисно для операцій з векторами та матрицями, які є основою 3D графіки.

Ігрові двигуни, такі як id-Tech 2-4, використовувані в іграх на кшталт Quake III Arena, є яскравими прикладами відкритих проектів, оптимізованих

під Intel SIMD - SSEn. Ці двигуни демонструють, як оптимізації SIMD можуть значно покращити швидкодію ігрових додатків, забезпечуючи більш плавну та реалістичну графіку. Однак, використання SIMD в ігрових двигунах також ставить перед розробниками виклик перетворення коду з плаваючою комою [51, 52] в код з фіксованою точкою [53, 54], зберігаючи при цьому прийнятний рівень точності та використання алгоритмічних «хаків». Це завдання може бути досить складним, оскільки вимагає від розробників глибокого розуміння як математичних основ, так і особливостей архітектури процесора.

Проблема перетворення коду з рухомою комою в код з фіксованою точкою є особливо актуальною у контексті оптимізацій SIMD, оскільки SIMD-операції часто ефективніші при роботі з даними фіксованої точності. Причиною того є зменшення обсягу пам’яті, що займає одна одиниця розрахунку (як змінна). Приклад дуже простий – uint16_t та float займають 2 та 4 байти відповідно. Тож зменшення обсягу даних при подальшій SIMD оптимізації є доволі очевидним. Це перетворення вимагало від розробників id Tech 3 знаходити тонкий баланс між точністю обчислень та швидкодією, що стало ключовим аспектом при розробці двигуна.

Розробники id Tech 3 використовували ряд інноваційних ”алгоритмічних хаків”для досягнення максимальної швидкодії. Одним з найвідоміших прикладів є алгоритм швидкого обчислення оберненого квадратного кореня, який став легендарним в індустрії. Цей алгоритм використовував особливості представлення чисел з рухомою комою для отримання приблизного результату з високою швидкістю, що було критично важливим для 3D графіки реального часу.

В той же час, в пропрієтарному програмному забезпеченні, де вартість розробки оптимізації є високою, багато з цих оптимізацій залишаються комерційними секретами. Це створює певну неоднорідність у доступі до передових технологій оптимізації між відкритими та пропрієтарними проектами. Така ситуація вимагає від розробників відкритого програмного забезпечення бути більш інноваційними та креативними у пошуку рішень, що дозволяють досягти високої швидкодії без доступу до закритих технологій.

З іншого боку, це також стимулює спільноту відкритого програмного забезпечення до співпраці та обміну знаннями, що веде до створення більш гнучких та адаптивних рішень. Відкриті проекти, такі як ігрові двигуни id Tech, часто стають платформами для експериментів та інновацій, дозволяючи розробникам

випробовувати нові підходи до оптимізації та вдосконалення швидкодії. Це, в свою чергу, сприяє загальному розвитку галузі, оскільки інновації, розроблені у відкритих проектах, часто адаптуються та використовуються у комерційних продуктах.

Оптимізація SIMD у відкритому програмному забезпеченні також відкриває можливості для освітніх та дослідницьких цілей. Студенти, вчені та незалежні розробники мають можливість вивчати та експериментувати з цими технологіями, що, в свою чергу, може призвести до розвитку нових, більш ефективних методів оптимізації, які будуть доступні широкому колу розробників.

У підсумку, хоча пропрієтарне програмне забезпечення часто має перевагу у вигляді доступу до передових технологій та високого бюджету на розробку, відкриті проекти, такі як ігрові двигуни id Tech, вносять значний вклад у розвиток технологій оптимізації. Інновації, започатковані в id Tech 3, продовжують впливати на сучасне програмування, демонструючи важливість відкритого обміну знаннями та технологіями для прогресу в галузі оптимізації програмного забезпечення.

1.3.6 Розгортання циклів

Розгортання циклів є однією з ключових технік оптимізації програмного коду, яка спрямована на підвищення швидкодії шляхом зменшення накладних витрат, пов’язаних з ітераціями циклу. У типовому циклі (наприклад, for або while) кожна ітерація супроводжується перевіркою умови завершення та оновленням лічильника, що реалізується через умовні переходи (branches) і арифметичні операції. Ці накладні витрати стають особливо відчутними у випадках, коли тіло циклу є невеликим, а кількість ітерацій значною. Розгортання циклів дозволяє замінити ітеративну конструкцію на лінійну послідовність команд, усуваючи або значно скорочуючи кількість умовних переходів.

На архітектурі ARM64, яка базується на RISC-принципах (Reduced Instruction Set Computer), кожна інструкція виконується за фіксований час, а швидкодія залежить від ефективності конвеєра команд. Умовні переходи, характерні для циклів, можуть призводити до зривів конвеєра (pipeline stalls) через помилки передбачення гілок (branch misprediction), що знижує загальну пропускну здатність процесора. Розгортання циклів мінімізує ці ефекти, дозволяючи процесору виконувати більше корисної роботи за один цикл.

Поряд з розгортанням циклів часто застосовується техніка плитковості [5, 55–57], яка тісно пов’язана з оптимізацією використання кеш-пам’яті. Плитковість передбачає розбиття великих масивів даних на менші ”плитки які краще відповідають розміру кеш-лінії процесора. Це дозволяє ефективніше використовувати кеш-пам’ять, зменшуючи кількість кеш-промахів та прискорюючи доступ до даних. Комбінація розгортання циклів та плитковості може призвести до значного підвищення швидкодії, особливо для алгоритмів, що працюють з великими обсягами даних, таких як матричні операції або обробка ЦЗ.

Розгортання циклів має ряд переваг:

1. Зменшення накладних витрат на керування циклом. У кожному циклі процесор витрачає час на оновлення лічильника (наприклад, інструкція ADD) та перевірку умови завершення (інструкція CMP з подальшим B.NE для умовного переходу). Для невеликих тіл циклів ці операції можуть становити значну частину загального часу виконання. Розгортання усуває ці витрати, замінюючи їх прямими обчисленнями. Наприклад, цикл із чотирма ітераціями:

```
for (int i = 0; i < abc_length; i++) {
    a[i] = b[i] * c[i];
}
```

після експертного розгортання буде мати вигляд:

```
for (int i = 0; i < abc_length / 4; i++) {
    a[0] = b[0] * c[0];
    a[1] = b[1] * c[1];
    a[2] = b[2] * c[2];
    a[3] = b[3] * c[3];
    a += 4; b += 4; c += 4;
}
for (int i=0, j = abc_length % 4; j; --j, ++i) {
    a[i] = b[i] * c[i];
}
```

У цьому випадку усувається три умовних переходи та три оновлення лічильника, що на ARM64 може заощадити до 12–15 тактів залежно від моделі процесора.

2. Оптимізація конвеєра. Умовні переходи викликають затримки через необхідність прогнозування гілок. На ARM64 навіть із вдосконаленими предикторами (наприклад, TAGE у Cortex-A77) помилка передбачення призводить до значної втрати швидкодії. Розгортання циклів зменшує кількість гілок, дозволяючи конвеєру працювати без перерв, що особливо важливо для коротких циклів із передбачуваною кількістю ітерацій.

3. Покращення використання NEON. Розгортання циклів відкриває можли-

вості для векторизації. Наприклад, замість обробки одного елемента за ітерацію можна завантажити та обробити чотири елементи одночасно за допомогою 128-бітних NEON-регістрів (Q-регістрів). Це зменшує загальну кількість інструкцій і підвищує пропускну здатність обчислень, що є критичним для застосунків, де є важливою максимальна швидкодія, таких як обробка сигналів чи машинне навчання.

4. Зменшення накладних витрат на кеш і пам’ять. Розгорнутий код сприяє лінійному доступу до даних, що оптимізує використання кеш-пам’яті та механізмів апаратної вибірки. На ARM64 це дозволяє уникнути пропусків кешу (cache misses), які можуть коштувати десятки циклів затримки при зверненні до оперативної пам’яті.

Чому це є оптимальним рішенням для швидкодіючого коду?

1. Збільшення швидкості виконання. Усунення умовних переходів і накладних витрат на керування циклом дозволяє значно скоротити час виконання, особливо для циклів із невеликим тілом. Наприклад, для циклу з 10 ітерацій та тілом, що складається з однієї операції множення, розгортання може зменшити загальну кількість циклів на **20–30%** завдяки економії на гілках і оновленнях лічильника.

2. Максимізація паралелізму. Розгорнутий код полегшує компілятору та апаратному забезпеченню використання паралельної обробки команд (instruction-level parallelism, ILP). На ARM64, де ядра підтримують суперскалярне виконання (наприклад, до 4–6 інструкцій за цикл у Cortex-A78), це дозволяє одночасно виконувати кілька операцій із розгорнутого циклу, підвищуючи загальну пропускну здатність.

3. Ефективність у реальних задачах. У програмах, таких як кодеки, криптографічні алгоритми чи обчислення з рухомою комою, цикли часто є вузьким місцем. Розгортання циклів на ARM64 дозволяє оптимізувати ці критичні ділянки коду, що призводить до суттєвого приросту швидкодії без значних змін у логіці програми.

4. Компроміс із розміром коду. Хоча розгортання циклів збільшує розмір машинного коду, сучасні ARM64-процесори мають достатньо великі кеші команд (наприклад, 64 КБ L1 у Cortex-A76), що мінімізує вплив цього фактора. У програмуванні швидкість виконання зазвичай є пріоритетнішою за розмір коду, особливо в системах із достатньою пам’яттю.

```
1  template <unsigned N>
2  struct func_unroll {
3      template <typename F>
4      static inline void call(F const& f) {
5          f();
6          func_unroll<N - 1>::call(f);
7      }
8  };
9
10 template <> struct func_unroll<0u> {
11     template <typename F>
12     static inline void call(F const&){}
13 };
14
15
16 struct do_unroll {
17     template <typename F, unsigned unrollDelta>
18     static inline void run(F const& f,
19     int &baseStep, int &restSteps) {
20         for (int j = baseStep; j; --j) {
21             func_unroll<unrollDelta>::call(f);
22         }
23         for (int j = restSteps; j; --j) {
24             f();
25         }
26     }
27 };
```

Рисунок 1.1 — Розгортання циклу за допомогою шаблонів C++

Архітектура ARM64, зокрема з розширенням NEON для SIMD-обчислень, має кілька характеристик, які роблять розгортання циклів особливо доцільним:

1. Конвеєрна обробка: Сучасні ARM64-процесори (наприклад, Cortex-A76, A78) використовують глибокий конвеєр (до 13 етапів у деяких моделях), що підвищує пропускну здатність за умови безперервного потоку команд. Умовні переходи в циклах порушують цей потік, викликаючи затримки, які розгортання циклів може усунути.

2. Передбачення гілок: ARM64-процесори оснащені механізмами передбачення гілок, але навіть сучасні алгоритми передбачення мають обмежену точність (зазвичай 90–95%). Кожна помилка передбачення призводить до очищення конвеєра та затримки в 10–20 циклів залежно від моделі ядра.

3. SIMD-можливості: Розширення NEON дозволяє обробляти кілька елементів даних однією інструкцією. Розгортання циклів полегшує використання векторних інструкцій, таких як vld1 (завантаження вектора) чи vmla (векторне множення з додаванням), зменшуючи кількість ітерацій і підвищуючи паралелізм даних;

4. Кеш-пам’ять: ARM64-процесори мають багаторівневу кеш-пам’ять (L1,

L2, іноді L3), яка чутлива до шаблонів доступу до даних. Розгортання циклів сприяє послідовному доступу до пам’яті, що покращує ефективність кешу та апаратної вибірки даних (hardware prefetching) [8, 57].

Розгортання циклів зазвичай виконується за допомогою компілятора або шляхом введення відповідного вбудованого асемблерного коду безпосередньо в додаток. Деякі бібліотеки, як-от ACL, можуть скористатися перевагами особливостей високорівневої мови програмування (наприклад, шаблонів у C++), щоб виконати розгортання циклів. У лістингу надано спрощений код в стилі ACL, щоб продемонструвати приклад реалізації техніки статичної Loop Unrolling оптимізації на рівні (з використанням) лямбда-функції ”обгортки”(див. рис. 1.1) [58].

Використання шаблонів C++ для статичного розгортання циклів, як показано на (див. рис. 1.1), має кілька ключових переваг:

1. Гнучкість: Дозволяє легко змінювати ступінь розгортання (unrollDelta) без зміни основної структури коду.
2. Оптимізація на етапі компіляції: Компілятор може застосувати додаткові оптимізації, знаючи точну кількість ітерацій на етапі компіляції.
3. Уникнення надлишкового коду: За допомогою шаблонів можна генерувати лише необхідну кількість розгорнутих ітерацій, уникаючи непотрібного дублювання коду.
4. Збереження читабельності: Основна логіка залишається компактною та зрозумілою, незважаючи на розгортання.

Значним плюсом такого підходу є надання можливості компілятору більш ефективно оптимізувати код, оскільки кількість ітерацій відома заздалегідь (unrollDelta = 8). Передавши відповідні параметри до do_unroll<...>::run(...) з (рис. 1.1), можна викликати функцію f(...) baseStep$\times$unrollDelta+restSteps разів, уникаючи порівнянь та декрементів лічильника unrollDelta-1. Досягнуте покращення швидкодії більш детально обговорюється в [8].

Ефективність розгортання циклів може суттєво відрізнятися залежно від архітектури процесора. Наприклад, на ARM32 ця техніка показала себе досить ефективною. Однак, на ARM64 ситуація дещо інша через суттєві відмінності в архітектурі. ARM64 має оптимізований механізм обробки переривань та умовних переходів, що може зробити прості цикли такими ж ефективними, як і розгорнуті варіанти.

Для дослідження ефективності розгортання циклів на ARM64 було проведено експеримент, в якому порівнювалася швидкість роботи двох еквівалентних функцій: однієї з розгортанням циклу, а іншої - без нього. Експеримент проводився на ARM64 ЦП з використанням операційної системи Ubuntu 22.04. Методологія включала [8]:

1. Реалізацію двох версій функції: з розгортанням циклу та без нього.
2. Компіляцію обох версій з використанням компілятора clang-11 з оптимізаціями.
3. Виконання бенчмарків для вимірювання часу виконання кожної версії.
4. Аналіз згенерованого асемблерного коду за допомогою IDA для підтвердження факту розгортання.

Результати експерименту виявилися несподіваними: функція з розгортанням циклу показала зниження швидкодії на 3-5%. Аналіз асемблерного коду підтвердив, що цикл дійсно був розгорнутий, і тіло ресурсомісткої частини функції повторювалося 8 разів. Однак, було виявлено, що компілятор додав надлишкові порівняння навколо розгорнутого коду, що і призвело до зниження швидкодії.

Важливо зазначити, що сучасні компілятори, такі як GCC та Clang, мають вбудовані можливості для автоматичного розгортання циклів. Наприклад, прапори компіляції -O3 -funroll-loops в обох компіляторах активують цю оптимізацію. Однак, як показав наш експеримент, автоматичне розгортання не завжди призводить до підвищення швидкодії, особливо на сучасних архітектурах, таких як ARM64.

Цей експеримент підкреслює важливість врахування специфіки цільової архітектури при застосуванні оптимізаційних технік. Те, що працює ефективно на одній архітектурі (наприклад, ARM32), може бути менш ефективним або навіть контрпродуктивним на іншій (ARM64). Це також демонструє необхідність емпіричного тестування оптимізацій на конкретній цільовій платформі, а не сліпого слідування загальним рекомендаціям.

У контексті розробки високошвидкодійного програмного забезпечення, особливо для різних архітектур, важливо:

1. Розуміти основні принципи роботи цільової архітектури.
2. Проводити емпіричні тести для кожної оптимізації на конкретній платформі.

3. Не покладатися виключно на автоматичні оптимізації компілятора, але і не ігнорувати їх повністю.
4. Враховувати, що оптимізації, ефективні для однієї архітектури, можуть бути неефективними для іншої.

Отже, хоча розгортання циклів залишається потужною технікою оптимізації, її застосування вимагає ретельного аналізу та тестування на цільовій платформі для досягнення оптимальних результатів.

1.3.7 Кеш-френдлі оптимізація коду

Кеш-френдлі оптимізація коду є критично важливою технікою для підвищення швидкодії сучасних програм, особливо тих, що працюють з великими обсягами даних [4]. Ця техніка базується на розумінні архітектури кеш-пам'яті процесора та оптимізації доступу до даних для максимально ефективного використання кешу [59].

```
void cachefr_mat_mul(int n, double *A, double *B, double *C, int block_size) {
    for (int i = 0; i < n; i += block_size) {
        for (int j = 0; j < n; j += block_size) {
            for (int k = 0; k < n; k += block_size) {
                for (int ii = i; ii < min(i + block_size, n); ii++) {
                    for (int jj = j; jj < min(j + block_size, n); jj++) {
                        for (int kk = k; kk < min(k + block_size, n); kk++) {
                            C[ii * n + jj] += A[ii * n + kk] * B[kk * n + jj];
                        }
                    }
                }
            }
        }
    }
}
```

Рисунок 1.2 — Приклад реалізації кеш-френдлі множення матриць

Основні принципи кеш-френдлі програмування включають:

1. Просторова локальність: Організація даних таким чином, щоб пов'язані елементи знаходились поруч у пам'яті, що збільшує ймовірність їх одночасного завантаження в кеш.
2. Часова локальність: Повторне використання даних, які вже знаходяться в кеші, перш ніж вони будуть витіснені.
3. Мінімізація кеш-промахів: Зменшення кількості випадків, коли процесору потрібно звертатися до основної пам'яті через відсутність даних в кеші.
4. Вирівнювання даних: Розміщення даних за адресами, кратними розміру кеш-лінії, для оптимізації завантаження в кеш.

Одним з класичних прикладів застосування кеш-френдлі оптимізації є множення матриць (рис. 1.2). Традиційний алгоритм множення матриць може бути неефективним з точки зору використання кешу, особливо для великих матриць. Кеш-френдлі підхід передбачає розбиття матриць на менші блоки, розмір яких відповідає розміру кеш-пам'яті.

Цей підхід, відомий як блочне множення матриць, значно покращує локальність даних і зменшує кількість кеш-промахів, особливо для великих матриць [60].

Інші техніки кеш-френдлі оптимізації включають:

1. Попереднє завантаження даних (prefetching): Завантаження даних у кеш до того, як вони знадобляться, що може бути реалізовано програмно або за допомогою спеціальних інструкцій процесора [61].

2. Оптимізація структур даних: Розміщення часто використовуваних полів структур поруч один з одним для покращення просторової локальності.

3. Розгортання циклів з урахуванням розміру кеш-лінії: Це дозволяє ефективніше використовувати завантажені в кеш дані.

4. Використання технік, таких як "структури масивів"замість "масивів структур"для покращення доступу до даних у певних сценаріях.

Важливо зазначити, що ефективність кеш-френдлі оптимізацій може варіюватися залежно від конкретної архітектури процесора та розміру його кеш-пам'яті. Тому критично важливо проводити профілювання та бенчмаркінг на цільовій платформі для визначення оптимальних параметрів [8, 22, 62, 63], у тому числі для розмірів блоку в блочному множенні матриць [64–66].

Кеш-френдлі оптимізація часто вимагає компромісу між читабельністю коду та швидкодією. У багатьох випадках оптимізований код може бути менш інтуїтивним і складнішим для розуміння. Тому важливо документувати такі оптимізації та, можливо, зберігати неоптимізовані версії коду для порівняння та налагодження.

У контексті сучасних процесорних архітектур, особливо в мобільних та вбудованих системах, де енергоефективність є критичною, кеш-френдлі оптимізації набувають ще більшого значення. Ефективне використання кеш-пам'яті не лише підвищує швидкодію, але й може значно знизити енергоспоживання, що є ключовим фактором для багатьох сучасних застосувань.

Розуміння та застосування принципів кеш-френдлі програмування є ва-

жливим навиком для розробників, які прагнуть створювати високошвидкодійне програмне забезпечення. Це особливо актуально в областях, де обробляються великі обсяги даних, таких як наукові обчислення, обробка ЦЗ та відео, машинне навчання та аналіз даних.

1.4 Узагальнений огляд SIMD-реалізації cv::filter2D() на основі OpenCV Universal Intrinsics

Огляд альтернативних методів векторізації/оптимізації коду слід продовжити з OpenCV Universal Intrinsics. OpenCV Universal Intrinsics є тонким абстрактним шаром, що надає єдиний набір SIMD-примітивів для різних наборів інструкцій (SSE/AVX, NEON, VSX) [67]. Детальний перелік векторних типів, функцій завантаження, розширення, FMA-обчислень та пакування подано в офіційній документації і у внутрішньому технічному аналізі [68].

Таблиця 1.1 — Стек викликів `cv::filter2D(…)` до моменту звернення до векторизованої функції `FilterVec_8u::operator()`

№	Функція	Опис
1	filter2D(…)	Зовнішній інтерфейс
2	cv::hal::filter2D(…)	Вибір реалізації HAL
3	cv::ocvFilter2D(…)	Підготовка ядра та буферів
4	cv::FilterEngine::apply(…)	Ковзання ядра по зображенню
5	cv::cpu_baseline::FilterEngine(…)	Передавання у SIMD-ядро
6	cv::cpu_baseline::Filter2D<…>::operator()	Підготовка до векторизації
7	cv::cpu_baseline::FilterVec_8u::operator()	Безпосереднє SIMD-обчислення

Переваги підходу OpenCV Universal Intrinsics:

1. Переносимість - один код охоплює x86-64, ARM і POWER без дублювання.
2. Швидкодія - використовуються найширші доступні регістри та FMA-інструкції.
3. Масштабованість - Поява нових ISA (наприклад, AVX-512, SVE) потребує лише розширення UI-шару.

4. Точність - обчислення у float32 усувають переповнення та підтримують дробові ядра.

У звичайному виклику cv::filter2D(…) стек виконує близько восьми шарів, доки не дійде до спеціалізованого обробника 8-бітних даних, як показано у (табл. 1.1).

Алгоритмічний контур FilterVec_8u є наступним:

1. Завантаження блоків пікселів у SIMD-регістри.
2. Розширення 8-бітних значень до розрядів, придатних для обчислень.
3. Перетворення у float32 та FMA-акумуляція з коефіцієнтами ядра.
4. Упаковка результатів у вихідний 8-бітний формат з насиченням.

Ці операції реалізовано виключно через функції UI, а конкретні інструкції (AVX-FMA, NEON-FML тощо) обираються компілятором згідно з наявними SIMD-розширеннями [60].

Отже, застосування OpenCV UI в cv::filter2D() демонструє сучасну стратегію оптимізації: універсальний SIMD-код, автоматично адаптований до апаратних можливостей, що забезпечує високу швидкодію та спрощує супровід бібліотеки.

1.5 Висновки

У першому розділі даного дослідження проведено комплексний аналіз сучасних підходів до оптимізації програмного забезпечення для ОЦЗ, з акцентом на підвищення швидкодії ОЗ ЦЗ при використанні 16-бітних цілочисельних обчислень. На основі проведеного дослідження зроблено наступні висновки:

1. Аналіз сучасних тенденцій оптимізації ПЗ показав, що для досягнення максимальної швидкодії в ресурсоємних задачах ОЦЗ, таких як ОЗ, покладання виключно на автоматичну векторизацію компіляторів (навіть з високими рівнями оптимізації -O3/-Ofast) часто є недостатнім. Для повного розкриття потенціалу сучасних процесорних архітектур, зокрема ARM64, необхідне застосування низькорівневих технік оптимізації, таких як використання SIMD-інтрінсиків або Inline Assembly.

2. Дослідження апаратних засобів прискорення виявило, що технологія SIMD є ключовим фактором для прискорення ОЗ. Для архітектури ARM64 центральну роль відіграє набір інструкцій NEON64, який забезпечує ефективну

обробку векторних даних, що є критично важливим для ОЗ ЦЗ. Порівняльний аналіз з іншими SIMD-архітектурами (наприклад, Intel AVX), а також з альтернативними прискорювачами (GPU з моделлю SIMT, спеціалізовані DSP типу Qualcomm Hexagon), підкреслив переваги NEON64 для мобільних та вбудованих систем завдяки балансу швидкодії та енергоефективності.

3. Огляд ПЗ оптимізації підтвердив важливу роль компіляторів (GCC, Clang/LLVM) у генерації ефективного коду, проте їх можливості авто-векторизації для складних алгоритмів, таких як ОЗ ЦЗ з різними типами даних та умовами, можуть бути обмежені. Спеціалізовані бібліотеки, такі як OpenCV та ACL, надають оптимізовані реалізації багатьох функцій обробки ЦЗ, часто використовуючи SIMD-інструкції. Однак їх універсальність може призводити до додаткових накладних витрат порівняно зі спеціалізованим, вручну оптимізованим кодом.

4. Аналіз низькорівневих технік оптимізації, таких як розгортання циклів (loop unrolling) та кеш-френдлі організація доступу до даних, показав їх значний потенціал для підвищення швидкодії. Водночас, ефективність цих технік суттєво залежить від специфіки архітектури процесора (зокрема, конвеєра та системи кеш-пам’яті ARM64) та характеристик компілятора, що вимагає ретельного тестування та профілювання для уникнення контрпродуктивних ефектів (наприклад, надлишкового розгортання циклів).

5. Розгляд реалізації функції cv::filter2D з бібліотеки OpenCV на основі Universal Intrinsics продемонстрував підхід до створення переносимого SIMD-коду. Було виявлено, що cv::filter2D використовує складний стек викликів та внутрішню конвеєрну обробку даних (наприклад, 8-бітні вхідні дані -> 16-бітні проміжні -> 32-бітні з плаваючою комою для акумуляції -> 8-бітні вихідні), що може вносити певні накладні витрати, але забезпечує універсальність. Також відзначено використання різних алгоритмічних шляхів залежно від розміру ядра згортки, зокрема можливий перехід на FFT-алгоритми для великих ядер.

Загалом, проведений у першому розділі аналіз обґрунтовує необхідність розробки спеціалізованого методу оптимізації ОЗ ЦЗ для 16-бітних цілочисельних обчислень на архітектурі ARM64 з використанням SIMD NEON64, котрий поєднував би переваги низькорівневого контролю над апаратними ресурсами та специфічну адаптацію до особливостей оброблюваних даних, що і стало предметом подальших досліджень у дисертаційній роботі.

РОЗДІЛ 2. ЛІНІЙНІ ОПЕРАТОРИ ДЛЯ ОБРОБКИ ЦИФРОВИХ ЗОБРАЖЕНЬ ПІД ЧАС 16-БІТНИХ ОБЧИСЛЕНЬ

У розділі подано лінійні оператори на основі ОЗ, що можуть бути використані під час 16-бітних обчислень. Сформульовано умови, яким мають відповідати ЯЗ для 16-бітних обчислень (підрозд. 2.1). Надано оператори низькочастотної фільтрації, контрастування, цифрової стабілізації ЦЗ, а також оператори двовимірного двократного збільшення лінійних розмірів ЦЗ (підрозд. 2.2). Подано лінійні оператори для визначення особливостей ЦЗ, які можуть мати застосування в задачі пошуку схожих фрагментів на різних цифрових знімках. Проведено детальний аналіз переваг беззнакових обчислень для використання SIMD NEON64 операцій (підрозд. 2.3), крім того розглянуто методику адаптації ЯЗ з від'ємними елементами для беззнакових обчислень (підрозд. 2.4). Розглянуто підходи до оптимізації обчислень, де необхідно перевіряти потрапляння в діапазон від 0 до 255 (підрозд. 2.5). Сформульовано висновки за розділом (підрозд. 2.6).

2.1 Умова відповідності маски фільтра згортки для 16-бітних обчислень

Не втрачаючи загальності, нехай задано деякий безрозмірний растр ЦЗ, кожному пікселю якого поставлено у відповідність пару індексів $\{(i,j)\}_{i,j\in\mathbb{Z}}$, що визначають його розташування. Позначимо $\{p_{i,j}\}_{i,j\in\mathbb{Z}}$ — для запису обчислювальної схеми при роботі з послідовностями кольорових складових (червоною, зеленою та синьою) або із зображенням у відтінках сірого [69].

У загальному випадку визначатимемо оператор згортки так:

$$F\left(p^{i,j}\right)=\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}\gamma_{ii-i,jj-j}p_{ii,jj} \tag{2.1}$$

де: $p_{i,j}$ — кольорова складова растру; (i,j) — індекс пікселя растру; $\gamma_{ii-i,jj-j}$ — елемент маски фільтра; $(2r_i+1)\times(2r_j+1)$ — розмір маски фільтра.

Загальновідомо, що 16-бітне ціле число може зберігати $2^{16}=65536$ унікальних значень. Якщо припускати, що значення можуть бути від'ємними, то 16-бітне ціле число може набувати значень від -32768 до 32767 [54, 70]. З огляду

на зазначене, нескладно сформулювати обмеження на коефіцієнти маски фільтра ОЗ (2.1) під час 16-бітних обчислень. Зокрема, з урахуванням того, що $p_{ii,jj}$ — є цілочисельним у діапазоні від 0 до 255, можемо записати:

$$\left| \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j} p_{ii,jj} \right| \leq 255 \cdot \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} |\gamma_{ii-i,jj-j}| \tag{2.2}$$

Отже, для забезпечення можливості виконання 16-бітної операції достатньо вимагати, щоб елементи маски (2.1) були типу ShortInt [54, 71] та при реалізації виразу (2.2) виконувалася нерівність

$$\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} |\gamma_{ii-i,jj-j}| \leq 127 \tag{2.3}$$

У подальшому викладі подамо приклади масок, що задовольняють умові (2.3) під час вирішення різноманітних задач цифрової обробки ЦЗ при 16-бітних обчисленнях [3]. Фактично, необхідно або перевірити виконання нерівності (2.3) для відомих операторів, або отримати нові маски для вирішення конкретних задач за згаданим підходом.

2.2 Фільтрація та масштабування зображень

Для реалізації субполосної фільтрації послідовності $\{p_{i,j}\}_{i,j\in\mathbb{Z}}$ припускаємо, що значення інтенсивності у кожному пікселі можна подати у вигляді суми [69]:

$$p_{i,j} = pL_{i,j} + pH_{i,j},$$

де: $pL_{i,j}$, $pH_{i,j}$ - відповідно, низько- та високочастотні складові, які можна визначати на основі таких лінійних операторів [72]:

$$pL_{i,j} = L\left(p^{i,j}\right) = \sum_{ii=i-1}^{i+1} \sum_{jj=j-1}^{j+1} \gamma L_{ii-i,jj-j} p_{ii,jj}, i, j \in \mathbb{Z},$$

$$pH_{i,j} = H\left(p^{i,j}\right) = \sum_{ii=i-1}^{i+1} \sum_{jj=j-1}^{j+1} \gamma H_{ii-i,jj-j} p_{ii,jj}, i, j \in \mathbb{Z},$$

де

$$\gamma L = \frac{1}{64}\begin{pmatrix} 1 & 6 & 1 \\ 6 & 36 & 6 \\ 1 & 6 & 1 \end{pmatrix}; \quad \gamma H = \frac{1}{64}\begin{pmatrix} -1 & -6 & -1 \\ -6 & 28 & -6 \\ -1 & -6 & -1 \end{pmatrix} \tag{2.4}$$

або

$$\gamma L = \frac{1}{36}\begin{pmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{pmatrix}; \quad \gamma H = \frac{1}{36}\begin{pmatrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{pmatrix}. \tag{2.5}$$

Зауважимо, що тут і далі в роботі виконання умови (2.3) вимагається для цілочисельних елементів масок згорток (2.4–2.5) до ділення на величину, винесену в знаменники. Також зазначимо, що маска γL в (2.5) забезпечує більший ступінь згладжування, ніж для (2.4); це пов'язано з тим, що вона отримана [73, 74] на основі лінійної комбінації *B*-сплайнів 3-го порядку, на відміну від *B*-сплайнів 2-го порядку, що мають менші згладжувальні властивості [75].

Якщо ж необхідно забезпечити більшу, ніж у (2.5), ступінь згладжування, то можна запропонувати оператори

$$pL_{i,j} = L\left(p^{i,j}\right) = \sum_{ii=i-2}^{i+2}\sum_{jj=j-2}^{j+2} \gamma L_{ii-i,jj-j} p_{ii,jj}, i, j \in \mathbb{Z},$$

$$pH_{i,j} = H\left(p^{i,j}\right) = \sum_{ii=i-2}^{i+2}\sum_{jj=j-2}^{j+2} \gamma H_{ii-i,jj-j} p_{ii,jj}, i, j \in \mathbb{Z},$$

де:

$$\gamma L = \frac{1}{121}\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 6 & 15 & 6 & 0 \\ 1 & 15 & 33 & 15 & 1 \\ 0 & 6 & 15 & 6 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}; \gamma H = \frac{1}{121}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -6 & -15 & -6 & 0 \\ -1 & -15 & 88 & -15 & -1 \\ 0 & -6 & -15 & -6 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}. \tag{2.6}$$

Для контрастування ЦЗ можна використовувати оператор

$$pK_{i,j} = K\left(p^{i,j}\right) = \sum_{ii=i-2}^{i+2}\sum_{jj=j-2}^{j+2} \gamma K_{ii-i,jj-j} p_{ii,jj}, i, j \in \mathbb{Z},$$

де

$\{pK_{i,j}\}_{i,j\in\mathbb{Z}}$ - кольорова складова відконтрастованого растру;

$$\gamma K = \frac{1}{16}\begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 3 & -13 & 3 & 0 \\ 2 & -13 & 48 & -13 & 2 \\ 0 & 3 & -13 & 3 & 0 \\ 0 & 0 & 2 & 0 & 0 \end{pmatrix}, \tag{2.7}$$

який, до того ж, є псевдозворотним оператором [72, 76–78] до оператора $L\left(p^{i,j}\right)$ низькочастотної фільтрації з маскою (2.5), тобто виконується приблизне співвідношення:

$$p_{i,j} \approx K\left(L\left(p^{i,j}\right)\right), i,j \in \mathbb{Z}. \tag{2.8}$$

Оператори $K\left(p^{i,j}\right)$, $L\left(p^{i,j}\right)$, які задовольняють співвідношенню (2.8), можуть мати застосування в задачах стиснення ЦЗ [74, 79], наприклад, доведено, що згладжене ЦЗ після стиснення поширеними методами з втратами має менший розмір. Якщо ж необхідно повернути початковий вигляд, то після «деархівації» до ЦЗ застосовують псевдозворотний оператор. Звісно, при такому підході мають місце деякі візуально непомітні втрати [79].

Стабілізація ЦЗ, спотвореного мікрорухом камери фіксації (підвищення різкості) [78] можлива за використанням такого оператора:

$$pR_{i,j} = R\left(p^{i,j}\right) = \sum_{ii=i-2}^{i+2}\sum_{jj=j-2}^{j+2} \gamma R_{ii-i,jj-j} p_{ii,jj}, i,j \in \mathbb{Z},$$

де

$\{pR_{i,j}\}_{i,j\in\mathbb{Z}}$ - кольорова складова растру з покращеною різкістю;

$$\gamma R = \frac{1}{42}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -8 & 1 & 0 \\ -1 & -8 & 74 & -8 & -1 \\ 0 & 1 & -8 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}, \tag{2.9}$$

або

$$\gamma R = \frac{1}{15}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 \\ -1 & -4 & 31 & -4 & -1 \\ 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}. \tag{2.10}$$

Зауважимо, що використання масок (2.9) та (2.10) забезпечує, відповідно, менший та більший ступінь підвищення різкості.

Для чотирикратного рекурентного поповнення кількості членів послідовності $\{p_{i,j,0}\}_{i,j\in\mathbb{Z}}$ (двократного рекурентного збільшення лінійних розмірів ЦЗ) [76, 79] достатньо на кожному κ-му ($\kappa = 1, 2, \ldots$) кроці рекурсії визначати члени нової послідовності кольорових складових растру $\{p_{2i,2j,\kappa}\}_{i,j\in\mathbb{Z}}$, на основі лінійних операторів, що засновані на даних попереднього кроку рекурсії:

$$p_{2i,2j,\kappa} = A\left(p^{\kappa-1,i,j}\right), p_{2i+1,2j,\kappa} = B\left(p^{\kappa-1,i,j}\right),$$

$$p_{2i,2j+1,\kappa} = C\left(p^{\kappa-1,i,j}\right), p_{2i+1,2j+1,\kappa} = D\left(p^{\kappa-1,i,j}\right),$$

$$i, j \in \mathbb{Z},$$

де $A\left(p^{\kappa-1,i,j}\right)$; $B\left(p^{\kappa-1,i,j}\right)$; $C\left(p^{\kappa-1,i,j}\right)$; $D\left(p^{\kappa-1,i,j}\right)$ можна задавати з умов масштабування. Так, якщо є потреба збільшувати зображення зі згладжуванням, то мають місце такі оператори:

$$p_{a,b,\kappa} = \sum_{ii=i-1}^{ii+1}\sum_{jj=j-1}^{j+1} \gamma\Lambda_{ii-i,jj-j}p_{ii,jj,\kappa-1},$$

де

$$\Lambda = \begin{bmatrix} A: a = 2i, b = 2j, \\ B: a = 2i+1, b = 2j, \\ C: a = 2i, b = 2j+1, \\ D: a = 2i+1, b = 2j+1; \end{bmatrix}, \tag{2.11}$$

$$\gamma A = \frac{1}{64}\begin{pmatrix} 1 & 6 & 1 \\ 6 & 36 & 6 \\ 1 & 6 & 1 \end{pmatrix}; \quad \gamma B = \frac{1}{16}\begin{pmatrix} 0 & 0 & 0 \\ 1 & 6 & 1 \\ 1 & 6 & 1 \end{pmatrix};$$

$$\gamma C = \frac{1}{16}\begin{pmatrix} 0 & 1 & 1 \\ 0 & 6 & 6 \\ 0 & 1 & 1 \end{pmatrix}; \quad \gamma D = \frac{1}{4}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{2.12}$$

Коли немає потреби збільшувати зі згладжуванням, то варто застосовувати такі оператори:

$$p_{a,b,\kappa} = \sum_{ii=i-2}^{ii+2} \sum_{jj=j-2}^{j+2} \gamma\Lambda_{ii-i,jj-j} p_{ii,jj,\kappa-1},$$

де Λ - визначається згідно (2.11);

$$\gamma A = \frac{1}{50}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ -1 & 2 & 46 & 2 & -1 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}; \quad \gamma B = \frac{1}{72}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -6 & 0 & 0 \\ -1 & 2 & 40 & 2 & -1 \\ -1 & 2 & 40 & 2 & -1 \\ 0 & 0 & -6 & 0 & 0 \end{pmatrix};$$

$$\gamma C = \frac{1}{72}\begin{pmatrix} 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 2 & 2 & 0 \\ 0 & -6 & 40 & 40 & -6 \\ 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & -1 & -1 & 0 \end{pmatrix}; \gamma D = \frac{1}{20}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 \\ 0 & -1 & 7 & 7 & -1 \\ 0 & -1 & 7 & 7 & -1 \\ 0 & 0 & -1 & -1 & 0 \end{pmatrix}. \tag{2.13}$$

Для рекурентного двократного збільшення масштабу (двократного зменшення горизонтального та вертикального розмірів) зображення необхідно на кожному κ-му ($\kappa = 1, 2, \ldots$) кроці рекурсії чотирикратно зменшувати кількість пікселів, звільнюючи у новому κ-му растрі місце під три пікселі: праворуч, зверху та зверху навскіс від кожного (i, j)-го пікселя $(\kappa - 1)$-го растру. Тобто, якщо $\{p_{i,j,\kappa}\}_{i,j\in\mathbb{Z}}$ — послідовність однієї із кольорових складових κ-го зменшеного растру, то

$$p_{i,j,\kappa} = p_{2i,2j,\kappa-1},$$

при цьому пам'ять під розміщення величин

$$p_{2i+1,2j,\kappa-1}, p_{2i,2j+1,\kappa-1}, p_{2i+1,2j+1,\kappa-1},$$

може бути вивільнена. Але, окрім тривіального визначення членів послідовності $\{p_{i,j,\kappa}\}_{i,j\in\mathbb{Z}}$ відповідно до (2.13), можна реалізувати збільшення масштабу зображення зі згладжуванням, контрастуванням, спрямованою фільтрацією тощо — залежно від конкретних потреб. У такому разі величини $p_{i,j,\kappa}$ визначаються на підставі деякого лінійного функціоналу

$$p_{i,j,\kappa} = \sum_{ii=2i-r}^{2i+r} \sum_{jj=2j-r}^{2j+r} \gamma_{ii-i,jj-j} p_{ii,jj,\kappa-1}, i, j \in \mathbb{Z}, r = 1, 2, \ldots, \tag{2.14}$$

що побудований на даних попереднього кроку рекурсії, а як γ можна використовувати маски (2.4–2.7) та (2.9) і (2.10).

2.3 Вплив на швидкодію ОЗ знакових та беззнакових обчислень

У контексті оптимізації алгоритмів обробки ЦЗ на архітектурі ARM64, особливо при реалізації ОЗ, беззнакові обчислення демонструють значно вищу ефективність порівняно з обчисленнями зі знаком. Розглянемо детально фактори, які обумовлюють цю перевагу, та їхній вплив на загальну швидкодію в межах дослідження.

2.3.1 Відсутність необхідності аналізу знаку

При роботі з беззнаковими числами відпадає потреба в додаткових операціях для визначення та обробки знаку операндів. Це особливо критично в контексті оптимізації з використанням NEON64, де кожна додаткова операція може значно вплинути на загальну швидкодію алгоритму [8, 22, 62]. Тож у випадку знакових чисел процесор має виконувати додаткові перевірки при кожній арифметичній операції, щоб правильно обробити знак результату. Це включає:

- Перевірку знаку операндів перед операцією.
- Визначення знаку результату після операції.

- Коригування результату відповідно до правил арифметики зі знаком.

Ці додаткові кроки, хоч і займають лише кілька тактів процесора кожен, у масштабах великих обсягів даних, характерних для ОЦЗ, сумарно призводять до значного зниження швидкодії.

2.3.2 Спрощення логіки обчислень

Для ЯЗ з беззнаковими елементами алгоритм може виконувати однорідні обчислення для всіх елементів без необхідності розгалуження логіки. Це дозволяє ефективно використовувати векторні операції NEON64 для одночасної обробки кількох елементів [8, 22].

У цьому дослідженні було оброблено одночасно три елементи маски та 32 елементи RGB ЦЗ, що значно прискорює процес згортки. Така однорідність обчислень максимально використовує потужність векторних інструкцій NEON64.

2.3.3 Уникнення умовних переходів

При роботі з беззнаковими числами відсутня необхідність у додаткових умовних переходах, які можуть порушувати роботу конвеєра процесора. Це особливо критично для сучасних ЦП ARM64 з глибоким конвеєром, де кожен незапланований перехід може призвести до втрат швидкодії. Розглянемо механізм роботи конвеєра процесора:

1. Вибірка інструкції.
2. Декодування інструкції.
3. Виконання.
4. Доступ до пам’яті (якщо потрібно).
5. Запис результату.

Умовний перехід може призвести до очищення конвеєра, якщо передбачення переходу виявилося неправильним. Це означає, що процесор має скасувати роботу, виконану на етапах 1-3, і почати заново з нової адреси. Такі ”промахи”можуть коштувати десятки тактів процесора. А у випадку з беззнаковими обчисленнями можна уникнути таких ситуацій, забезпечуючи більш передбачуваний потік виконання інструкцій.

2.3.4 Відмова від використання регістру стану ЦП для підвищення швидкодії СО:

Беззнакові обчислення дозволяють більш ефективно використовувати 128-бітні векторні регістри NEON64, оскільки не потрібно резервувати біти для зберігання інформації про знак та стан порівняння тощо [19, 80]. Це дозволяє обробляти більше даних за одну векторну операцію та уникнути додаткового аналізу [3].

У випадку 16-бітних обчислень, що використовуються у даному дослідженні, використовується на постійній основі 12 векторних регістрів для збереження попередньо обрахованих даних RGB ЦЗ. Тут треба пояснити, що ця група поділена на 3 частини (тобто 4 вектори груп (R|G|B)), яка зберігає попередньо оброблені частини ЦЗ. Таким чином, для кольору (R|G|B) використовується 4 векторних регістри. А з цього виходить, що для беззнакових операцій потрібно 6 операцій umlal та 6 операцій umlal2, що виконують множення поточного значення ЯЗ на групу значень з даного спектру (R|G|B) ЦЗ і виконують додавання до результату цієї відповідної групи.

Але у випадку з наявністю від'ємного значення у ЯЗ даний (майже ідеальний) підхід не працюватиме. Через це було проведено попереднє дослідження, котре виявило, що навіть за достатньої наявності векторних регістрів (їх у SIMD-розширенні NEON64 — 32 на кожне ядро ЦП) має бути виконано двічі попередньо описаний алгоритм, і більше того, має бути попередньо визначено 24 регістри та виконано щонайменше 6 разів umlal і 6 разів umlal2 «+» 6 разів umlsl та 6 разів umlsl2, і наостанок — операцію вибору bls, що вибирає біти з другого (Vn) або третього (Vm) вектора на основі значень бітів у першому векторі (Vd). Таким чином, ще «+» 12 операцій, і це не враховуючи 12 операцій mov або dup для того, щоб можна було задавати значення першому вектору (Vd) (через те, що mov або dup менш швидкі, їх не було взято до уваги, але вони використовувалися). Простіше кажучи, у поточному дослідженні не було використано такого екзотичного підходу, а був, як це не дивно, використаний підхід з порівнянням, але без використання регістру стану.

2.3.5 Спрощення інструкцій SIMD:

Інструкції NEON для беззнакових операцій часто простіші та виконуються швидше, ніж їх аналоги для знакових чисел. Наприклад, беззнакове множення може бути реалізоване ефективніше, ніж знакове.

Розглянемо різницю між знаковим (SMULL) та беззнаковим (UMULL) множенням:

- SMULL/SMLSL повинен правильно обробляти знаки обох операндів та визначити знак результату. Це включає додаткову логіку всередині АЛП і має додаткові накладні навантаження на обрахунок результатів.
- UMULL/UMLAL/UMLSL просто виконує множення бітових представлень чисел, що є простішою операцією на рівні логічних вентилів без додаткових накладних навантажень.

Ця різниця може здаватися незначною для одиничної операції, але при виконанні мільйонів таких операцій під час обробки ЦЗ, сумарний виграш у швидкодії стає суттєвим. У випадку виконання множення з відніманням (UMLSL) та додаванням (UMLAL), слід зазначити, що зокрема вони є «виконавчим ядром» ЯІТ [18].

Для спрощення сприйняття цілочисельні значення можуть приймати такі значення, залежно від того, чи вони від'ємні чи додатні:

$$min(u8/u16/u32) = 0 \qquad \begin{aligned} &max\,(u8) = 0xFF \\ &max(s8) = max(u8) \gg 1 = 0x7F \\ &min(s8) = -(max(s8) + 1) = -0x80 \end{aligned}$$

$$\begin{aligned} &max\,(u16) = 0xFFFF \\ &max(s16) = max(u16) \gg 1 = 0x7FFF \\ &min(s16) = -(max(s16) + 1) = -0x8000 \end{aligned} \tag{2.15}$$

$$\begin{aligned} &max\,(u32) = 0xFFFFFFFF \\ &max(s32) = max(u32) \gg 1 = 0x7FFFFFFF \\ &min(s32) = -(max(s32) + 1) = -0x80000000 \end{aligned}$$

Спираючись на значення максимумів і мінімумів стандартних цілих типів (2.15), можна отримати уявлення про максимальне та мінімальне значення, що

буде сформоване у кожній комірці векторного регістра NEON64. Тож за умови 16-бітних знакових обчислень діапазон значень, що зберігатиметься у комірках векторного регістра, буде [-0x8000; 0x7FFF] — за наявності в ЯЗ від'ємних значень. Як було зазначено, для того щоб обчислення були проведені правильно, буде задіяно додаткову логіку всередині АЛП, що має призвести (і на практиці призводить) до додаткових накладних витрат на обчислення результатів. Саме тому у подальших розділах буде окремо розглянуто варіант кластеризації ЯЗ за критерієм «0»/«не 0» та за критерієм, чи від'ємне значення в елементі ЯЗ, чи додатне. Це призведе до одноманітності та передбачуваності варіації операції множення (чи це буде операція UMLSL, чи UMLAL) і, у свою чергу, до підвищення швидкодії [22].

2.3.6 Зменшення кількості інструкцій:

У випадку з беззнаковими обчисленнями, алгоритм ОЗ може бути реалізований з використанням меншої кількості інструкцій. Це не тільки прискорює виконання, але й зменшує розмір коду, що позитивно впливає на використання кеш-пам'яті інструкцій.

Менша кількість інструкцій означає:

- Зменшення кількості вибірок інструкцій з пам'яті
- Менше навантаження на декодер інструкцій
- Потенційно кращу ефективність кеш-пам'яті інструкцій

Таким чином, це в сукупності призводить до більш ефективного використання ресурсів процесора.

На противагу цьому, при роботі з ЯЗ, що містять від'ємні елементи, виникає необхідність у додаткових операціях та умовних переходах. Розглянемо детально ці додаткові кроки та їх вплив на швидкодію:

1. Аналіз знаку елементів ЯЗ:
   cmlt v0.16b, v0.16b, #0
   Ця інструкція порівняння виконує наступні дії:
   (а) Завантажує 16 байт елементів ЯЗ у векторний регістр v0.
   (б) Порівнює кожен з 16 байт v0 з нулем.
   (в) Записує результат порівняння як 0xff/0x00 для кожного елемента v0.
   Хоча ця операція виконується векторно, і вона набагато швидша за використання порівняння cmp, вона все одно додає накладні витрати, яких немає

при роботі з беззнаковими числами.

2. Передача результатів порівняння:
   umov w0, v0.b[0]
   Ця інструкція виконує наступне:
   (а) Вибирає перший байт з векторного регістра.
   (б) Переміщує його в скалярне 32-бітне представлення X0 регістра загального призначення.
   Єдиним недоліком є ця операція, що створює ”вузьке місце”у потоці даних, оскільки ми переходимо від широких векторних операцій до вузьких скалярних.
3. Умовні переходи. Кожен з умовних переходів потенційно може призвести до порушення роботи конвеєра процесора. Якщо передбачення переходу виявиться неправильним, процесор буде змушений очистити конвеєр та почати виконання з правильної адреси, що призведе до втрати кількох десятків тактів.
   Для вирішення проблеми використання інструкції CMP, яка впливає на регістр станів, була розроблена стратегія з використанням команди CBNZ. Хоча це і є оптимізацією порівняно з прямим використанням CMP, все ж таки це створює додаткові накладні витрати порівняно з беззнаковими обчисленнями, але тим не менш у даному випадку не йде використання регістру стану та його аналізу бітів стану, лише чи значення X0 не дорівнює нулю, і якщо ні, то робиться перехід до локальної мітки вперед (f – свідчить саме про то). Тут слід зауважити, що CBNZ не порівнює значення з нулем у сенсі ”більше чи менше воно просто перевіряє на нерівність нулю. І якщо Для 64-бітного регістра X0, будь-яке значення, де хоча б один біт (з 64х) встановлений, це призведе до переходу на мітку [81].
   Тож підхід/алгоритм доволі простий:
   cbnz X0, 1f /// якщо X0$>$0, то елемент ЯЗ $<$ 0 і йде перехід
   /// векторні обчислення для позитивного елемента маски
   b 2f /// перехід до кроків підготовки наступної ітерації
   1:
   /// векторні обчислення для негативного елемента маски
   2:
   /// кроки для підготовки наступного кроку обчислень

Отже, інструкція CBNZ не змінює регістр станів, що є корисним для уникнення побічних ефектів використання регістру стану ЦП. Однак вона все ще вносить елемент непередбачуваності у потік виконання інструкцій, що може дещо негативно вплинути на ефективність роботи конвеєра та предиктора переходів. Втім, такий підхід є ефективнішим за просте використання порівняння, що призводить до звернення до регістру стану ЦП.

Висновок: беззнакові обчислення демонструють вищу ефективність у контексті ОЗ на ARM64 архітектурі завдяки комплексу факторів: 1) Спрощення логіки обчислень; 2) Зменшення кількості інструкцій; 3) Ефективніше використання SIMD можливостей; 4) Уникнення умовних переходів; 5) Оптимальне використання регістрів та кеш-пам’яті.

Ці фактори в сукупності призводять до значного підвищення швидкодії алгоритму ОЗ, особливо при обробці великих обсягів даних, що є типовим для задач комп’ютерного зору та обробки ЦЗ.

Важливо відзначити, що хоча беззнакові обчислення і демонструють кращу швидкодію, вони можуть бути не підходящими для всіх типів задач обробки ЦЗ. У випадках, де необхідно працювати з від’ємними значеннями або де діапазон значень виходить за межі беззнакового представлення, все ще може бути необхідним використання знакових обчислень. Тому вибір між знаковими та беззнаковими обчисленнями повинен базуватися на конкретних вимогах задачі та характеристиках вхідних даних.

У контексті нашого дослідження, перехід до беззнакових обчислень дозволив значно підвищити швидкодію алгоритму ОЗ, що відкриває нові можливості для реалізації складних алгоритмів комп’ютерного зору на пристроях з обмеженими обчислювальними ресурсами.

2.4 Адаптація МЗ для беззнакових обчислень з компенсаційною нормалізацією результатів ОЗ

Якщо прийняти, що всі елементи $\gamma_{ii-i,jj-j}$ в операторі (2.1) додатні, то умова (2.2) може бути переформульована так:

$$\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j} \leq 255 \tag{2.16}$$

Тоді для масок (2.4, 2.5, 2.10, 2.13), що містять і додатні, і від'ємні елементи, можна запропонувати наступні перетворення, що забезпечать беззнакові 16-бітні обчислення з виконанням умови (2.16).

Нехай $\gamma_{\min}$ – найбільший за модулем від'ємний елемент маски фільтра:

$$\gamma_{\min} = \max_{i,j} \{|\gamma_{i,j}|\}.$$

Позначимо

$$\gamma_{bias} = \gamma_{\min} + 1. \tag{2.17}$$

Тоді, для отримання нових масок з додатніми ненульовими коефіцієнтами, піддамо елементи масок (2.4, 2.5, 2.10, 2.13) такому перетворенню:

$$\gamma_{i,j}^{new} = \gamma_{i,j} + \gamma_{bias}, \gamma_{i,j}^{new} > 0 \tag{2.18}$$

Після виконання згортки з модифікованими за (2.18) масками

$$F^{new}\left(p^{i,j}\right) = \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{new} p_{ii,jj},$$

маємо забезпечити досягнення ефекту фільтра, що визначено його властивостями до перетворення. Тому для отримання початкового оператора $F\left(p^{i,j}\right)$, як до модифікації, потрібно від згортки з додатніми коефіієнтами маски відняти «згортку» зсуву маски:

$$F\left(p^{i,j}\right)=\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}\gamma^{new}_{ii-i,jj-j}p_{ii,jj}-\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}\gamma_{bias}p_{ii,jj}=$$
$$=\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}\left(\gamma_{ii-i,jj-j}+\gamma_{bias}\right)p_{ii,jj}-\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}\gamma_{bias}p_{ii,jj}= \tag{2.19}$$
$$=F^{new}\left(p^{i,j}\right)-\gamma_{bias}\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}p_{ii,jj}$$

Виходячи з (2.18) та враховуючи (2.16) нескладно отримати нові маски фільтрів для беззнакових 16-бітних обчислень [8, 22, 62]. Зокрема, для високочастотної фільтрації при субполосній фільтрації можна скористатись масками (зсунені маски (2.4) та (2.5)):

$$\gamma L=\frac{1}{64}\begin{pmatrix}1&6&1\\6&36&6\\1&6&1\end{pmatrix};\quad \gamma H_{bias}=7,\gamma H^{new}=\frac{1}{64}\begin{pmatrix}6&1&6\\1&35&1\\6&1&6\end{pmatrix} \tag{2.20}$$

$$\gamma L=\frac{1}{36}\begin{pmatrix}1&4&1\\4&16&4\\1&4&1\end{pmatrix};\quad \gamma H_{bias}=5,\gamma H^{new}=\frac{1}{36}\begin{pmatrix}4&1&4\\1&25&1\\4&1&4\end{pmatrix}. \tag{2.21}$$

Для підвищення різкості зображення, як аналог маски (2.10), використаємо таку маску:

$$\gamma_{bias}=5,\gamma R^{new}=\frac{1}{15}\begin{pmatrix}0&0&4&0&0\\0&6&1&6&0\\4&1&36&1&4\\0&6&1&6&0\\0&0&4&0&0\end{pmatrix}. \tag{2.22}$$

Для забезпечення можливості двократного рекурентного збільшення лінійних розмірів ЦЗ, як аналоги масок (2.13), при беззнакових розрахунках про-

понується використовувати такі маски:

$$\gamma A^{new} = \frac{1}{50}\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 1 & 4 & 48 & 4 & 1 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \quad \gamma B^{new} = \frac{1}{72}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 6 & 9 & 40 & 9 & 6 \\ 6 & 9 & 40 & 9 & 6 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix},$$

$$\gamma A_{bias} = 2, \quad \gamma B_{bias} = 7, \tag{2.23}$$

$$\gamma C^{new} = \frac{1}{72}\begin{pmatrix} 0 & 0 & 6 & 6 & 0 \\ 0 & 0 & 9 & 9 & 0 \\ 0 & 1 & 47 & 47 & 1 \\ 0 & 0 & 9 & 9 & 0 \\ 0 & 0 & 6 & 6 & 0 \end{pmatrix}, \quad \gamma D^{new} = \frac{1}{20}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 9 & 9 & 1 \\ 0 & 1 & 9 & 9 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix},$$

$$\gamma C_{bias} = 7, \quad \gamma D_{bias} = 2.$$

Нескладно переконатися, що всі наведені маски (2.20–2.23) задовольняють умові (2.16), тож можуть бути рекомендовані для організації 16-бітних беззнакових обчислень [82, 83].

Позитивним, при використанні даної методики, є те, що виконання ОЗ відбувається у беззнаковому діапазоні значень ЯЗ, що, у теорії, має призвести до підвищення швидкодії. Негативним є те, що необхідно робити обчислення ОЗ та бокс-фільтра водночас із подальшим відніманням другого ($\gamma_{bias} \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} p_{ii,jj}$) від першого $F^{new}\left(p^{i,j}\right)$ (2.19).

Прототип для підтвердження у вигляді програми мовою С цієї модифікації/оптимізації ОЗ ЦЗ було реалізовано та перевірено, з отриманням значення абсолютної похибки в межах норми (абсолютна похибка = 1 — для пікселя 8-bit RGB ЦЗ). Саме тому поточний варіант було розглянуто як перспективний для подальших досліджень (дод. А.8).

2.5 Оптимізація обчислень, де необхідно перевіряти потрапляння в діапазон від 0 до 255

Як було вже описано у (2.2) та (2.3), у цьому дослідженні спираємося на факт потрапляння результату обчислення скалярного добутку в діапазон [0; 0xFFFF] (або $\left[0; 2^{16}\right]$) унікальних значень. Якщо припускати, що значення можуть бути від'ємними, то — у діапазон [-0x8000; 0x7FFF] (або [-32768; 32767]). Щодо значень $\gamma_{ii-i,jj-j}$, то діапазон має відповідати (2.16) у випадку беззнакових елементів ЯЗ або (2.3) — у випадку наявності хоча б одного від'ємного елемента у ЯЗ.

При виході значень елементів ЯЗ $\gamma_{ii-i,jj-j}$ за межі означених значень треба використовувати інший алгоритм обчислення ОЗ ЦЗ.

Нехай маємо ядро згортки $\gamma_{ii-i,jj-j}$ з різнорідними елементами у діапазоні значень від 0 до $2^{32}-1$, тоді:

$$\gamma = \begin{pmatrix} 0x00FEDCBA & 0x00000000 & 0x00FEDCBA \\ 0x00000000 & 0xFEDCBA98 & 0x00000000 \\ 0x00FEDCBA & 0x00000000 & 0x00FEDCBA \end{pmatrix} \tag{2.24}$$

така маска вже не буде відповідати умові виразу (2.2) і на перший погляд її не можливо використати у 16-бітних обчисленнях. Але якщо розбити елементи МЗ (2.24) на байти тоді вийде 4 маски з 32-бітної маски котрі будуть мати у своїх елементах байтові компоненти базової ЯЗ:

$$\gamma^{(0)} = \begin{pmatrix} 0xBA & 0x00 & 0xBA \\ 0x00 & 0x98 & 0x00 \\ 0xBA & 0x00 & 0xBA \end{pmatrix}, \quad \gamma^{(1)} = \begin{pmatrix} 0xDC & 0x00 & 0xDC \\ 0x00 & 0xBA & 0x00 \\ 0xDC & 0x00 & 0xDC \end{pmatrix},$$

$$\gamma^{(2)} = \begin{pmatrix} 0xFE & 0x00 & 0xFE \\ 0x00 & 0xDC & 0x00 \\ 0xFE & 0x00 & 0xFE \end{pmatrix}, \quad \gamma^{(3)} = \begin{pmatrix} 0x00 & 0x00 & 0x00 \\ 0x00 & 0xFE & 0x00 \\ 0x00 & 0x00 & 0x00 \end{pmatrix}. \tag{2.25}$$

Щоб отримати ЯЗ (2.24) з ЯЗ (2.25) треба виконати наступну операцію:

$$\gamma_{i,j} = \gamma_{i,j}^{(0)} + \gamma_{i,j}^{(1)} \cdot 2^8 + \gamma_{i,j}^{(2)} \cdot 2^{16} + \gamma_{i,j}^{(3)} \cdot 2^{24}. \tag{2.26}$$

Кожен елемент ЯЗ (2.25) було отримано (2.24) таким чином:

1. $\gamma_{i,j}^{(0)} = \gamma_{i,j}\ mod\ 2^8$ - найменш значущий байт;
2. $\gamma_{i,j}^{(1)} = \left(\frac{\gamma_{i,j}}{2^8}\right) mod\ 2^8$ - другий байт;
3. $\gamma_{i,j}^{(2)} = \left(\frac{\gamma_{i,j}}{2^{16}}\right) mod\ 2^8$ - третій байт;
4. $\gamma_{i,j}^{(3)} = \left(\frac{\gamma_{i,j}}{2^{24}}\right)$ - найбільш значущий байт.

Тоді алгоритм згортки ЯЗ з ЦЗ (2.1) можна переписати, використавши отримані ЯЗ (2.25) наступним чином:

$$\begin{aligned} F\left(p^{i,j}\right) = & \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(0)} p_{ii,jj} + \\ & 2^8 \cdot \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(1)} p_{ii,jj} + \\ & 2^{16} \cdot \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(2)} p_{ii,jj} + \\ & 2^{24} \cdot \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(3)} p_{ii,jj}. \end{aligned} \tag{2.27}$$

Але, як видно з виразу (2.27), можна все записати у формі, що задіює програмну оптимізацію зсуву значень вліво:

$$\begin{aligned} F\left(p^{i,j}\right) = & \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(0)} p_{ii,jj} + \\ & \left(\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(1)} p_{ii,jj} \ll 8\right) + \\ & \left(\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(2)} p_{ii,jj} \ll 16\right) + \\ & \left(\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}^{(3)} p_{ii,jj} \ll 24\right). \end{aligned} \tag{2.28}$$

Таким чином все заводиться до операції множення та зсуву «вліво», що теоретично призведе до підвищення швидкодії [16].

2.6 Висновки

У другому розділі дисертаційної роботи було проведено дослідження лінійних операторів згортки для обробки ЦЗ, адаптованих для використання 16-бітних цілочисельних обчислень, що є ключовою передумовою для розробки високошвидкодіючої ОЗ на архітектурі ARM64 ЦП. На основі проведеного аналізу та розробок зроблено такі висновки:

1. Сформульовано умови відповідності ЯЗ для 16-бітних цілочисельних обчислень. Визначено, що для коректної роботи без переповнень на проміжних етапах коефіцієнти ЯЗ мають належати до діапазону 8-бітних цілих (uint8_t), а сумарний результат ОЗ перед нормалізацією не повинен виходити за межі діапазону 16-бітних знакових цілих (int16_t, від -32768 до 32767). Це обмеження є визначальним при проектуванні ЯЗ для 16-бітної арифметики.
2. Розроблено та адаптовано набір лінійних операторів на основі ОЗ для 16-бітних обчислень, що включає оператори для низькочастотної фільтрації (згладжування, зокрема на базі B-сплайнів другого та третього порядку), контрастування, цифрової стабілізації ЦЗ (підвищення різкості), а також оператори для рекурентного двократного збільшення та зменшення масштабу ЦЗ. Це демонструє практичну застосовність 16-бітних обчислень для широкого спектра задач обробки ЦЗ.
3. Проведено детальний аналіз впливу використання знакових та беззнакових 16-бітних цілочисельних обчислень на потенційну швидкодію ОЗ на архітектурі ЦП ARM64. Встановлено, що беззнакові обчислення мають суттєві переваги, зокрема: відсутність необхідності аналізу знаку операндів, спрощення логіки обчислень, уникнення умовних переходів, що можуть спричиняти зриви конвеєра, ефективніше використання векторних регістрів NEON64 (оскільки не резервуються біти для знаку на проміжних етапах множення-акумуляції), а також можливість використання простіших та потенційно швидших SIMD-інструкцій (наприклад, UMULL/UMLAL проти SMULL/SMLAL).

4. Запропоновано методику адаптації ЯЗ, що містять від'ємні коефіцієнти, для їх використання в рамках беззнакових 16-бітних обчислень. Метод полягає у зсуві всіх елементів ЯЗ на величину найбільшого за модулем від'ємного елемента для отримання ЯЗ з виключно додатними коефіцієнтами. Для компенсації цього зсуву та отримання коректного результату вихідної ОЗ необхідно від фінального результату відняти результат згортки вхідного зображення з ЯЗ, що складається з константних значень цього зсуву (фактично, операція бокс-фільтрації з відповідним коефіцієнтом). Це дозволяє виконувати основні обчислення в швидшому беззнаковому діапазоні.
5. Розглянуто та запропоновано підходи до оптимізації обчислень на етапі конвертації проміжних 16-бітних результатів ОЗ у фінальні 8-бітні значення пікселів, зокрема, для забезпечення коректного потрапляння значень у діапазон [0, 255]. Це включає використання операцій насичення (saturating arithmetic), які ефективно реалізовані в NEON64. Також запропоновано підхід до обробки ЯЗ, елементи яких виходять за межі 8-бітного діапазону, шляхом їх розбиття на кілька 8-бітних компонент та виконання кількох ОЗ з подальшим об'єднанням результатів, що дозволяє зберегти переваги 16-бітних проміжних обчислень.

Таким чином, у другому розділі закладено теоретичне та методичне підґрунтя для розробки ОЗ з підвищеною швидкодією на базі 16-бітних цілочисельних обчислень, обґрунтовано вибір беззнакової арифметики для NEON64 оптимізації та запропоновано шляхи адаптації наявних лінійних операторів до цих умов.

РОЗДІЛ 3. ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОБРОБКИ ЦИФРОВИХ ЗОБРАЖЕНЬ ІЗ ЗАСТОСУВАННЯМ ВЕКТОРИЗОВАНОЇ ОПЕРАЦІЇ ЗГОРТКИ НА ПЛАТФОРМІ ARM64

У цьому розділі представлено розроблену ІТ обробки ЦЗ, що базується на ВОЗ, оптимізованій/реалізованій для платформи ARM64 із використанням 16-бітних цілочисельних обчислень NEON64. Особливу увагу приділено формалізованій нотації як універсальному науковому методу опису векторизованих обчислень, та обґрунтовано її розширені можливості застосування, теоретичну значущість і внесок у методологію досліджень (дод. А.1-А.10 та в підрозд. 3.1). Розкрито концепцію ІТ, що спрямована на підвищення швидкодії обробки ЦЗ на вбудованих системах (підрозд. 3.2). Детально описано архітектуру розробленої ІТ, включаючи структуру низькорівневої С-бібліотеки (libconv_lib.so) та високорівневого Python-застосунку, а також механізми їхньої взаємодії (підрозд. 3.3, 3.4). Представлено програмну реалізацію ядра інформаційної технології (ЯІТ), зокрема інтерфейс та особливості функції conv_RGB_NEON(...), компіляцію для цільових платформ та попереднє порівняння швидкодії з аналогами (підрозд. 3.4). Розглянуто методику застосування ІТ для обробки різних типів ЦЗ, включаючи статичні зображення, відеопотоки, операції масштабування й використання різних типів ядер згортки (ЯЗ) (підрозд. 3.5). Описано розробку програмного інтерфейсу для ІТ, що включає Python-обгортку, інтеграцію з NumPy та реалізацію CLI-інтерфейсу (підрозд. 3.6). Наведено приклади комплексного застосування ІТ для типових задач обробки ЦЗ, а також сформульовано обмеження та рекомендації щодо її використання (підрозд. 3.7). Узагальнюючі висновки до розділу наведено в підрозділі 3.8.

3.1 Формалізована нотація як універсальний науковий метод опису векторизованих обчислень.

Розроблена формалізована нотація для опису векторизованих обчислень виходить за межі конкретної реалізації ШОЗ на платформі ARM64 і є універсальним науковим методом формального опису SIMD-операцій (дод. А.1-А.10). На відміну від наявних підходів, які фокусуються на описі конкретних інструкцій для специфічних архітектур, запропонована нотація забезпечує архітектурно-незалежний спосіб формалізації векторних обчислень, що робить її цінним

інструментом для широкого кола задач паралельних обчислень [7, 67]. Систематизоване відображення цієї нотації як пункту наукової новизни (включно з порівнянням з аналогами, відмінностями від відомих підходів та локалізацією доказової бази у тексті дисертації) наведено у Дод. Б, П.2 (section Б.5).

3.1.1 Загальний огляд формалізованої нотації.

Формалізована нотація складається з:

1. формалізації запису векторних регістрів NEON64 представлених у вигляді математичної нотації (A.12–A.17), що дозволяє формалізувати іменування векторних регістрів NEON64, тип операції (sign/unsigned) і відповідно тип даних (u8, s8, u16, s16, u32, s32, f32) та діапазон індексів елементів векторних регістрів (дод. A.3);
2. набору операцій над векторними регістрами:
    - дод. A.4.1: обнуління векторів (A.18);
    - дод. A.4.2: завантаження даних у вектори (A.22). Це загальна операція завантаження даних з ЦЗ (дод. A.4.2, A.4.3) та завантаження даних з ЯЗ (дод. A.4.4). Також слід зауважити про важливість збереження результатів обчислення що зберігаються у векторах, що описано процедурою (дод. A.5.5) - (A.44);
    - дод. A.4.5: обчислення векторизованого скалярного добутку, за допомогою операцій множення та додавання та множення та віднімання (A.29–A.31);
    - дод. A.4.6: циклічний зсув значень векторів (cshl), що уособлюють RGB ЦЗ, для забезпечення роботи з наступним елементом ЯЗ та ЦЗ без надлишкових завантажень даних у вектори з ЦЗ (A.32–A.34);
    - дод. A.4.7: завершення обчислення значень векторизованого скалярного добутку та підготовка до отримання кінцевого результату (A.35);
    - тощо.

Загалом комбіноване використання даних операцій дуже детально представлено блок-схемою (рис. A.4), котра ілюструє послідовність/взаємодію різних операцій над векторними регістрами для досягнення тільки однієї мети - обчислення ВСД, що є ядром ШОЗ.

Таблиця 3.1 — Зіставлення запропонованої нотації із наявними засобами опису SIMD-обчислень

Засіб опису	**Рівень / характер**	**Принципова відмінність від запропонованої нотації**
ARM A-profile Architecture Reference Manual	Формально-асемблерний (псевдокод окремих мнемонік: XOR, LD3, SMLAL тощо)	Описує *окремі інструкції*, а не алгоритми; немає узагальнень для багатоінструкційних обчислень; немає параметризації типу даних
ARM NEON Intrinsics	C-API виду `vmlaq_s16(…)`, `vld3q_u8(…)`	Програмна, а не математична абстракція; окрема функція для кожного типу даних; не пристосовано до публікаційного викладення
OpenCV Universal Intrinsics	Кросплатформений C++-шар: `v_uint8x16`, `v_load`, `v_mul`, `v_pack`	Найближче за духом, проте імперативне і прив'язане до C++; не дозволяє формального аналізу складності та коректності
Arm Compute Library (ACL)	C++-ядра задач комп'ютерного зору	Реалізаційний рівень; немає нотації як такої; не виокремлює узагальнених операторів
Підручники з ARM64 ASM та системного програмування	Інструктивний / навчальний	Інструкції розглядаються по одній; немає узагальненої формалізації багатоінструкційних алгоритмів
Наукова література з SIMD-парадигми (таксономія Флінна тощо)	Таксономічно-абстрактний	Класифікує паралелізм даних на високому рівні; не формалізує конкретних операцій векторних регістрів
Класичні описи згорткових алгоритмів у наукових публікаціях	Псевдокод або математичні вирази у класичній (скалярній) формі	Не формалізують векторні операції — оперують скалярами або абстрактними тензорами; не охоплюють специфіки SIMD-регістрів
Запропонована нотація	**Математичний (множинно-теоретичний), архітектурно-незалежний**	**Параметризація $dtype$ як першокласного об'єкта; узагальнені оператори ($CShL$, $maxmin_vdt$); експліцитні діапазони індексів ($\mathcal{I}^{dtype}_{start,end}$, i_1hlf, i_2hlf); самодокументоване іменування (V_R0007^{u16}); потенціал автогенерації коду**

3.1.2 Зіставлення із наявними засобами опису SIMD-обчислень.

Запропонована формалізована нотація позиціонується серед уже відомих засобів опису SIMD-обчислень як проміжний *архітектурно-незалежний математичний* рівень — між офіційною документацією архітектур та програмними API (intrinsics). У (табл. 3.1) наведено стисле зіставлення нотації з основними групами наявних засобів опису. Розгорнутий перелік аналогів із бібліографічними посиланнями та докладнішим аналізом наведено у Дод. Б, П.2 (підрозд. Б.5.2).

З наведеного зіставлення видно, що жоден з відомих засобів не одночасно поєднує математичну формальність, архітектурну незалежність та орієнтованість саме на алгоритмічний (а не інструктивний) рівень опису. Саме цю прогалину заповнює запропонована нотація. Бібліографічні посилання на конкретні джерела за кожною з груп аналогів, а також деталізовані відмінності — наведено у Дод. Б, П.2 (підрозд. Б.5.2–Б.5.3).

3.1.3 Застосування нотації для опису обчислювального ядра ШОЗ.

Для ефективного обрахунку ШОЗ, було запропоновано алгоритмізацію двох варіацій ВСД для ЯЗ, що представлено у вигляді блок-схем (рис. А.4) та (рис. А.6). Блок-схема (рис. А.4) представляє обрахунок ВСД, де кожен елемент ЯЗ буде використано для обрахування ШОЗ (дод. А.4.7), а блок-схема (рис. А.6) представляє оптимізований алгоритм обрахунку ВСД для розріджених ЯЗ, де елементи ЯЗ кластеризуються за критерієм як знаку так і того що значення елементу != 0 (дод. А.7.1). Так чи інакше, кожен з цих методів адаптовано для обчислення ВСД, в основі якого може бути ЯЗ, у якого кількість елементів більша ніж 16. Відповідно, найадаптивнішим є метод обчислення ВСД представлений (рис. А.6), через його гнучкість використання *index_map* (дод. А.7.1).

Після завершення обрахування ВСД йде процедура нормалізації ВСД (у котру включено процедури конвертування типів тощо) та збереження обчислення (А.44) ШОЗ для 32-х елементів ЦЗ, що представлено у вигляді (А.36–А.39) та (А.41–А.43). Більш детально це описано в (підрозд. А.5).

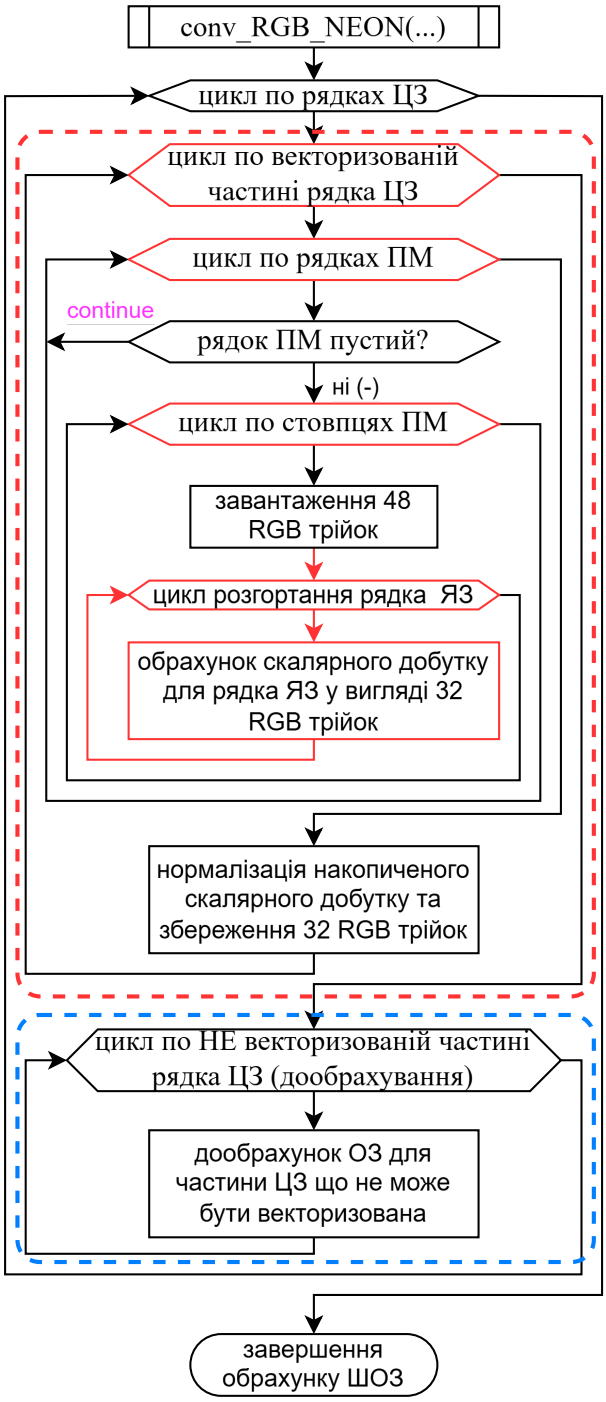


Рисунок 3.1 — Загальна блок-схема обчислення алгоритму ШОЗ

Процес інтеграції нормалізації ВСД у ШОЗ представлено у двох поданнях: узагальненому (рис. 3.1) та деталізованому (рис. А.5).

3.1.4 Розширені можливості застосування.

Запропонована нотація може бути використана для: формального опису алгоритмів обробки даних на різних SIMD-архітектурах (x86 SSE4/AVX2, RISC-V тощо); автоматичної генерації оптимізованого коду для різних векторних

розширень; формального доведення коректності векторизованих алгоритмів; аналізу та оптимізації алгоритмів з точки зору ефективності використання векторних інструкцій; створення абстрактних моделей для опису SIMD обчислень у навчальних та дослідницьких цілях.

3.1.5 Теоретична значущість і методологічний внесок.

Кількісно запропонована нотація (дод. А.1–А.10) охоплює:

– **7 базових типів даних** (u8, s8, u16, s16, u32, s32, f32) у єдиному параметризованому записі $VC_kI_e^{dtype}$ (А.12) — замість 7 окремих сімейств імен інтрінсиків (`vmla_u8`, `vmla_s16`, `vmla_f32` тощо);

– **6 класів базових векторних операцій** (обнуління, завантаження `ld1`/`ld3`, множення-акумуляція `smlal`/`smlsl`, циклічний зсув, конвертація типів, збереження `st3`) — (А.18, А.22, А.29, А.32, А.40, А.44);

– **2 узагальнені оператори** ($CShL$ (А.32) та $maxmin_vdt$ (А.40)), які інкапсулюють родини ARM64 NEON64 інструкцій (ext-варіанти; SQXTN/UQXTN/SQXTN2/UQXTN2);

– **3 блок-схеми** обчислення ШОЗ та ВСД (рис. А.4–А.6) повністю записано в термінах нотації, без використання асемблерних мнемонік — це практично підтверджує її повноту для розглянутого класу алгоритмів.

З теоретичної точки зору, нотація заповнює прогалину між математичним описом алгоритмів та їх практичною реалізацією через інтрінсики або Inline Assembly, надаючи проміжний абстрактний рівень, придатний для формального аналізу складності, доведення коректності та публікаційного викладення векторизованих алгоритмів — що має значення для теорії паралельних обчислень [84–86].

Як методологічний інструмент для подальших досліджень у галузі паралельних обчислень, нотація відкриває можливості: систематичного зіставлення стратегій векторизації; формалізації та класифікації шаблонів векторизації для типових алгоритмів; розробки формальних методів верифікації векторизованих алгоритмів; створення теоретичних моделей для прогнозування швидкодії векторизованих обчислень [87, 88].

3.2 Концепція інформаційної технології для високошвидкодійної обробки ЦЗ

Сучасний розвиток цифрових технологій і поширення SBC на базі архітектури ARM64 створюють нові можливості для обробки ЦЗ (ОЦЗ) на вбудованих системах. Однак наявні програмні рішення для ОЦЗ не завжди враховують особливості архітектури ARM64, зокрема наявність технології NEON64 [84].

Запропонована ІТ спрямована на вирішення цієї проблеми шляхом розробки спеціалізованого підходу до реалізації ОЗ ЦЗ з використанням векторизованих обчислень на архітектурі ЦП ARM64. Основною ідеєю ІТ є оптимізація найбільш ресурсомісткої операції — ОЗ, через її векторизацію, використовуючи NEON64 як інструмент паралелізації на рівні даних однією командою. Розроблена ІТ базується на трьох ключових принципах:

1. Векторизація обчислювального ЯІТ для ARM64. Ядром ІТ є ВОЗ, оптимізована/реалізована з використанням NEON64. Детальний математичний опис ШОЗ та набір спеціалізованих 8-бітних ЯЗ представлено у підрозд. 2.2, а опис її векторизації/оптимізації — у підрозд. А.3–А.7. Ключовою особливістю ЯІТ є використання 16-бітних цілочисельних обчислень, що дозволяє досягти оптимального співвідношення між точністю результату обробки ЦЗ та швидкодією його обробки [84, 89];

2. Модульність та інтероперабельність. ІТ побудована за модульним принципом, де обчислювальне ядро реалізовано у вигляді окремої бібліотеки (libconv_lib.so яка може використовуватися різними програмними компонентами. Для забезпечення інтероперабельності розроблено відповідні інтерфейси для взаємодії з високорівневими мовами програмування, зокрема Python;

3. Уніфікація опису векторизованих обчислень. Важливим компонентом ІТ є розроблена формалізована нотація для опису векторизованих обчислень/операцій з використанням NEON64, що представлена в (дод. А.3). На відміну від наявних підходів, ця нотація забезпечує чіткий і стандартизований спосіб опису векторних/SIMD операцій, що значно спрощує розробку та оптимізацію алгоритмів для ARM64 архітектури.

Запропонована формалізована нотація є важливою, оскільки подібних стандартизованих підходів у науковій літературі практично не існує. Більшість досліджень в області векторизованих обчислень фокусуються на конкретних

алгоритмах без формалізації процесу опису самих векторних операцій. Розроблена нотація дозволяє абстрагуватися від конкретних асемблерних інструкцій SIMD, представляючи операції у вигляді математичних виразів, що робить їх більш зрозумілими для дослідників та розробників.

Загальна схема ІТ показана на (див. рис. 3.2), де представлено взаємозв'язок між основними компонентами ІТ та їх функціональне призначення. Як видно з (див. рис. 3.2), центральним елементом ІТ є функція conv_RGB_NEON(…), яка реалізує ШОЗ ЦЗ. Ця функція інкапсульована в бібліотеці libconv_lib.so, що може використовуватися як з C/C++ програм, так і через розроблений Python-інтерфейс. Такий підхід забезпечує гнучкість у використанні ІТ в різних сценаріях — від вбудованих систем з обмеженими ресурсами до складних систем обробки ЦЗ, де потрібна інтеграція з високорівневими мовами програмування [22, 81].

Розроблена ІТ дозволяє досягти значного підвищення швидкодії ОЗ ЦЗ на ARM64 платформах порівняно з наявними рішеннями. Як буде показано в розд. 4, ВОЗ, реалізована в рамках запропонованої ІТ (дод. А.3–А.7), демонструє суттєве прискорення порівняно з аналогічними функціями з бібліотеки OpenCV, яка є де-факто стандартом в області обробки ЦЗ.

Таким чином, запропонована ІТ є спеціалізованим підходом до вирішення задачі обробки ОЗ ЦЗ на платформах ARM64, що включає як теоретичну базу у вигляді формалізованої нотації для опису векторизованих обчислень, так і практичну реалізацію, яка також включає високошвидкодійні 16-бітні ЯЗ, що значно підвищують якість ЦЗ, а ШОЗ підвищує швидкодію ОЗ ЦЗ порівняно з аналогами.

3.3 Архітектура інформаційної технології

Розроблена ІТ підвищення швидкодії обробки ЦЗ базується на методі ШОЗ для платформ ARM64 та складається з таких компонентів:

1. Набір спеціалізованих лінійних операторів для обробки ЦЗ під час 16-бітних обчислень, що складаються з 8-бітних цілочисельних ЯЗ розміром 3x3-5x5 та коефіцієнтів нормалізації (підрозд. 2.2), і забезпечують: низькочастотну (2.5), високочастотну (2.4) та субсмугову фільтрацію (2.6); контрастування та підвищення різкості зображень (2.9, 2.10); рекурентне

двократне масштабування з покращенням якості (2.11).

2. Метод ШОЗ — обґрунтований підхід до виконання ОЗ для ЦЗ, що ґрунтується на: використанні 16-бітних цілочисельних обчислень замість 32-бітних з плаваючою комою, що забезпечує оптимальне співвідношення точності та швидкодії; паралельній обробці даних за допомогою SIMD NEON64 (дод. А.1-А.10).
3. Алгоритмічне забезпечення ядра ІТ ШОЗ включає: алгоритм розгортання циклів ШОЗ для мінімізації накладних витрат на керування циклами (підрозд. 1.3.6); кеш-ефективну організацію доступу до даних зображення (дод. А.10); алгоритм кластеризації ЯЗ та формування ПМ (*index_map*) для оптимізації обробки розріджених матриць під час виконання ШОЗ (дод. А.7.1).
4. Формалізована нотація для опису векторизованих обчислень SIMD NEON64, що дозволяє абстрагуватися від конкретних асемблерних інструкцій та представляти операції у вигляді математичних виразів (дод. А.1-А.10 та в підрозд. 3.1).
5. Програмна реалізація ІТ у вигляді: обчислювального ядра conv_RGB_NEON (підрозд. 3.4.2), реалізованого як С-бібліотека (libconv_lib.so); програмного інтерфейсу для інтеграції з Python-застосунками.

Програмна складова розробленої ІТ ґрунтується на модульній архітектурі, яка забезпечує гнучкість, масштабованість та ефективність при вирішенні задач обробки ОЗ ЦЗ. Архітектура ІТ включає два основні рівні: високорівневий Python-застосунок, що забезпечує зручний користувацький інтерфейс та керування процесом обробки, за допомогою низькорівневої С-бібліотеки, яка реалізує високошвидкодійні векторизовані алгоритми обробки ЦЗ [81]. Детальна структура програмної складової ІТ представлена на (див. рис. 3.2), що демонструє компонентну модель системи. Розгляньмо основні компоненти та їхню взаємодію.

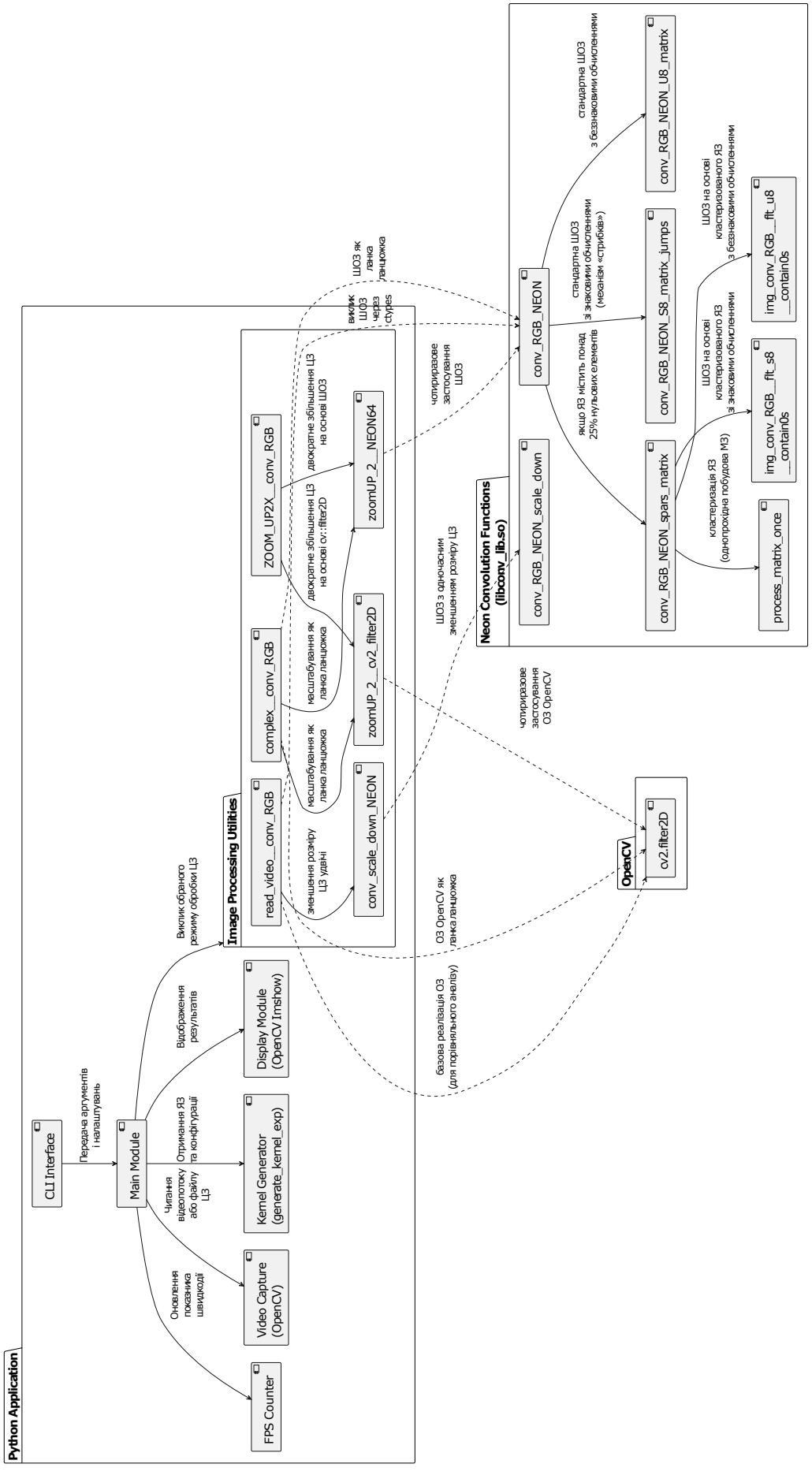


Рисунок 3.2 — Загальна схема інформаційної технології обробки ЦЗ з використанням векторизованої операції згортки

3.3.1 Структура Python-застосунку

Python-застосунок є високорівневою частиною ІТ і включає наступні компоненти:

- **CLI Interface** — інтерфейс командного рядка, який забезпечує взаємодію користувача з системою, дозволяючи налаштовувати параметри обробки ЦЗ, вибирати ЯЗ, або згенерувати нове, та режими роботи. Цей компонент відповідає за парсинг аргументів командного рядка та передачу налаштувань до Main Module;
- **Main Module** — основний модуль, що координує роботу всіх інших компонентів системи. Він виконує ініціалізацію, керує процесом обробки ЦЗ та забезпечує взаємодію між компонентами Python-застосунку та С-бібліотекою;
- **Video Capture (OpenCV)** — компонент, відповідальний за отримання вхідних даних з відеофайлів або камери, використовуючи можливості бібліотеки OpenCV. Цей модуль забезпечує послідовне читання кадрів для подальшої обробки;
- **Kernel Generator** — генератор ЯЗ, котрий створює різні розміри та типи ЯЗ: спарс ЯЗ; ЯЗ що містить від'ємні та додатні елементи (змішане ЯЗ); ромбовидні ЯЗ і тп. Цей компонент забезпечує гнучкість у виборі ЯЗ для різних задач обробки ЦЗ і формує ЯЗ відповідно до вимог користувача;
- **Display Module (OpenCV Imshow)** — модуль відображення результатів обробки, використовує функціонал відображення бібліотеки OpenCV. Дозволяє користувачу візуально оцінити результат обробки ЦЗ;
- **FPS Counter** — компонент для вимірювання швидкодії обробки кадрів/ЦЗ. Цей модуль є важливим для оцінки швидкодії різних алгоритмів обробки та конфігурацій.

Окремо слід розглянути модуль **Image Processing Utilities,** котрий використовує ШОЗ з **libconv_lib.so,** і тим самим надає можливість ЯІТ використовувати спеціалізовані операції збільшення/зменшення масштабу, підвищення якості ЦЗ тощо. Він/модуль являє собою набір функцій для обробки ЦЗ, які реалізують різні режими роботи системи, такі як:

- **read_video__conv_RGB** — режим обробки відеопотоку однією обраною ВОЗ. Через **CLI Interface**/прапорці задається конкретна реалізація: ЯІТ — conv_RGB_NEON(…), conv_scale_down_NEON(…), а також реалізації-

аналоги (cv::filter2D(…) та ін.). Саме цей режим використано для експериментального вимірювання швидкодії ШОЗ та її порівняння з аналогами (розд. 4);

- **ZOOM_UP2X__conv_RGB** — режим обробки відеопотоку операцією двократного збільшення лінійних розмірів ЦЗ. Через **CLI Interface**/прапорці обирається реалізація масштабування — на основі ЯІТ або на основі cv::filter2D(…). Цей режим використано для експериментального вимірювання швидкодії операцій масштабування (розд. 4);
- **complex__conv_RGB** — режим комплексної обробки ЦЗ, що дозволяє формувати ланцюжки з кількох послідовних операцій (детально розглянуто у підрозд. 3.5.5 та 3.7). Ланками ланцюжка можуть бути як ЯІТ та операції масштабування на його основі, так і реалізації-аналоги. Цей режим демонструє гнучкість розробленої ІТ і у вимірюваннях швидкодії не використовувався;
- **zoomUP_2__NEON64** — збільшення масштабу ЦЗ вдвічі на основі ВОЗ, реалізоване як чотириразове застосування ЯІТ — conv_RGB_NEON(…);
- **zoomUP_2__cv2_filter2D** — аналогічне масштабування з використанням ОЗ OpenCV — cv::filter2D(…). Реалізовано/створено для можливості проведення порівняльного аналізу швидкодії з ЯІТ;
- **conv_scale_down_NEON** — програмна обгортка над функцією С-бібліотеки conv_RGB_NEON_scale_down(…), що виконує зменшення лінійних розмірів ЦЗ удвічі з одночасним застосуванням ВОЗ.

Обидва режими — **read_video__conv_RGB** та **ZOOM_UP2X__conv_RGB** — реалізують єдину схему вимірювання: зчитування чергового ЦЗ, застосування обраної ВОЗ та оновлення показника швидкодії за допомогою компонента **FPS Counter**. Це забезпечує однакові умови вимірювання для ЯІТ та для реалізацій-аналогів, що є передумовою коректності порівняльного аналізу (розд. 4).

Як і Python-застосунок, так і його модулі/компоненти (як **Kernel Generator** та **Image Processing Utilities**), існують для демонстрації робастності ЯІТ та можливості проведення порівняльного аналізу швидкодії та якості ЯІТ з його аналогами. Також Python-застосунок є демонстрацією, що ЯІТ може бути вбудовано у ПЗ, що має брак швидкодії при обробці ОЗ ЦЗ.

3.3.2 Структура С-бібліотеки (libconv_lib.so)

Низькорівнева С-бібліотека є ЯІТ і реалізує високошвидкодійну ШОЗ для обробки ЦЗ. Бібліотека надає набір різних варіацій ВОЗ, оптимізованих для архітектури ARM64 з використанням NEON64, і залежних лише від вмісту ЯЗ. Основним API бібліотеки є функція conv_RGB_NEON(…), яка має кілька спеціалізованих варіантів для різних типів ЯЗ:

- **conv_RGB_NEON_scale_down** — спеціалізована версія функції для зменшення розміру ЦЗ з одночасним застосуванням ВОЗ, що дозволяє оптимізувати процес зниження роздільної здатності ЦЗ в 2 рази;
- **conv_RGB_NEON_spars_matrix** — версія функції для роботи з розрідженими ЯЗ, які містять значну кількість (>25%) нульових елементів, що було описано в (дод. А.7). Ця оптимізація дозволяє уникнути непотрібних множень на 0, і значно підвищити швидкодію за рахунок урахування знаку елементу ЯЗ (підрозд. 2.3.5, дод. А.7.1);
- **process_matrix_once** — компонент, що реалізує кластеризацію ЯЗ, як описано в (дод. А.7.1) та роботі [23]. Ця функція дозволяє оптимізувати обробку розріджених ЯЗ шляхом групування за критерієм знаку та того що значення елементу != 0;
- **img_conv_RGB__flt_s8__contain0s** — реалізація ШОЗ для розріджених знакових 8-біт ЯЗ;
- **img_conv_RGB__flt_u8__contain0s** — реалізація ШОЗ для розріджених беззнакових 8-біт ЯЗ;
- **conv_RGB_NEON_S8_matrix_jumps** — стандартна версія ВОЗ для знакових 8-бітних ЯЗ, оптимізована для 16-бітних обчислень, з реалізацією механізму ”стрибків”для пропуску множення на 0 та швидкого вибору «знаку дії»;
- **conv_RGB_NEON_U8_matrix** — стандартна версія ВОЗ для беззнакових 8-бітних ЯЗ, оптимізована для беззнакових 16-бітних обчислень. Найшвидша версія ШОЗ ЯІТ.

Таким чином, бібліотека надає набір спеціалізованих функцій, які дозволяють обрати найбільш ефективний варіант ШОЗ залежно від конкретної задачі та типу ЯЗ.

Таблиця 3.2 — Відповідність мнемонічних позначень функцій реальним ідентифікаторам у коді бібліотеки libconv_lib.so

Позначення у роботі	**Ідентифікатор у коді**	**Призначення**
conv_RGB_NEON	conv_RGB_NEON	Точка входу (API); вибір реалізації ШОЗ за вмістом ЯЗ
conv_RGB_NEON_ spars_matrix	lib_img_conv_RGB_img_us8flt__with_ zero__devide_on_pos_neg_mix	Гілка обробки розріджених ЯЗ (>25% нульових елементів)
process_matrix_once	process_matrix_once	Однопрохідна кластеризація ЯЗ та формування МЗ
img_conv_RGB__flt_ s8__contain0s	lib_img_conv_RGB_img__flt_s8__fcls_ mt16__x64___1x3_32_step___contain0s__ mix_pos_neg	ШОЗ на основі кластеризованого ЯЗ; знакові обчислення
img_conv_RGB__flt_ u8__contain0s	lib_img_conv_RGB_img__flt_u8__fcls_ mt16__x64___1x3_32_step___contain0s	ШОЗ на основі кластеризованого ЯЗ; беззнакові обчислення
conv_RGB_NEON_S8_ matrix_jumps	lib_img_conv_RGB_img__flt_s8__fcls_ mt16__x64___1x3_32_step__with_jumps	Стандартна ШОЗ; знакові обчислення, механізм «стрибків»
conv_RGB_NEON_U8_ matrix	lib_img_conv_RGB_img__flt_u8__fcls_ mt16__x64___1x3_32_step	Стандартна ШОЗ; беззнакові обчислення (найшвидша)
conv_RGB_NEON_ scale_down	conv_RGB_NEON_scale_down	ШОЗ із одночасним зменшенням лінійних розмірів ЦЗ удвічі

З міркувань читабельності тексту та рисунків у роботі вжито скорочені (мнемонічні) позначення функцій С-бібліотеки. Відповідність цих позначень реальним ідентифікаторам у програмному коді libconv_lib.so, що забезпечує можливість верифікації та відтворення наведених результатів, подано у (табл. 3.2).

Вихідний код бібліотеки libconv_lib.so та Python-інтерфейсу надається автором за запитом, що дозволяє перевірити коректність реалізації описаних методів та відтворити наведені у розд. 4 показники швидкодії.

3.3.3 Взаємодія між компонентами

Взаємодія між високорівневим Python-застосунком та низькорівневою C-бібліотекою здійснюється за допомогою модуля ctypes, який дозволяє викликати функції з динамічних бібліотек безпосередньо з Python-коду [22]. Цей підхід забезпечує ефективний «міст» між зручністю високорівневої мови програмування та швидкодією низькорівневого коду.

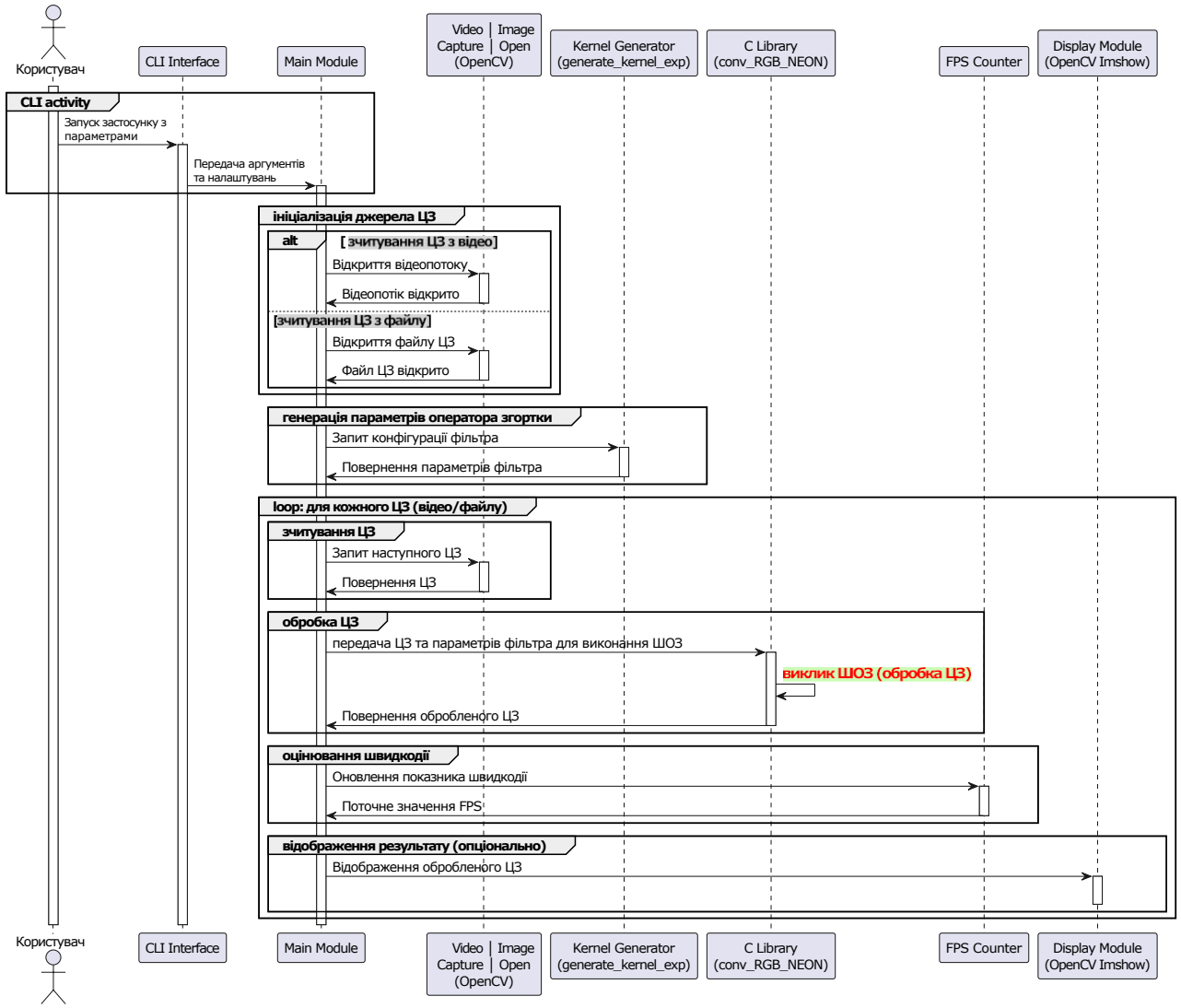


Рисунок 3.3 — Інтеграція ШОЗ як ключового елемента підвищення швидкодії

Процес взаємодії між компонентами можна представити у вигляді послідовності етапів обробки ЦЗ:

1. CLI Interface отримує параметри від користувача і передає їх до Main Module;
2. Main Module ініціалізує компоненти системи, включаючи завантаження C-бібліотеки;

3. Video Capture отримує ЦЗ із відеопотоку;
4. Kernel Generator створює або зчитує ЯЗ, відповідно до вимог;
5. Main Module викликає відповідну функцію з С-бібліотеки через модуль ctypes;
6. С-бібліотека виконує ШОЗ ЦЗ;
7. Оброблене ЦЗ повертається до Main Module;
8. FPS Counter оновлює інформацію про швидкість обробки;
9. Display Module відображає результат обробки (якщо потрібно);
10. Процес повторюється для зчитаного ЦЗ.

Для більш детального розуміння процесу обробки кадру можна представити діаграму послідовностей, яка показує взаємодію між компонентами в часі (див. рис. 3.3).

Ключовим елементом розробленої ІТ є ШОЗ ЦЗ, що детально описана у (дод. А.6). Інтеграція ЯІТ (ВОЗ) у загальну архітектуру ІТ здійснюється через С-бібліотеку, яка надає різні спеціалізовані варіанти виконання ШОЗ через функцію conv_RGB_NEON(…). Вибір конкретної версії функції здійснюється автоматично залежно від ЯЗ.

Важливою особливістю архітектури ІТ є підтримка 16-бітних цілочисельних обчислень ЯІТ, що забезпечує оптимальний баланс між точністю та швидкодією і вигідно вирізняє цей підхід серед аналогічних рішень.

Як показано у (дод. А.1-А.10), використання 16-бітних обчислень дозволяє ефективно обробляти 8-ми 16-бітних значень у векторному регістрі NEON64 за одну інструкцію та уникати потенційних переповнень під час обчислення ВСД.

Додатковим елементом оптимізації є механізм кластеризації ЯЗ, який дозволяє ефективно обробляти розріджені ЯЗ шляхом групування ненульових елементів. Цей підхід, описаний у (дод. А.7), забезпечує значне прискорення для специфічних типів ЯЗ.

Таким чином, архітектура розробленої ІТ забезпечує оптимальне використання можливостей ЦП ARM64 для високошвидкодійної обробки ОЗ ЦЗ. Модульна структура та різноманітні оптимізаційні стратегії дозволяють застосовувати ІТ для широкого спектра задач обробки ОЗ ЦЗ на платформах з обмеженими ресурсами, таких як SBC та вбудовані системи.

3.4 Програмна реалізація ядра інформаційної технології

ЯІТ реалізовано у вигляді динамічної бібліотеки libconv_lib.so (для Linux), що забезпечує високошвидкодійну реалізацію ШОЗ для обробки ЦЗ. Програмна реалізація ЯІТ базується на математичних основах ВОЗ, описаних у (розд. 2), та алгоритмах векторизації, представлених у (розд. 3).

3.4.1 Реалізація бібліотеки libconv_lib

Бібліотека libconv_lib реалізована мовою С з використанням Inline Assembly для оптимального використання NEON64 інструкцій. Основою бібліотеки є набір функцій, що забезпечують ефективну реалізацію ШОЗ для різних типів ЯЗ та режимів обробки (див. рис. 3.2). Структура бібліотеки включає:

1. Головну функцію conv_RGB_NEON — єдину точку входу для виклику ВОЗ, яка аналізує параметри ЯЗ та викликає найбільш ефективний варіант реалізації.
2. Спеціалізовані реалізації ВОЗ:
    - conv_RGB_NEON_U8_matrix — оптимізована для беззнакових 8-біт ЯЗ;
    - conv_RGB_NEON_S8_matrix_jumps — оптимізована для знакових 8-біт ЯЗ;
    - conv_RGB_NEON_spars_matrix — оптимізована для розріджених ЯЗ;
    - conv_RGB_NEON_scale_down — спеціалізована для зменшення розміру ЦЗ.
3. Допоміжні функції:
    - process_matrix_once — реалізує кластеризацію ЯЗ (дод. А.7.1);
    - img_conv_RGB__flt_s8__contain0s та img_conv_RGB__flt_u8__contain0s — реалізують ШОЗ для розріджених ЯЗ.

3.4.2 Інтерфейс функції conv_RGB_NEON

Інтерфейс функції conv_RGB_NEON спроектовано з урахуванням вимог до гнучкості та ефективності.

Таблиця 3.3 — сигнатура функції conv_RGB_NEON.

```
void conv_RGB_NEON( int src_cols, int src_rows, unsigned char *src,
                    unsigned char *dst, int *flt, int flt_cols,
                    int flt_rows, int divisor, int out_img_top,
                    int out_img_left );
```

Основні параметри функції (табл. 3.3) включають: ширину (src_cols) та висоту (src_rows) вхідного ЦЗ у пікселях; вказівник (src) на вхідне ЦЗ; вказівник (dst) на вихідне ЦЗ; вказівник (flt) на цілочисельне ЯЗ; ширину (flt_cols) та висоту (flt_rows) ЯЗ; коефіцієнт нормалізації для ЯЗ - divisor; зсув виведення результату відносно лівого верхнього краю ЦЗ (out_img_top, out_img_left).

Функція автоматично аналізує параметри ЯЗ (розмір, наявність нульових елементів, знаковість) та обирає найбільш ефективну реалізацію ВОЗ, що дозволяє забезпечити оптимальну швидкодію для різних типів задач (див. рис. 3.2).

3.4.3 Особливості програмної реалізації

Ключові особливості програмної реалізації ЯІТ, що забезпечують високу швидкодію обробки:

1. Використання 16-біт цілочисельних обчислень для ефективної реалізації ВОЗ. Як показано в (підрозд. 2.3), такий підхід забезпечує оптимальний баланс між точністю та швидкодією.
2. Векторизація обчислень з використанням NEON64 інструкцій для ARM64, що дозволяє обробляти до 8-ми 16-біт значень або 16-ти 8-бітних значень одночасно (підрозд. 3.1).
3. Оптимізація доступу до пам’яті через ефективне використання кеш-пам’яті процесора. Дані організовані таким чином, щоб мінімізувати кількість промахів кешу та забезпечити послідовний доступ до пам’яті.
4. Розгортання циклів обробки для зменшення накладних витрат на організацію циклів та підвищення ефективності використання конвеєра ЦП (підрозд. 1.3.6, 3.1).
5. Механізм кластеризації ЯЗ для ефективної обробки розріджених матриць, що дозволяє уникнути непотрібних обчислень при наявності нульових елементів (підрозд. 3.1, дод. А.7.1).
6. Спеціалізовані варіанти обробки ШОЗ ЦЗ для різних типів ЯЗ, що дозволяє

адаптувати алгоритм обробки до конкретних умов задачі (дод. А.4.7, А.7.1).

3.4.4 Компіляція бібліотеки для цільових платформ

Компіляція бібліотеки libconv_lib здійснюється з використанням компілятора GCC/Clang для Linux з активацією оптимізацій для ARM64 архітектури. Процес компіляції включає такі етапи:

1. Підготовка середовища розробки — встановлення необхідних інструментів (GCC, Make, CMake, ccache).
2. Налаштування опцій компіляції — активація оптимізацій (-O3/Ofast), включення підтримки NEON64 (-march=armv8-a+simd), та інших специфічних для ЦП ARM64 оптимізацій.
3. Компіляція бібліотеки — збірка об’єктних файлів та створення динамічної бібліотеки.
4. Тестування бібліотеки — перевірка коректності роботи на тестових даних.

Більш детальний опис процесу компіляції, розгортання, тестування бібліотеки та ЯІТ на цільових платформах подано у розд. 4.

3.4.5 Порівняння швидкодії з аналогами

Для обґрунтування ефективності запропонованого ЯІТ було проведено порівняльний аналіз швидкодії з бібліотекою OpenCV, яка є де-факто стандартом в області обробки ЦЗ. Порівнювалась швидкодія функції conv_RGB_NEON та функції cv::filter2D, що реалізує ОЗ ЦЗ в OpenCV.

Результати порівняння для різних розмірів ЦЗ та ЯЗ показали, що реалізований ЯІТ забезпечує прискорення до 11,1 раза (Orange Pi 5 Pro) та до 10,2 раза (Raspberry Pi 4) порівняно з OpenCV на платформах ARM64. Найбільший приріст швидкодії спостерігається для розріджених ЯЗ та для випадків, коли розмір ЦЗ дозволяє ефективно використовувати кеш-пам’ять процесора.

Такий значний приріст швидкодії досягається завдяки:

- Оптимізації обчислень для конкретної архітектури ARM64.
- Ефективному використанню NEON64 інструкцій.
- Спеціалізованим оптимізаціям для різних типів ЯЗ.
- Мінімізації накладних витрат на організацію обчислень.

Детальні результати порівняльного аналізу швидкодії представлено у розд. 4.

Таким чином, програмна реалізація ЯІТ забезпечує високошвидкодійну обробку ЦЗ на ARM64 платформах, значно перевершуючи наявні аналоги за показниками швидкодії при збереженні якості обробки.

3.5 Методика застосування ІТ для обробки різних типів ЦЗ

Розроблена ІТ забезпечує ефективну реалізацію ШОЗ для ЦЗ та сценаріїв її використання. У цьому підрозділі описуються методики застосування ІТ для вирішення типових задач обробки ОЗ ЦЗ, відеопотоку, масштабування ЦЗ за допомогою ЯІТ.

3.5.1 Обробка статичних зображень

Застосування розробленої ІТ для обробки статичних ЦЗ реалізується через виклик функції conv_RGB_NEON(...), що здійснює ШОЗ для окремого ЦЗ.

Таблиця 3.4 — API ІТ

Мова	Назва функції
python/C/C++	lib.conv_RGB_NEON(ctypes.c_int(cols), ctypes.c_int(rows), src_ptr, dst_prt, flt_ptr, ctypes.c_int(flt_cols), ctypes.c_int(flt_rows), ctypes.c_int(dekernel_divisor_i32), ctypes.c_int(flt_cols//2), ctypes.c_int(flt_rows//2))
python	zoomUP_2__NEON64(cols, rows, src_ptr, dst_frame_np__list, flt_vec_data__list, flt_cols, flt_rows, dekernel_divisor_i32__list)
python	conv_scale_down_NEON(frame, flt_ptr, flt_cols, flt_rows, dekernel_divisor_i32, scale)
python	complex__conv_RGB(args.flt_path, in_file, str(args.flt_type), args.show_result)

У (табл. 3.4) представлено основні функції API розробленої ІТ, які забезпечують різні режими обробки ЦЗ: від базової операції згортки до масштабування та комплексної обробки ЦЗ.

Процес обробки статичного зображення включає такі етапи:

1. Завантаження вхідного зображення - зчитування ЦЗ з файлу або іншого джерела даних, за допомогою бібліотечного виклику cv2.imread бібліотеки OpenCV.
2. Підготовка ЯЗ - формування або завантаження ЯЗ.

3. Обробка ЦЗ - виклик функції conv_RGB_NEON з відповідними параметрами. Для статичних ЦЗ типовий виклик функції має вигляд, що надано у (табл. 3.4).
4. Збереження або відображення результату на екрані або, залежно від вимог задачі, результат перформанс тесту чи тесту якості.

Використання ШОЗ для статичних ЦЗ дозволяє ефективно вирішувати такі задачі, як фільтрація шумів, виділення контурів, розмиття та інші операції просторової фільтрації.

3.5.2 Обробка відеопотоків

Обробка відеопотоків є однією з найбільш ресурсомістких задач обробки ЦЗ, що вимагає високої швидкодії алгоритмів. Розроблена ІТ забезпечує ефективну обробку відеопотоків завдяки високій швидкодії ВОЗ. Процес обробки відеопотоку ШОЗ представлено функцією read_video__conv_RGB у Python-інтерфейсі та включає такі етапи:

1. Відкриття відеопотоку - встановлення з’єднання з джерелом відео, яким може бути відеофайл або камера (за допомогою cv2.VideoCapture).
2. Підготовка ЯЗ - аналогічно до обробки статичних ЦЗ.
3. Циклічна обробка кадрів - послідовне зчитування та обробка кожного кадру відеопотоку:
    - Зчитування кадру з джерела відео.
    - Виклик функції conv_RGB_NEON для обробки кадру.
    - Відображення або збереження обробленого кадру (за потреби).
    - Вимірювання швидкості обробки (FPS).
4. Завершення обробки.

Ключовою особливістю обробки відеопотоків є необхідність забезпечення достатньої швидкодії для обробки в реальному часі. Розроблена ІТ забезпечує швидкодію, достатню для обробки відео з роздільністю до HD (1280×720) на платформах ARM64 з частотою до 30–60 кадрів за секунду, залежно від складності ЯЗ та швидкодії конкретної платформи.

3.5.3 Масштабування зображень з використанням ядра ІТ

Одним із спеціалізованих застосувань розробленої ІТ є масштабування ЦЗ з одночасним покращенням якості. На відміну від стандартних методів масштабування, які часто призводять до втрати деталей та появи артефактів, підхід на основі ШОЗ дозволяє зберегти та навіть покращити якість зображення при зміні його розміру.

Для збільшення масштабу зображення розроблена ІТ пропонує два основних методи:

1. zoomUP_2__NEON64 - збільшення масштабу вдвічі з використанням спеціалізованих ядер згортки та ВОЗ. Метод базується на застосуванні чотирьох різних ЯЗ для формування чотирьох різних ЦЗ, які потім об'єднуються в одне зображення з подвоєним розміром (табл. 3.4) - zoomUP_2__NEON64(...);
2. complex__conv_RGB - послідовне застосування декількох ОЗ + можливість масштабування та постобробка за допомогою ШОЗ - conv_RGB_NEON(...). Цей метод дозволяє формувати складні ланцюжки обробки для досягнення бажаних ефектів.

Математичні основи масштабування на основі ШОЗ описані у підрозд. 2.4, а їх реалізація — у підрозд. 3.5–3.8.

Для зменшення масштабу розроблена спеціалізована версія функції conv_RGB_NEON_scale_down(...) (див. табл. 3.4), яка поєднує ОЗ зі зменшенням розміру ЦЗ. Цей підхід дозволяє не лише зменшити розмір, але й забезпечити фільтрацію для уникнення аліасингу та інших артефактів. Використання спеціалізованої функції замість послідовного застосування згортки та зменшення розміру, дозволяє значно підвищити швидкодію обробки.

3.5.4 Використання різних типів ядер згортки.

Розроблена ІТ підтримує роботу з різними типами ЯЗ, що дозволяє адаптувати її для вирішення широкого спектру задач обробки ЦЗ. Вибір конкретного типу ЯЗ залежить від вимог задачі та бажаного ефекту обробки.

Модуль Kernel Generator (generate_kernel_exp) забезпечує формування різних типів ЯЗ:

1. Стандартні ЯЗ - типові ЯЗ для операцій розмиття, підвищення різкості, виділення контурів тощо. Ці ЯЗ можуть бути сформовані за допомогою

функції generate_convolution_kernel або завантажені з файлу.

2. Спеціалізовані ЯЗ для масштабування - набір з чотирьох ЯЗ для реалізації алгоритму збільшення масштабу (підрозд. 3.3.1).
3. Розріджені ЯЗ - ЯЗ з великою кількістю нульових елементів, для яких застосовується спеціалізований алгоритм обробки, описаний у (дод. А.7).

Для забезпечення максимальної швидкодії ІТ автоматично обирає оптимальний варіант реалізації ШОЗ залежно від типу ЯЗ:

1. Для беззнакових ЯЗ - використовується реалізація conv_RGB_NEON_U8_matrix.
2. Для знакових ЯЗ - застосовується conv_RGB_NEON_S8_matrix_jumps.
3. Для розріджених ЯЗ - використовується conv_RGB_NEON_spars_matrix.
4. Для ЯЗ зменшення масштабу - застосовується спеціалізована реалізація conv_RGB_NEON_scale_down.

Усі ці функції детально описані в (підрозд. 3.3.2). Такий підхід забезпечує оптимальне використання обчислювальних ресурсів для кожного конкретного випадку застосування ІТ.

3.5.5 Комплексна обробка ЦЗ.

Для вирішення складних задач обробки ОЗ ЦЗ розроблена ІТ підтримує послідовне застосування декількох ОЗ з різними ЯЗ. Цей підхід реалізовано у функції complex__conv_RGB(...), яка послідовно викликає функції обробки, передаючи вихідне ЦЗ кожної з них на вхід наступної. Послідовність задається аргументом (args.flt_type) — рядком, у якому операції розділено оператором «+», а парний до нього аргумент (args.flt_path) містить послідовність шляхів до відповідних ЯЗ. Наприклад, ланцюжок conv_RGB_NEON(…) → zoomUP_2__NEON64(…) → conv_RGB_NEON(…) виконує попередню фільтрацію, дворазове збільшення масштабу та завершальну фільтрацію отриманого ЦЗ. Кожна операція в ланцюжку отримує результат попередньої операції як вхідні дані, що дозволяє реалізувати складні алгоритми обробки через комбінацію простих ОЗ.

3.6 Розробка програмного інтерфейсу для ІТ

Для забезпечення зручності використання розробленої ІТ було створено високорівневий програмний інтерфейс, який дозволяє ефективно застосовувати

ШОЗ у різних задачах обробки ЦЗ.

3.6.1 Python-обгортка для взаємодії з бібліотекою libconv_lib.

Основою програмного інтерфейсу є Python-обгортка, яка забезпечує доступ до функцій бібліотеки libconv_lib через механізм ctypes [87]. Цей підхід дозволяє поєднати високу швидкодію низькорівневого коду C/C++ з гнучкістю та зручністю використання Python.

Таблиця 3.5 — Процес завантаження та ініціалізації бібліотеки libconv_lib

Завантаження та ініціалізації бібліотеки:

```
path_to_lib = args.lib_path
if path_to_lib == "":
  if os.name == 'posix':
    path_to_lib = os.path.join(cur_dir_path, 'libconv_lib.so')
  else:
    path_to_lib = os.path.join(cur_dir_path, 'libconv_lib.dll')
lib = ctypes.CDLL(path_to_lib)
```

Приклад визначення типів параметрів функцій:

```
lib.conv_RGB_NEON.argtypes = [ctypes.c_int,  ctypes.c_int,
ctypes.POINTER(ctypes.c_ubyte), ctypes.POINTER(ctypes.c_ubyte),
ctypes.POINTER(ctypes.c_int), ctypes.c_int, ctypes.c_int,
ctypes.c_int, ctypes.c_int, ctypes.c_int]

lib.conv_RGB_NEON.restype = None
```

Процес завантаження та ініціалізації бібліотеки libconv_lib реалізується, як показано у (табл. 3.5). Цей фрагмент демонструє автоматичне визначення шляху до бібліотеки залежно від операційної системи, що забезпечує кросплатформну сумісність розробленого інтерфейсу. Після завантаження бібліотеки необхідно визначити типи параметрів функцій, які будуть викликатися з Python. Це здійснюється за допомогою механізму argtypes бібліотеки ctypes (табл. 3.5). Таке визначення типів забезпечує коректну передачу даних між Python та C/C++ кодом, що є критичним для безпомилкової роботи всієї системи.

3.6.2 Інтеграція з NumPy для ефективної обробки масивів даних.

Для забезпечення ефективної обробки багатовимірних масивів даних, які представляють ЦЗ, розроблений інтерфейс ЯІТ - conv_RGB_NEON(...) інтегрується з бібліотекою NumPy [88]. Ця інтеграція забезпечує зручний механізм конвертації даних та передачі їх в С-функції, а також дозволяє використовувати ефективні структури даних NumPy для зберігання ЦЗ.

Таблиця 3.6 — Конвертація ЦЗ у форматі NumPy array до вказівників C/C++

```
frame_np = np.asarray(frame)
src_ptr = frame_np.ctypes.data_as(ctypes.POINTER(ctypes.c_ubyte))
dst_frame = np.zeros_like(frame)
dst_frame_np = np.asarray(dst_frame)
dst_prt = dst_frame_np.ctypes.data_as(ctypes.POINTER(\
          ctypes.c_ubyte))
flt_ptr = (ctypes.c_int * (flt_cols * flt_rows))(*flt_array_1D)
```

Конвертація ЦЗ у форматі NumPy array до вказівників C/C++ показана у (табл. 3.6). Цей механізм дозволяє ефективно передавати дані між Python та С-кодом без копіювання масивів, що значно підвищує швидкодію обробки великих ЦЗ. Аналогічно, для ЯЗ використовується той самий механізм конвертації. Цей підхід забезпечує коректне перетворення двовимірного масиву ЯЗ у одновимірний масив C/C++ типів, який може бути переданий у функцію conv_RGB_NEON(...).

3.6.3 Реалізація CLI-інтерфейсу в ІТ

Для забезпечення зручного використання розробленої ІТ було реалізовано інтерфейс командного рядка (CLI) на основі модуля argparse [90]. Цей інтерфейс дозволяє гнучко налаштовувати параметри обробки, вибирати режими роботи та вхідні/вихідні файли.

Таблиця 3.7 — Основні параметри CLI-інтерфейсу

Параметр	Скорочення	Тип	Опис
`--input_video_file`	-i	`str`	Шлях до вхідного відеофайлу
`--imge_chan`	-e	`str`	Формат кольорів (RGB/GRAY/AUTO)
`--lib_path`	-l	`str`	Шлях до бібліотеки (dll/so)
`--show_result`	-s	`bool`	Показ результату обробки
`--flt_path`	-f	`str`	Шлях до файлу з ЯЗ
`--flt_cols`	-c	`int`	Ширина ЯЗ
`--flt_rows`	-r	`int`	Висота ЯЗ
`--flt_type`	-t	`str`	Тип фільтрації (conv_RGB_NEON / cv2_filter2D / …)

Параметр (–flt_path) може бути вказаний кілька разів для послідовного застосування/завантаження різних ЯЗ для ШОЗ ЦЗ, що дозволяє формувати складні ланцюги обробки (табл. 3.7).

Вищеописаний підхід забезпечує зручний та гнучкий спосіб налаштування параметрів обробки через командний рядок [91, 92].

3.6.4 Передача даних між Python та C-бібліотекою

Ефективна передача даних між Python та C/C++ бібліотекою є критичним аспектом розробленого програмного інтерфейсу. Розгляньмо основні механізми передачі різних типів даних:

1. Передача ЦЗ - здійснюється за допомогою перетворення NumPy array у вказівники C-масивів, як описано в (підрозд. 3.6.2).
2. Передача ЯЗ - включає конвертацію двовимірного масиву Python у одновимірний C-масив з відповідним типом даних.
3. Передача скалярних параметрів - здійснюється шляхом явного перетворення Python-типів у відповідні C-типи за допомогою конструкторів ctypes - (див. табл. 3.4 та атрибути функції conv_RGB_NEON(...)).
4. Отримання результатів обробки - оскільки вказівник на вихідне зображення передається в C-функцію, після завершення обробки результат автоматично доступний у відповідному NumPy array без необхідності додаткової передачі даних.

Такий підхід забезпечує ефективну передачу даних між Python та C-кодом без надлишкового копіювання, що є особливо важливим при обробці великих обсягів даних, таких як відеопотоки високої роздільної здатності.

3.6.5 Утилітарні функції програмного інтерфейсу

Для спрощення використання розробленої ІТ було реалізовано набір утилітарних функцій, які інкапсулюють типові сценарії застосування технології:

1. read_video__conv_RGB — функція для обробки відеопотоку з використанням одного ЯЗ;
2. ZOOM_UP2X__conv_RGB - функція для збільшення масштабу відеопотоку вдвічі з використанням спеціалізованих ЯЗ;
3. complex__conv_RGB - функція для послідовного застосування декількох ОЗ з різними ЯЗ;
4. get_filter_from_file - функція для зчитування ЯЗ з текстового файлу або генерації ЯЗ за заданими параметрами;
5. combine_frames_checkerboard - функція для об’єднання чотирьох ЦЗ у одне за принципом шахової дошки, що використовується в алгоритмі збільшення масштабу;
6. FpsCounter - клас для вимірювання швидкості обробки кадрів (FPS), що дозволяє оцінювати швидкодію різних алгоритмів обробки.

Ці функції значно спрощують використання розробленої ІТ для типових задач обробки ЦЗ, дозволяючи зосередитися на налаштуванні параметрів обробки, а не на деталях реалізації.

3.6.6 Особливості та переваги розробленого програмного інтерфейсу

Розроблений програмний інтерфейс має ряд особливостей та переваг, які забезпечують його ефективність та зручність використання:

1. Гнучкість конфігурації - можливість налаштування різних параметрів обробки через CLI-інтерфейс.
2. Кросплатформність.
3. Висока швидкодія - ефективна передача даних між Python та C-кодом без надлишкового копіювання.
4. Розширюваність - можливість легкого додавання нових функцій та режимів

обробки за рахунок модульної структури інтерфейсу.

5. Інтеграція з екосистемою Python - використання стандартних бібліотек (NumPy, OpenCV), що забезпечує сумісність з іншими інструментами обробки ЦЗ;
6. Зрозумілість та простота використання - наявність високорівневих функцій, які інкапсулюють типові сценарії застосування технології.

Таким чином, розроблений програмний інтерфейс забезпечує зручний та ефективний спосіб використання ШОЗ для вирішення різноманітних задач обробки ЦЗ на платформах ARM64.

3.7 Комплексне застосування ІТ для задач обробки ЦЗ

Розроблена ІТ дозволяє вирішувати широкий спектр задач обробки ЦЗ завдяки гнучкості архітектури, високій швидкодії ЯІТ та зручному програмному інтерфейсу. У даному підрозділі розглядаються комплексні сценарії застосування розробленої ІТ для вирішення практичних задач обробки ЦЗ.

3.7.1 Використання ІТ для реалізації складних операцій з масштабуванням зображень

Масштабування ЦЗ з одночасним підвищенням якості є однією з найбільш ресурсомістких операцій в обробці ЦЗ. Розроблена ІТ пропонує ефективне рішення цієї задачі завдяки спеціалізованій функції zoomUP_2__NEON64(...), яка реалізує алгоритм збільшення масштабу ЦЗ вдвічі з використанням ШОЗ та спеціальних ЯЗ.

Процес масштабування ЦЗ включає такі етапи:

1. Зчитування спеціалізованих ЯЗ - формування чотирьох різних ядер згортки, кожне з яких оптимізоване для відновлення певної частини інформації ЦЗ.
2. Обробка вхідного ЦЗ - застосування кожного з чотирьох ЯЗ до вхідного ЦЗ з використанням функції conv_RGB_NEON(...), що дозволяє отримати чотири різних ЦЗ, кожне з яких містить певні особливості вхідного ЦЗ.
3. Об’єднання результатів - комбінування чотирьох ЦЗ у одне з подвоєним розміром за принципом шахової дошки, що реалізується функцією combine_frames_checkerboard(...).

Цей підхід дозволяє не лише збільшити розмір ЦЗ, але й підвищити його якість, зберігаючи дрібні деталі та контури. На відміну від стандартних методів масштабування, таких як білінійна або бікубічна інтерполяція, підхід на основі ШОЗ забезпечує кращу якість результату, особливо для ЦЗ з чіткими контурами та дрібними деталями.

Для практичної реалізації масштабування ЦЗ в розробленій ІТ використовується функція ZOOM_UP2X__conv_RGB(...), яка інкапсулює всі етапи алгоритму та забезпечує зручний інтерфейс для його використання. Ця функція може бути викликана як напряму з Python-коду, так і через CLI-інтерфейс, що забезпечує гнучкість використання.

3.7.2 Послідовне застосування кількох ядер згортки для досягнення складних ефектів

Для вирішення складних задач обробки ЦЗ часто необхідно послідовно застосувати декілька ОЗ з різними ЯЗ. Розроблена ІТ підтримує цей сценарій завдяки функції complex__conv_RGB, яка дозволяє формувати ланцюжки операцій обробки у вигляді строки з операціями, розділеними знаком ”+”.

Типові ланцюжки операцій включають:

1. Фільтрація шуму з подальшим підвищенням різкості - послідовне застосування ЯЗ для гаусівського розмиття та ЯЗ для виділення контурів, що дозволяє видалити шум та підвищити чіткість ЦЗ.
2. Масштабування з попередньою та наступною обробкою - ланцюжок виду conv_RGB_NEON(…) → zoomUP_2__NEON64(…) → conv_RGB_NEON(…), де перша функція підготовлює ЦЗ до масштабування (наприклад, видаляє шум), друга виконує масштабування, а третя покращує якість результату (наприклад, підвищує різкість);
3. Послідовне виділення різних структур ЦЗ - застосування різних ЯЗ для виділення горизонтальних, вертикальних та діагональних структур з наступним об’єднанням результатів.
4. Комбінована фільтрація для підвищення якості ЦЗ - послідовне застосування різних фільтрів для покращення різних аспектів якості ЦЗ (яскравість, контрастність, чіткість).

Кожна операція в ланцюжку отримує результат попередньої операції як вхідні дані, що дозволяє поетапно перетворювати ЦЗ для досягнення бажаного

результату. Такий підхід забезпечує гнучкість та ефективність обробки, дозволяючи адаптувати алгоритм до конкретних вимог задачі.

Таблиця 3.8 — Приклад виклику функції complex__conv_RGB.

виклик функції:
complex__conv_RGB(args.flt_path, in_file, str(args.flt_type), args.show_result)
аргумент (args.flt_type) визначає послідовність операцій:
”conv_RGB_NEON+2x_zoom__NEON64+conv_RGB_NEON”
аргумент (args.flt_path) містить перелік шляхів до файлів із ЯЗ

Для практичної реалізації послідовного застосування кількох ядер згортки використовується синтаксис, як показано у (табл. 3.8). Кожен елемент цього рядка вказує функцію, яку буде викликано на відповідному кроці ланцюжка: елемент conv_RGB_NEON — функцію conv_RGB_NEON(…), а елемент 2x_zoom__NEON64 — функцію zoomUP_2__NEON64(…), що чотири рази застосовує ШОЗ для чотирьох ЯЗ A, B, C, D (див. (2.13)).

3.7.3 Оптимізація обробки для різних розмірів ядер згортки

Ефективність обробки ЦЗ значною мірою залежить від розміру ЦЗ та ЯЗ. Розроблена ІТ включає ряд оптимізацій, які дозволяють адаптувати алгоритм обробки до конкретних умов задачі.

Оптимізація для різних розмірів та типів ЯЗ

Залежно від характеристик ЯЗ (розмір, наявність нульових елементів, знаковість), автоматично обирається оптимальний варіант реалізації ВОЗ:

1. **Для малих ЯЗ (3×3, 5×5)** - використовується оптимізований алгоритм з розгортанням циклів та максимальним використанням векторизації.
2. **Для розріджених ЯЗ** - використовується алгоритм з кластеризацією елементів, що дозволяє уникнути непотрібних обчислень для нульових елементів.
3. **Для ЯЗ з різною знаковістю** - використовується алгоритм з кластеризацією елементів, що дозволяє використати групи однакових за знаком елементів ЯЗ, що підвищує швидкодію за рахунок підключення додаткової логіки всередині АЛП для групи значень ЯЗ, що дійсно потребує цього.

Такий диференційований підхід забезпечує максимальну швидкодію обробки для різних комбінацій розмірів та групп елементів ЯЗ.

3.7.4 Обмеження та рекомендації щодо використання ІТ

При використанні розробленої ІТ слід враховувати певні обмеження та рекомендації, які дозволять досягти оптимальних результатів.

Обмеження технології

1. **Обмеження на розмір ЯЗ** - для ефективної векторизації розмір ЯЗ обмежується максимальним значенням (31×31). Більші ЯЗ можуть бути використані, але це призведе до зниження швидкодії.
2. **Обмеження на тип даних** - ЯІТ оптимізоване для роботи з 8-бітними RGB ЦЗ, та використовує 16-бітні обчислення. Підтримка інших форматів (наприклад, 16-біт ЦЗ або операцій з плаваючою комою) може вимагати модифікації алгоритму.
3. **Архітектурні обмеження** - технологія оптимізована для ARM64 архітектури з підтримкою NEON64. Використання на інших архітектурах (x86, RISC-V) вимагатиме адаптації алгоритмів векторизації.
4. **Обмеження на розмір ЦЗ** - для забезпечення швидкодії в режимі реального часу (30+ FPS) на типових SBC рекомендується обмежити розмір ЦЗ до HD (1280×720) або FullHD (1920×1080) залежно від складності ЯЗ.

Рекомендації щодо використання

1. **Вибір оптимального ЯЗ** - для досягнення максимальної швидкодії рекомендується використовувати ЯЗ з розміром, що є кратним 2 (наприклад, 4×4, 6×6, 8×8). Це забезпечує оптимальне вирівнювання даних та ефективне використання векторизації.
2. **Використання розріджених ЯЗ** - для складних операцій фільтрації рекомендується використовувати розріджені ЯЗ, де значна частина елементів є нульовими. Це дозволить скористатися перевагами оптимізації для розріджених матриць.
3. **Послідовне застосування простих ЯЗ** - замість одного складного ЯЗ великого розміру рекомендується використовувати послідовність простих ЯЗ малого розміру. Це часто дає кращі результати як з точки зору якості обробки, так і з точки зору швидкодії.
4. **Балансування навантаження** - при обробці відеопотоків в режимі ре-

ального часу рекомендується підбирати параметри обробки (розмір ЯЗ, кількість операцій) таким чином, щоб забезпечити стабільну частоту кадрів (FPS).

5. **Використання попередньої обробки** - для покращення результатів ШОЗ рекомендується застосовувати попередню обробку ЦЗ (наприклад, нормалізацію, видалення шуму), що може бути реалізовано через ланцюжок операцій в complex__conv_RGB.

Дотримання цих рекомендацій дозволить максимально ефективно використовувати можливості розробленої ІТ для вирішення різноманітних задач ОЦЗ.

3.7.5 Приклади комплексного застосування ІТ для типових задач ОЦЗ

Розгляньмо приклади комплексного застосування розробленої ІТ для вирішення типових задач ОЦЗ.

Для **обробки відеопотоку з камери спостереження** задля поліпшення якості ЦЗ та виділення рухомих об’єктів можна використати наступний ланцюжок операцій:

1. Фільтрація шуму за допомогою гаусівського розмиття (conv_RGB_NEON з відповідним ЯЗ).
2. Підвищення контрастності та виділення контурів (conv_RGB_NEON з ЯЗ для виділення контурів).
3. Обробка в режимі реального часу з вимірюванням FPS.

Такий підхід дозволяє значно поліпшити якість відеопотоку та підготувати його для подальшого аналізу (наприклад, для систем комп’ютерного зору).

Для **відновлення та масштабування старих фотографій** з низькою роздільною здатністю можна використати такий ланцюжок операцій:

1. Фільтрація шуму та артефактів (conv_RGB_NEON з відповідним ЯЗ).
2. Збільшення масштабу вдвічі (zoomUP_2__NEON64).
3. Підвищення різкості та контрастності (conv_RGB_NEON з відповідним ЯЗ).

Цей підхід дозволяє значно поліпшити якість старих фотографій, відновити дрібні деталі та підготувати їх для друку або використання у високоякісних цифрових матеріалах.

Для **обробки медичних ЦЗ** (як от - рентгенівських знімків) задля поліпшення їх діагностичної цінності можна використати такий ланцюжок операцій:

1. Фільтрація шуму (conv_RGB_NEON з відповідним ЯЗ).
2. Підвищення контрастності та виділення структур різної інтенсивності (conv_RGB_NEON з відповідним ЯЗ).
3. Масштабування для детального аналізу окремих областей (zoomUP_2__NEON64).

Цей підхід дозволяє поліпшити якість медичних ЦЗ, зробити більш помітними дрібні структури та підготувати ЦЗ для автоматизованого аналізу. Таким чином, розроблена ІТ надає гнучкі та ефективні засоби для комплексного вирішення широкого спектру задач ОЦЗ, забезпечуючи високу швидкодію обробки на платформах ARM64.

3.8 Висновки

У даному розділі представлено розроблену ІТ обробки ЦЗ із застосуванням ВОЗ, оптимізовану для платформи ARM64. За результатами розробки та дослідження ІТ можна зробити наступні висновки:

1. Розроблено комплексну ІТ, що забезпечує високошвидкодійну обробку ЦЗ на платформах ARM64. Технологія базується на трьох ключових принципах: векторизації обчислень ЯІТ, модульній архітектурі та уніфікації опису векторизованих обчислень. Запропонована формалізована нотація для NEON64 є важливим науковим внеском, що не лише спрощує опис та розробку векторизованих алгоритмів, але й створює теоретичний фундамент для формального опису, аналізу та верифікації SIMD-обчислень незалежно від конкретної платформи.

2. Створена архітектура ІТ забезпечує ефективну взаємодію між компонентами системи та оптимальне використання обчислювальних ресурсів платформи. Ключовим елементом архітектури є функція conv_RGB_NEON, яка реалізує ВОЗ, оптимізовану для різних типів ЯЗ та задач обробки ЦЗ. Модульна структура ІТ забезпечує гнучкість та розширюваність, дозволяючи легко додавати нові функціональні можливості.

3. Програмна реалізація ядра ІТ у вигляді бібліотеки libconv_lib використовує оптимізовані NEON64 інструкції та механізми кластеризації ЯЗ для досягнення максимальної швидкодії. Порівняльний аналіз з аналогами показав прискорення до 11,1 раза (Orange Pi 5 Pro) та до 10,2 раза (Raspberry Pi 4) порівняно з бібліотекою OpenCV на платформі ARM64, що підтверджує ефективність

запропонованого підходу.

4. Розроблені методики застосування ІТ для різних типів ЦЗ демонструють універсальність технології та її здатність ефективно вирішувати широкий спектр задач обробки ЦЗ. Особливо значущими є методики масштабування ЦЗ з одночасним підвищенням якості та обробки відеопотоків у режимі реального часу.

5. Створений програмний інтерфейс забезпечує зручний доступ до функціональності ІТ через високорівневу мову Python, що значно спрощує інтеграцію технології в наявні системи обробки ЦЗ. Інтеграція з бібліотекою NumPy та реалізація CLI-інтерфейсу розширюють можливості застосування ІТ у різних сценаріях використання.

6. Комплексне застосування ІТ для типових задач обробки ЦЗ, таких як фільтрація, масштабування та покращення якості, підтверджує практичну цінність розробленої технології. Визначені обмеження та рекомендації щодо використання ІТ дозволяють оптимально використовувати її можливості в реальних умовах.

7. Розроблена формалізована нотація для опису векторизованих обчислень представляє собою самостійний теоретичний результат, що має наукову цінність поза межами конкретної реалізації. Ця нотація (див. підрозд. 3.1) заповнює важливу методологічну прогалину в формалізації SIMD-обчислень, створюючи універсальний інструмент для абстрактного представлення векторних операцій, що може бути застосований до різних архітектур та задач паралельних обчислень. Теоретична значущість нотації підтверджується можливістю її використання для: формального доведення коректності векторизованих алгоритмів; систематичного дослідження та порівняння різних стратегій векторизації; формалізації та класифікації типових шаблонів SIMD-обчислень; створення теоретичних моделей для прогнозування швидкодії.

Таким чином, розроблена ІТ забезпечує ефективне рішення для обробки ЦЗ на платформі ARM64, поєднуючи високу швидкодію, гнучкість та зручність використання. Запропонований підхід має як практичну значущість, демонструючи значне прискорення обробки порівняно з наявними аналогами, так і теоретичну цінність, пропонуючи новий формальний метод опису векторизованих обчислень, що може стати основою для подальших наукових досліджень у галузі паралельних обчислень та обробки даних.

РОЗДІЛ 4. ДОСЛІДЖЕННЯ ШВИДКОДІЇ АЛГОРИТМУ ЛІНІЙНОЇ ОПЕРАЦІЇ ЗГОРТКИ З ВИКОРИСТАННЯМ SIMD-ОПТИМІЗАЦІЙ ПРИ ОБРОБЦІ ЦИФРОВОГО ЗОБРАЖЕННЯ ЗА УМОВИ ВИКОРИСТАННЯ 16-БІТНИХ ОБЧИСЛЕНЬ

У цьому розділі представлено результати експериментального дослідження швидкодії розробленої ІТ та її ЯІТ – ШОЗ ЦЗ conv_RGB_NEON(...), призначеної для високопродуктивної обробки ЦЗ на платформах з архітектурою ARM64 із застосуванням 16-бітних цілочисельних обчислень та SIMD-інструкцій NEON64. Викладено опис експерименту (підрозд. 4.1), що включає його мету та систематичний підхід до тестування. Детально описано використані прилади та апаратне забезпечення (підрозд. 4.2), а також його програмне налаштування (підрозд. 4.3) та загальний опис ПЗ, задіяного у дослідженнях (підрозд. 4.4), включаючи середовища розробки, компілятори, інтерпретатори та інструменти оптимізації. Представлено загальний опис методики тестування швидкодії ІТ/ЯІТ за допомогою Shell-скриптів (підрозд. 4.5), що охоплює базовий алгоритм тестування швидкодії, етапи тестування C/C++ функції conv_RGB_NEON(...) та її Python-обгорток, а також деталізовану структуру процедур тестування. Надано аналіз показників швидкодії ШОЗ (підрозд. 4.6), включаючи порівняння C++ реалізацій conv_RGB_NEON(...) та cv::filter2D, аналіз швидкодії Python-інтерфейсу та операцій масштабування ЦЗ. Наприкінці розділу сформульовано висновки за результатами експериментальних досліджень (підрозд. 4.7), що підтверджують ефективність запропонованої ІТ.

4.1 Опис експерименту

Метою експериментального дослідження є оцінка ефективності розробленої ІТ обробки ЦЗ із застосуванням ШОЗ на платформі ARM64. Зокрема, експеримент спрямований на порівняльний аналіз швидкодії розробленого ЯІТ conv_RGB_NEON(...) з традиційним підходом, реалізованим у бібліотеці OpenCV через функцію cv::filter2D(...).

Експериментальне дослідження побудовано на основі систематичного підходу, що включає:

Комплексне дослідження швидкодії для різних комбінацій параметрів обробки: розміри ядра згортки: від 2×2 до 15×15 (14 варіацій); Роздільна зда-

тність ЦЗ: від 256×144 до 3840×2160 (8 варіацій); Тип кольорової моделі - RGB (24-бітний колір).

Стандартизовані умови тестування: однакові апаратні платформи на базі архітектури ARM64; ідентичні вхідні дані для всіх алгоритмів; автоматизовані вимірювання показників швидкодії (FPS); Збір статистично значущої вибірки даних (не менше 150 замірів для кожної комбінації параметрів).

Багатофакторний аналіз швидкодії, що включає: залежність швидкодії від розміру ядра згортки; залежність швидкодії від роздільної здатності зображення; порівняльний аналіз ефективності різних реалізацій алгоритму згортки [93].

Основним показником ефективності обрано кількість оброблених кадрів за секунду (FPS), що дозволяє безпосередньо оцінити швидкодію алгоритму в умовах, наближених до реальних застосувань, таких як обробка відеопотоків у режимі реального часу.

Для забезпечення достовірності результатів та виключення впливу сторонніх факторів, таких як фонові процеси операційної системи або флуктуації швидкодії процесора, застосовано методику статистичної обробки результатів з використанням медіанних значень.

4.2 Прилади та апаратне забезпечення

Для проведення експерименту з дослідження швидкодії реалізації алгоритму ВОЗ під час обробки RGB ЦЗ на архітектурі ARM64, за умови 16-бітних обчислень, було обрано дві популярні розробницькі плати (SBC): Orangepi 5 Pro [94] та Raspberry Pi 4 [95, 96]. Вибір цих пристроїв зумовлений їхнім широким поширенням у сфері вбудованих систем та IoT, а також різноманітністю їхніх процесорних архітектур, що дозволяє провести порівняльний аналіз ефективності оптимізацій на різних платформах. Наприклад, Orangepi 5 Pro з вбудованим у його SoC NPU може бути використаний як для створення TV-приставок, так і для навчання CNN (пристрій широкого спектра застосування). На противагу також було використано Raspberry Pi 4 через його поширеність та доступність за критеріями ціни й якості.

Таблиця 4.1 — Апаратне забезпечення для збору експериментальних даних

Назва SBC	**Orangepi 5 pro**	**Raspberry Pi 4**
CPU:	Rockchip RK3588S octa-core (4x Cortex-A76 + 4x Cortex-A55)	Broadcom BCM2711 quad-core Cortex-A72
Особливості CPU:	ARMv8.2-A NPU: RKNN зі швидкодією 6 TOPS	ARMv8-A Без NPU
RAM	16 GB - LPDDR4X	4 GB - LPDDR4
Назва GPU	Mali-G610 MP4	VideoCore VI
ROM	Kingston 64GB microSDXC class 10 UHS-I U3 Canvas Go	
OS	Ubuntu 24.04	Raspberry Pi OS (x64). Основано на Debian 2024-04-09.

У (табл. 4.1) наведено обладнання/SBC та їхні основні характеристики. Усі SBC та їхні SoC базуються на архітектурі ARMv8-A ЦП, що підтримує SIMD-інструкції NEON64 [91].

4.3 Програмне налаштування апаратного забезпечення

На обох пристроях було встановлено Debian-подібні ОС, що забезпечує однорідне середовище для проведення експериментів. Вибір ОС цього сімейства обумовлений наступними факторами: 1. Поширеність на різноманітних апаратних платформах/SBC; 2. Наявність актуальних версій компіляторів та бібліотек, необхідних для реалізації та оптимізації алгоритмів обробки ЦЗ; 3. Наявність засобів розробки та профілювання; 4. Можливість отримати максимальну швидкодію ОС за рахунок її тонкого налаштування.

Особливості налаштування системи для експерименту: 1. Відключення GUI ОС для мінімізації фонового навантаження на систему; 2. Налаштування параметрів CPU governor для забезпечення максимальної швидкодії процесора під час тестів; 3. Відключення непотрібних системних служб для зменшення споживання ресурсів.

Такий підхід до вибору ОС та налаштування SBC дозволяє провести комплексний аналіз ефективності розробленого ЯІТ на розповсюджених архітектурах ARM64 SBC, забезпечуючи репрезентативність результатів.

4.4 Загальний опис програмного забезпечення (ОС тощо)

Для проведення дослідження та розробки оптимізованих алгоритмів обробки ЦЗ було використано великий спектр ПЗ, яке відігравало важливу роль у різних аспектах, але надалі буде описано/згадано основні.

4.4.1 Середовище розробки та управління проектом

Для розробки даної ІТ було використано IDE Visual Studio Code (далі VSCode) та плагіни, що забезпечують отримання SSH доступу через локальну мережу. Було використано наступні плагіни: SSH; C/C++ від Microsoft; сукупність плагінів для CMake (збірка та конфігурування проекту ІТ).

Поточне налаштування VSCode надало можливість кросархітектурної взаємодії ОС Windows та Ubuntu при розробці ІТ.

4.4.2 Компілятори, інтерпретатор та середовище віртуалізації та емуляції.

Для формування/реалізації даної ІТ було використано такі компілятори як GCC [39] та [97]. GCC є основним компілятором для збірки проєкту на вищезгаданих (табл. 4.1) SBC. Clang використовувався для крос-компіляції проекту ІТ під GNU ARM64 ОС на GNU x86 ОС (Ubuntu 24.04) і подальшого запуску/перевірки результату на VM QEMU [98]. Обидва компілятори підтримують SIMD-інструкції NEON64.

Інтерпретатор Python використовувався для розробки скриптів аналізу швидкодії ЯІТ, генерації та візуалізації результатів досліджень, а також для розробки демонстраційної частини ІТ.

Інтерпретатор bash/shell використовувався для запуску і накопичення експериментальних даних: автоматизації процесу запуску тестів; забезпечення послідовного виконання серії експериментів з різними вхідними параметрами; автоматизації процесу збору та первинної обробки результатів, включаючи запис показників FPS у файли для подальшого аналізу.

Для розробки даної ІТ було використано також такі інструменти віртуалізації ОС як VMware та QEMU. VMware використовувалося для створення ізольованого середовища розробки та тестування, а саме віртуальної машини з ОС Ubuntu 24.04 (x64) + компілятори та інтерпретатори. QEMU використовувався для емуляції ARM архітектури (і не тільки) на x86 ЦП. QEMU дозволяє проводити попереднє тестування та налагодження коду перед розгортанням на вищезгаданих (табл. 4.1) SBC. Отже, комбінація VMware + Ubuntu 24.04 (x64) + Clang крос-компілятор + QEMU використовувалась для перевірки, тестування та емуляції роботи ПЗ/ІТ на GNU/Linux ARM64 ОС, без запуску результату на SBC.

Використання цього комплексного набору програмних інструментів дозволило ефективно розробити, оптимізувати та протестувати алгоритми обробки ЦЗ на різних апаратних платформах.

4.4.3 Інструменти для оптимізації ядра інформаційної технології

Основним інструментом інтеграції NEON64 інструкцій в C/C++ код програми є Inline Assembly вставки. Усі ключові етапи/блоки обчислення ВОЗ/ЯІТ для цілочисельних ЯЗ, що зображено на (рис. А.5), було реалізовано за допомогою Inline Assembly вставок з урахуванням/використанням NEON64 інструкцій. Також слід зазначити, що програмно/практично було реалізовано алгоритм обчислення ШОЗ для ЯЗ (рис. А.6), що враховує кількість елементів рядка більшу ніж 16, що дозволило забезпечити гранулювання C/C++ та Inline Assembly NEON64 інструкцій та отримати підвищення швидкодії для ШОЗ при обробці RGB-ЦЗ з використанням 16-бітних розріджених ЯЗ.

Також було використано такі інструменти профілювання програмного коду як Callgrind з набору інструментів програмного пакету Valgrind. Valgrind дозволяє, хоч і не в повній мірі, здійснити комплексний аналіз швидкодії будь-якого коду, що може бути запущеним у середовищі його віртуальної машини. Також за допомогою інструменту massif (heap profiler) [48] було частково виконано виявлення потенційних проблем в оптимізованому коді, таких як ”витік пам’яті”(memory leak).

4.5 Загальний опис тестування швидкодії інформаційної технології за допомогою Shell-скриптів.

Опис проведення експериментів передусім вимагає зазначити, що всі тести на всіх приладах були запущені за однакових умов, тобто всі скрипти та їхні команди є ідентичними. Єдина відмінність, яку можна винести на розгляд, це використання на одному приладі Ubuntu 24.04, а на другому — Raspberry Pi OS (на базі Debian). Зокрема, це єдине, чим програмно відрізняються прилади, на яких було проведено дослідження (без урахування: драйверів Linux і конфігурації її налаштувань (dtb і dtbo, SELinux тощо); процесу завантаження ядра Linux тощо (кожен виробник SBC розробляє свої варіанти драйверів та процес завантаження ядра Linux)). Тому також слід зазначити, що ця різниця в ОС може мати певний вплив на результати через можливі оптимізації як ядра ОС, так і системних бібліотек, які були задіяні. Але ІТ розроблена за допомогою одного й того ж коду на C/C++ та Python, і, оскільки ЯІТ/ВОЗ реалізовано з додаванням Inline Assembly вставок NEON64 інструкцій, це нівелює вплив різних версій компілятора та його рівня АВ коду. Хай там як, ПЗ лишається незмінним для цих 2 SBC.

Для створення адаптивного скрипта/програми тестування накопичення даних щодо показників швидкодії для cv::filter2D(…) та conv_RGB_NEON(…) було використано систему прапорів CLI-інтерфейсу, яка надає можливість гнучко налаштовувати параметри обробки відеопотоку з різними розмірами ЯЗ, що дозволяє обробляти як відеопотік з камери, так і відеопотік з файлу (підрозд. 4.6). Програмно цей підхід реалізовано за допомогою бібліотеки OpenCV та її компонентів, що дозволяють ефективно працювати з різними джерелами відеопотоку. Вибір OpenCV обумовлений її широкими можливостями та систематичною оптимізацією її компонентів, зокрема для роботи з відео та зображеннями.

Також було використано багатопоточні обчислення у частині C/C++ коду, що реалізує зчитування фреймів ЦЗ з відеопотоку/файлу (далі просто потоку, незалежно від того, чи це відео, чи камера). Причиною такого рішення став парадоксальний факт: час обробки ЦЗ за допомогою ШОЗ виявився значно меншим, ніж час процесу зчитування нового фрейму з потоку та передачі його в чергу обробки. Тобто час обробки фрейму в рази менший, ніж час його зчитування (за певних умов). Такий парадокс зумовлений тим, що швидкодія оптимізованої ВОЗ/ЯІТ дозволяє обробляти ЦЗ швидше, ніж відбувається його зчитування з

відеопотоку. При цьому важливо зазначити, що така поведінка спостерігається під час обробки ЦЗ із шириною від 256 до 640 пікселів. Таким чином, багатопоточність дозволяє ефективно розподілити навантаження між процесами зчитування та обробки [99, 100].

4.5.1 Базовий алгоритм тестування швидкодії.

Тестові скрипти, що використовують функцію conv_RGB_NEON(…), написану на мові C/C++, використовують наступні ключові особливості:

1. **Оптимізація кількості потоків зчитування фреймів ЦЗ:** Експериментально було встановлено, що кількість потоків зчитування ЦЗ з відеопотоку суттєво впливає на швидкодію. Оптимальна кількість потоків для більшості сценаріїв виявилася рівною 3. Тому, якщо користувач явно не вказує інше значення через відповідний прапор, програма за замовчуванням використовує 3 потоки. Це можна виразити математично як:

$$N_{threads} = \max(3, N_{user_specified}) \tag{4.1}$$

де $N_{threads}$ - фактична кількість потоків, а $N_{user_specified}$ - кількість потоків, вказана користувачем (якщо вказано).

У випадку виконання тестового скрипту для Python програми кількість потоків постійна і дорівнює 1.

2. **Динамічне управління пам'яттю:** Для запобігання переповнення оперативної пам'яті (ОЗП) встановлено адаптивну межу для вектора фреймів. Цей вектор складається з черг, кількість яких дорівнює кількості потоків. Загальна/максимальна кількість фреймів у чергах вектора розраховується за формулою:

$$M_{max} = 0{,}7 \times RAM_{total} \times \frac{1}{S_{frame}} \tag{4.2}$$

де M_{max} - максимальна кількість фреймів у чергах вектора, RAM_{total} - загальний обсяг вільного ОЗП (в байтах), S_{frame} - розмір одного фрейму (в байтах). Коефіцієнт 0,7 відповідає 70% використання ОЗП і може бути адаптований залежно від конкретних умов системи (або в коді ПП).

3. **Балансування обробки та збереження даних:** Виділена пам'ять (70% ОЗП) розподіляється між двома частинами: для оброблених фреймів та

для фреймів, що очікують обробки. Це можна представити як:

$$M_{processed} = M_{to_be_processed} = \left\lfloor \frac{M_{max}}{2} + 0{,}5 \right\rfloor \tag{4.3}$$

Коли $M_{processed}$ досягає свого ліміту, відбувається збереження результату у відеофайл, паралельно з продовженням завантаження нових фреймів у $M_{to_be_processed}$. Цей механізм забезпечує ефективне балансування навантаження системи. А $M_{to_be_processed}$ розташовується у чергах вектора зчитування, кількість фреймів у кожній черзі M_{seq} можна розрахувати таким чином:

$$M_{seq} = \left\lfloor \frac{M_{to_be_processed}}{N_{threads}} \right\rfloor \tag{4.4}$$

4. **Забезпечення цілісності відеопотоку:** Кожен фрейм у черзі супроводжується порядковим номером, що дозволяє зберегти послідовність кадрів при обробці та збереженні. Математично це можна представити як упорядковану послідовність:

$$F = \{(f_1, 1), (f_2, 2), ..., (f_n, n)\} \tag{4.5}$$

де f_i - і-й фрейм, а друге число в кожній парі - його порядковий номер.

Важливо зазначити, що збереження результату у відеофайл може знизити загальну швидкодію системи. Альтернативний підхід з використанням ОЗП для тимчасового зберігання, хоча і більш ефективний з точки зору швидкодії, створює додаткові складнощі в управлінні пам’яттю та подальшому об’єднанні фрагментів відео.

Ефективність обробки можна оцінити за допомогою метрики кадрів на секунду (FPS):

$$FPS = \frac{N_{processed_frames}}{T_{total}} \tag{4.6}$$

де $N_{processed_frames}$ - кількість оброблених фреймів, а T_{total} - загальний час обробки (зазвичай не більше 1,5 с).

Дане дослідження фокусується на оптимізації саме процесу обробки ЦЗ, залишаючи оптимізацію збереження результатів як потенційний напрямок для подальших досліджень. Такий підхід дозволяє точніше оцінити ефективність розроблених алгоритмів ОЗ без впливу сторонніх факторів, пов’язаних з опера-

ціями введення-виведення. Отже, якщо відомостей про збереження результату обробки у відеофайл не було надано, то усі оброблені ЦЗ знищуються, коли $M_{processed}$ досягає свого ліміту.

Скрипт проводить тестування, фіксуючи визначену кількість показників FPS, наприклад 150, але не більше обраного значення (цей параметр опціонально коригується) для кожного набору параметрів. Після досягнення необхідної кількості накопичених показників FPS обчислення зупиняються, і скрипт переходить до наступної конфігурації, збільшуючи розмір ЯЗ та коригуючи інші параметри.

Розміри ЯЗ змінюються з кроком 1, починаючи з 2×2 до 15×15, що повністю відповідає виразу (А.3):

$$flt_rows = flt_cols = \overline{2, 15} \tag{4.7}$$

де flt_rows та flt_cols - кількість рядків та стовпців ЯЗ відповідно.

Після досягнення максимального розміру ЯЗ (15×15) скрипт переходить до обробки наступного відео з іншою роздільною здатністю. Весь описаний процес повторюється для кожного відеофайлу. У дослідженні використовувалися відеофайли з наступними роздільними здатностями, що відображені в назвах відеофайлів.

Результатом такого тестування є матриця розміром 8×14, де кожен елемент представляє порівняння швидкодії запропонованого алгоритму conv_RGB_NEON(…) зі стандартним cv::filter2D(...).

Ця матриця може бути представлена у декількох варіаціях:

1. Матриці медіанних показників FPS для кожного алгоритму:
   $F_{conv_RGB_NEON} = \{f_{ij}^{conv_RGB_NEON}\}_{8\times14}$ та $F_{filter2D} = \{f_{ij}^{filter2D}\}_{8\times14}$
2. Матриця порівняння швидкодії:
   $S = \{s_{ij}\}_{8\times14}$, де $s_{ij} = \frac{FPS_{conv_RGB_NEON}}{FPS_{filter2D}}$
3. Матриця різниці показників швидкодії:
   $D = \{d_{ij}\}_{8\times14}$, де $d_{ij} = FPS_{conv_RGB_NEON} - FPS_{filter2D}$
   де і відповідає роздільній здатності відео, а j - розміру ЯЗ.

Для статистичної обробки результатів можна використовувати як середнє значення, так і медіану:

$$FPS_{avg} = \frac{1}{n}\sum_{i=1}^{n} FPS_i \tag{4.8}$$

$$FPS_{median} = \begin{cases} FPS_{\frac{n+1}{2}}, & \text{якщо n не парне} \\ \frac{FPS_{\frac{n}{2}} + FPS_{\frac{n}{2}+1}}{2}, & \text{якщо n парне} \end{cases} \quad (4.9)$$

де n - кількість замірів FPS (у нашому випадку не менше 150 для кожної комбінації параметрів).

Таблиця 4.2 — Розміри ЦЗ/фрейму та назв відеофайлів для проведення експериментів

№	Назва відеофайлу
1	256x144_30.webm
2	426x240_30.webm
3	640x360_30.webm
4	854x480_30.webm
5	1280x720_30.webm
6	1920x1080_60.webm
7	2560x1440_60.webm
8	3840x2160_60.webm

Використання медіани може бути більш стійким до викидів, які можуть виникнути через тимчасові флуктуації в швидкодії системи [25, 57, 86].

4.5.2 Етапи тестування ядра інформаційної технології у Shell.

Розроблена ІТ складається з двох ключових компонентів: програмного модуля на мові Python та бібліотеки на мові C++, що представляє набір функцій ЯІТ (підрозд. 3.3.1, 3.4.2). Як було зазначено (підрозд. 3.5.1) - обидва компоненти використовують ВОЗ.

Тести та їх структура реалізовані з використанням мови bash/shell та представлені функцією run_test(…), що розташована у файлах run.test.sh та run.test.py.sh. Основне призначення даної функції полягає у проведенні комплексного порівняльного аналізу швидкодії двох реалізацій ВОЗ:

1. cv::filter2D(…) – ШОЗ з бібліотеки OpenCV;
2. conv_RGB_NEON(…) – ШОЗ ІТ/ЯІТ.

Обидві функції створені під час компіляції C++ коду за допомогою системи збірки CMake. Варто зазначити, що обидві реалізації ШОЗ (conv_RGB_NEON(…) та cv::filter2D(…)) використовують інструкції NEON64. Ключова відмінність між реалізаціями полягає у форматі представлення даних ЯЗ та специфічних алгоритмічних оптимізаціях. Функція run_test() взаємодіє з цими двома функціями або через виконуваний файл безпосередньо, або через Python код/скрипт (див. підрозд. 3.6.3).

Важливо підкреслити, що процес формування ЯЗ виконується перед початком обробки відеопотоку, що дозволяє виключити вплив підготовчих операцій на оцінку швидкодії. Такий підхід забезпечує отримання об'єктивних порівняльних характеристик дослідження.

Розроблені тестові скрипти та середовище забезпечують автоматичне виконання серії експериментів з 14 різними розмірами ЯЗ (від 2×2 до 15×15) на 8 відеофайлах різної роздільної здатності (від 256×144 до 3840×2160), що дозволило отримати репрезентативний масив даних зі 112 комбінацій параметрів для ґрунтовного аналізу переваг розробленої ІТ у різних сценаріях застосування.

Опис тестування C/C++ функцій cv::filter2D(…) та conv_RGB_NEON(…)

Отже, саме тестування швидкодії ІТ у вигляді/формі C++ програми, складається з декількох ключових етапів:

1. Запуск функції тестування run_test(), з файлу run.test.sh, що ініціалізує програму з відповідними параметрами через систему прапорців. Ці прапорці включають:
   (а) **-k**: визначає або розмір ядра згортки, або шлях до текстового файлу з ядром та дільником. Це дозволяє гнучко налаштовувати параметри ШОЗ для різних сценаріїв обробки. А оскільки stdio дозволяє передавати числові значення у формі тексту, то було додано можливість аналізувати, чи передається розмір ЯЗ, чи шлях до файлу;
   (б) **-f**: вказує функцію, що буде використовуватися для фільтрації потоку. Основні опції включають open_cv (стандартна реалізація ШОЗ OpenCV cv::filter2D(…)) та neon, що викликає ШОЗ - ЯІТ;
   (в) **-g**: визначає градієнт потоку/ЦЗ, що буде фільтровано. Може бути у двох станах: rgb (повнокольорове зображення) або gray (градієнт сірого). Це дозволяє оцінити ефективність алгоритмів для різних

типів вхідних даних;

(г) **-i**: вказує шлях до джерела відеопотоку, будь то файл чи пристрій захоплення відео (наприклад, /dev/video0 для веб-камери);

(д) **-t**: кількісний показник, що задає число потоків для зчитування фреймів ЦЗ з потоку. Цей параметр дозволяє оцінити вплив багатопоточності на загальну швидкодію системи, в контексті кількості потоків зчитування відеопотоку.

Інші допоміжні прапорці, які, хоча і не є ключовими для даного дослідження, забезпечують додаткову гнучкість та можливості налаштування, як пропуск фрейму **-s** і тп.

2. Важливою особливістю розробленої програми ІТ є її здатність постійно інформувати користувача про поточну швидкодію через показник FPS. Це відбувається з частотою приблизно раз на секунду (або рідше ніж раз на 1 сек (мається на увазі 1 раз на 2 сек і рідше), залежно від поточної апаратної платформи та розміру ЦЗ), що дозволяє в реальному часі відстежувати сумарні зміни швидкодії розробленого ПЗ. Така функціональність була реалізована для дослідження впливу різних факторів, включаючи операції зберігання оброблених фреймів, на загальну швидкодію програмного забезпечення.
3. У межах цього дослідження було прийнято рішення не розглядати детально вплив збереження результатів обробки ЦЗ у відеофайл. Збереження результатів обробки ЦЗ у відеофайл має свої особливості та принципи/підходи до їхнього нівелювання, тому його вплив на загальну швидкодію ІТ не було розглянуто. Основна причина в тому, що процес збереження відео вводить додаткову складність у вигляді взаємодії між драйвером ОС, що записує результат (набір байтів) у відеофайл, та кодеком, який формує кодовану послідовність фреймів для передачі цьому драйверу тощо. Аналіз цієї взаємодії становить собою окреме комплексне дослідження з багатьма невідомими факторами, що значно ускладнило б інтерпретацію результатів основного експерименту. Більше того, на практиці часто не є ефективним збереження усіх оброблених файлів, натомість логічним є зберігати необхідну кількість в оперативній пам’яті й після досягнення потрібного обсягу зберігати все одразу. Тому, для забезпечення чистоти експерименту та фокусуванні на аналізі швидкодії обробки ЦЗ даною ІТ,

було прийнято рішення не використовувати прапорець **-o**, який відповідає за зберігання результуючого відео. Це стосується всіх варіацій запуску ПЗ/ІТ у рамках даного дослідження. Таким чином, увагу було зосереджено виключно на ефективності реалізації алгоритму обробки ЦЗ (що виражається в показнику підвищення швидкодії), мінімізуючи вплив сторонніх факторів на результати експерименту.

Опис тестування Python обгортки C/C++ функцій cv::filter2D(…) та conv_RGB_NEON(…): Тестування швидкодії ІТ у формі Python-скрипта реалізовано з використанням мови shell через файл run.test.py.sh, в якому запускається функція run_test(). Для реалізації ШОЗ в задачах збільшення, зменшення та розмиття RGB ЦЗ застосовано бібліотеку ctypes, що забезпечує міжмовну взаємодію мови Python та бібліотеки, написаної на C/C++.

Система вимірювання швидкодії: Для об'єктивної оцінки швидкодії розроблено клас FpsCounter, що обчислює кількість оброблених кадрів за секунду на основі часу виконання операцій. Даний клас використовує функції cv2.getTickCount() та time.time() для точного вимірювання часових інтервалів. Результати вимірювань оновлюються з частотою приблизно раз на секунду, що дозволяє спостерігати динаміку швидкодії в реальному часі.

Візуалізація та аналіз результатів: Результати тестування швидкодії систематизуються та візуалізуються у вигляді матриць розміром 8×14, що відображають ефективність розробленого алгоритму conv_RGB_NEON(…) у порівнянні з стандартною реалізацією cv::filter2D(...). Матриці формуються для різних комбінацій роздільної здатності відео та розмірів ЯЗ, що дозволяє комплексно оцінити ефективність запропонованої ВОЗ/ЯІТ в різних умовах експлуатації. Детальний аналіз отриманих результатів представлено в (підрозд. 4.6).

4.5.3 Методика тестування швидкодії інформаційної технології

Для забезпечення системного підходу до тестування швидкодії розробленої ІТ, було створено модульну структуру тестового середовища, представлену у вигляді ієрархії процедур. UML Діаграми активності, розроблені згідно зі стандартом SDL (Specification and Description Language), відображають логічну структуру та послідовність виконання операцій в процесі тестування. Нижче наведено детальний аналіз кожної з основних процедур:

1. Загальна процедура тестування швидкодії:

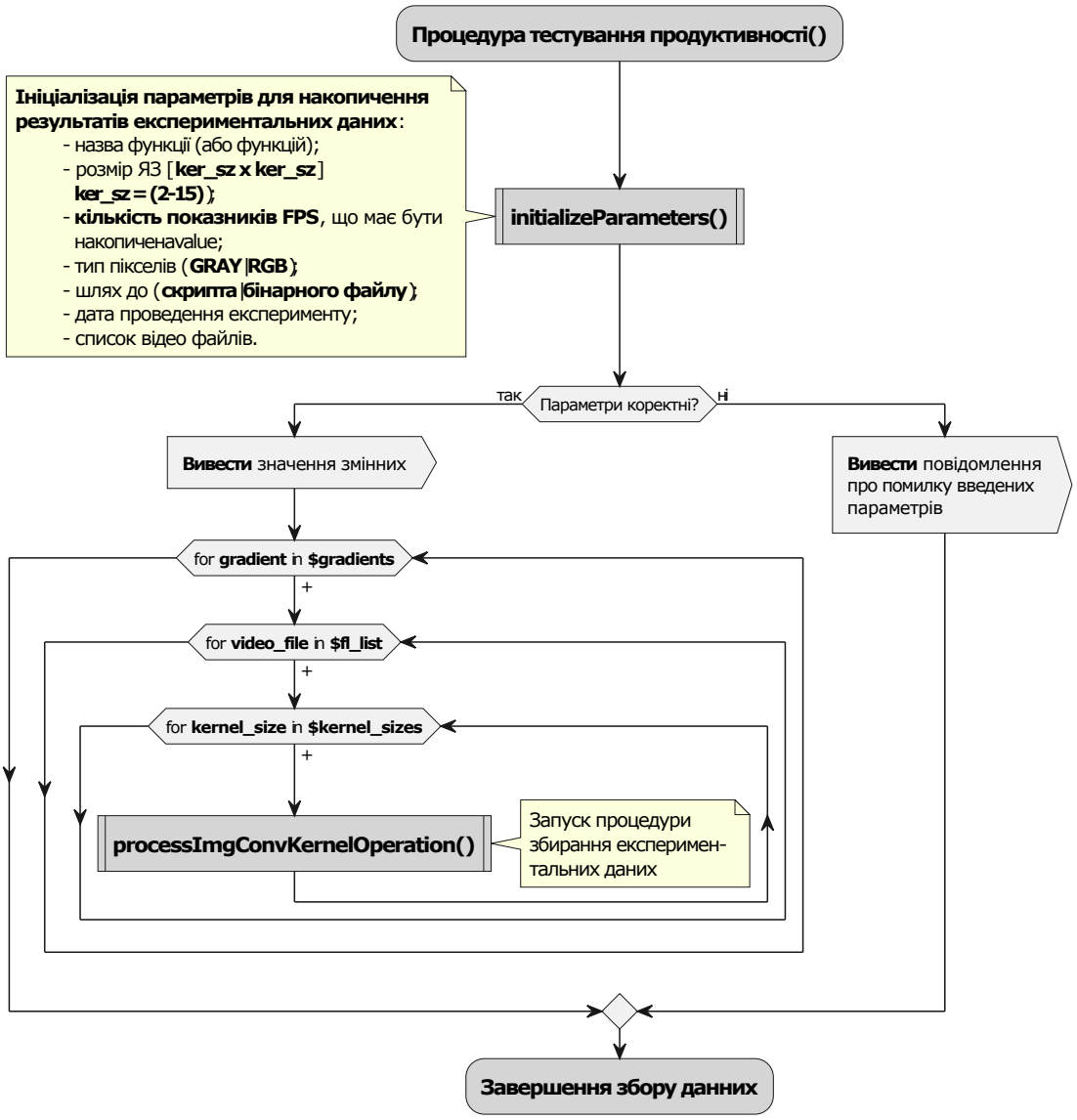


Рисунок 4.1 — UML/SDL Діаграма активності тестування швидкодії.

Основу тестового середовища складає головна процедура (рис. 4.1), яка реалізує комплексний підхід до накопичення експериментальних даних щодо швидкодії алгоритмів згортки. Процедура включає такі основні етапи:

1. Ініціалізація параметрів (initializeParameters()) – встановлення вхідних параметрів для тестування, включаючи назву функції, розміри ЯЗ, кількість показників FPS для збору, тип пікселів, шлях до скрипта/бінарного файлу, дату проведення експерименту та список відеофайлів для обробки;
2. Виведення значень параметрів – відображення ініціалізованих параметрів для контролю коректності налаштування експерименту.
3. Чотирирівнева система вкладених циклів: а) Цикл по градієнтах (gradients)

– дозволяє тестувати різні колірні простори (RGB/GRAY) (у випадку даного дослідження використовується RGB); б) Цикл по відеофайлах (fl_list) – забезпечує тестування на різних відеофайлах; в) Цикл по розмірах ЯЗ (kernel_sizes) – дозволяє оцінити залежність швидкодії від розміру ЯЗ.

4. Виконання тестової процедури – для кожної комбінації параметрів викликається процедура processImgConvKernelOperation(), що відповідає за накопичення даних про швидкодію.

Така багаторівнева структура забезпечує повний перебір усіх можливих комбінацій параметрів, що дозволяє провести всебічне дослідження швидкодії розробленої інформаційної технології.

2. Процедура обробки ОЗ - processImgConvKernelOperation():

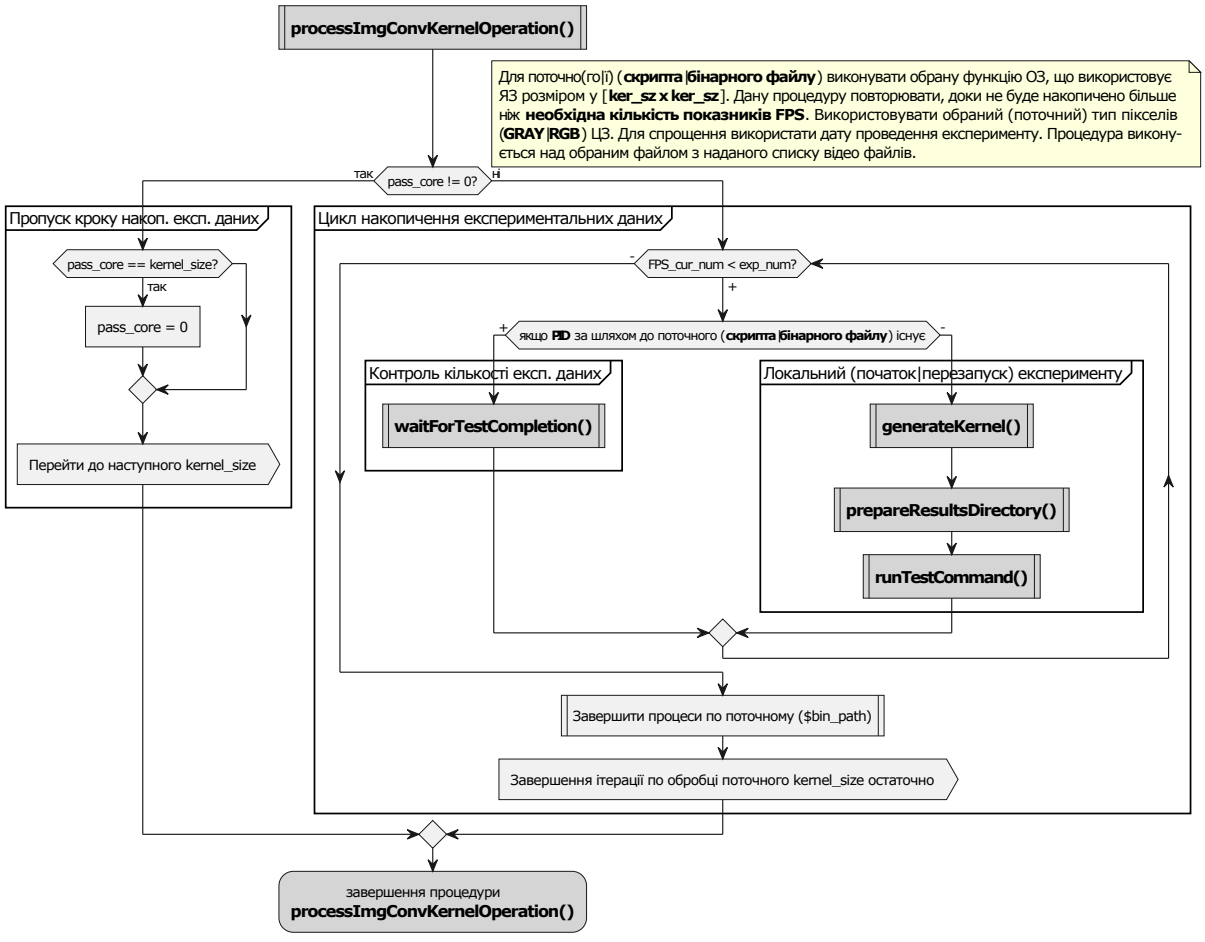


Рисунок 4.2 — UML/SDL Діаграма алгоритму обробки статистичних даних.

Процедура processImgConvKernelOperation() (рис. 4.2) є ключовою складовою тестового середовища і відповідає за формування експериментальних даних для кожної комбінації параметрів. Процедура включає два основні режими роботи:

1. Режим пропуску кроку – активується, коли параметр pass_core не дорівнює нулю. Цей режим дозволяє пропустити тестування для певних розмірів ЯЗ, що забезпечує гнучкість у налаштуванні експериментів.
2. Режим накопичення експериментальних даних – основний режим, що реалізує циклічний процес збору даних про швидкодію. Процес включає:
3. Перевірку наявності активного процесу за заданим шляхом та PID.
4. Якщо процес PID відсутній – запуск нового експерименту через послідовність процедур: generateKernel(), prepareResultsDirectory(), runTestCommand(). Це означає що, або даний процес природньо завершив виконання, або це перший раз коли він буде запущений.
5. Якщо процес активний – відбувається контроль кількості накопичених показників FPS через процедуру waitForTestCompletion().
6. Завершення циклу після досягнення необхідної кількості показників FPS.

Такий підхід забезпечує автоматизоване накопичення статистично значущої вибірки даних для кожної комбінації параметрів тестування.

Далі буде розглянуто допоміжні процедури тестового середовища.

3. Процедура генерації ядра згортки - generateKernel():

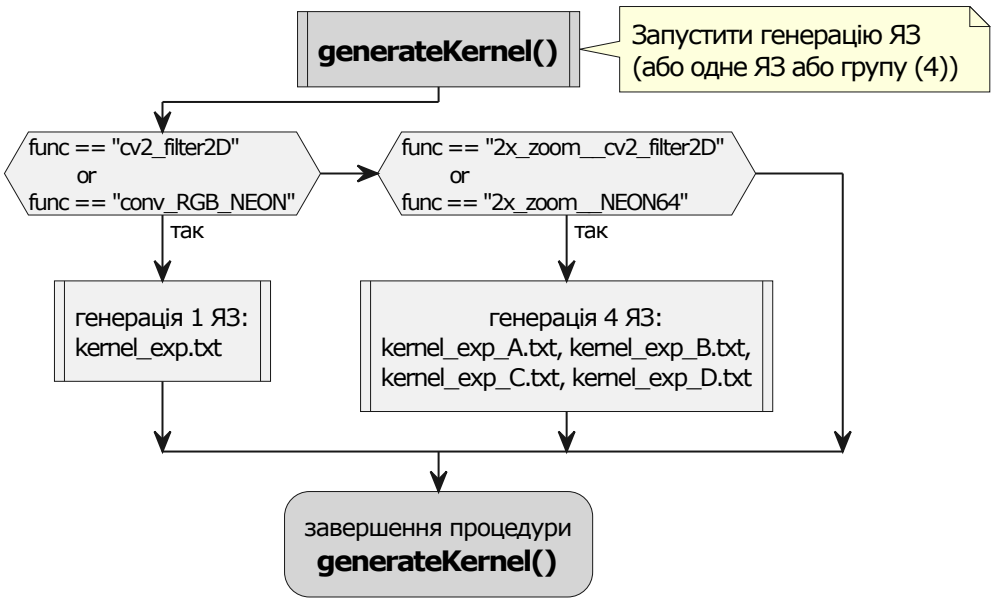


Рисунок 4.3 — UML/SDL Діаграма процедури генерації ЯЗ для різних режимів роботи.

Процедура generateKernel() (рис. 4.3) відповідає за формування файлів з параметрами ЯЗ.

Залежно від типу функції обробки, процедура реалізує два основні сцена-

рії:

1. Для функцій cv2_filter2D(…) або conv_RGB_NEON(…) генерується один файл kernel_exp.txt, що містить параметри ядра згортки.
2. Для функцій zoomUP_2__cv2_filter2D(…) або zoomUP_2__NEON64(…) генеруються чотири файли - kernel_exp_A.txt - kernel_exp_D.txt, що відповідають різним ЯЗ для операції масштабування ЦЗ (підрозд. 2.2).

Така дворівнева структура забезпечує гнучкість у тестуванні як базових ОЗ ЦЗ, так і комплексних операцій, таких як масштабування ЦЗ.

4. Процедура підготовки директорії результатів досліджень - prepare ResultsDirectory():

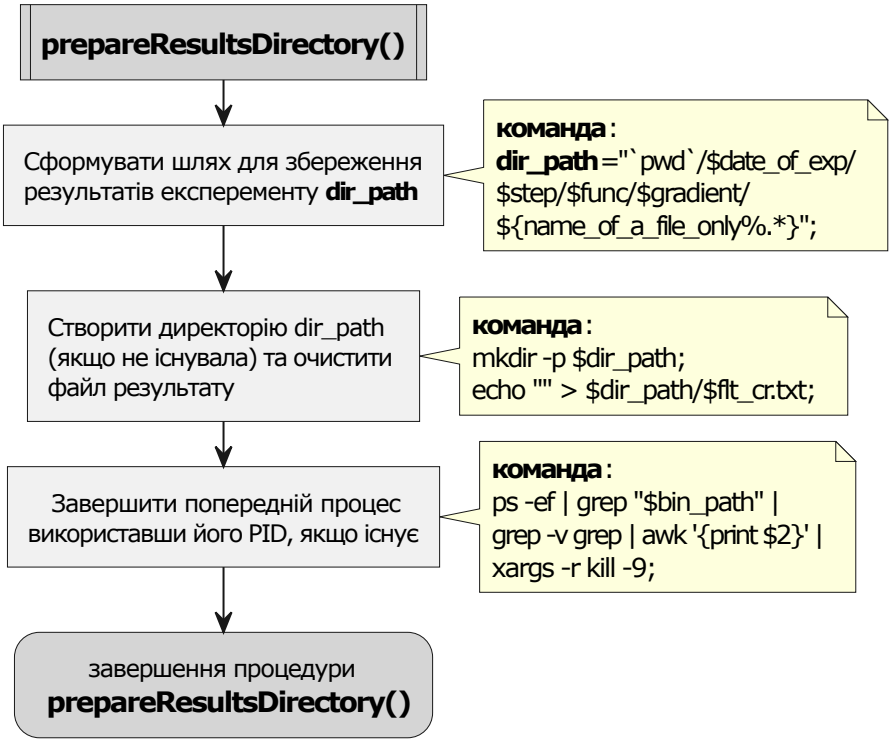


Рисунок 4.4 — UML/SDL Діаграма процедури підготовки/створення файлів/каталогів для збереження експериментальних даних.

Процедура prepareResultsDirectory() (рис. 4.4) забезпечує організацію ієрархічної структури директорій для зберігання результатів експериментів.

Основні етапи процедури включають:

1. Формування шляху до директорії результатів (dir_path) на основі параметрів експерименту, включаючи дату, крок, назву функції, тип градієнта та назву відеофайлу.
2. Створення відповідної директорії (якщо вона не існує) та очищення і створення файлу результатів.

3. Завершення попередніх процесів за заданим шляхом для уникнення конфліктів між різними тестовими запусками.

Такий підхід забезпечує систематизоване зберігання результатів, що спрощує їх подальший аналіз та інтерпретацію.

4. Процедура запуску тестової команди - runTestCommand():

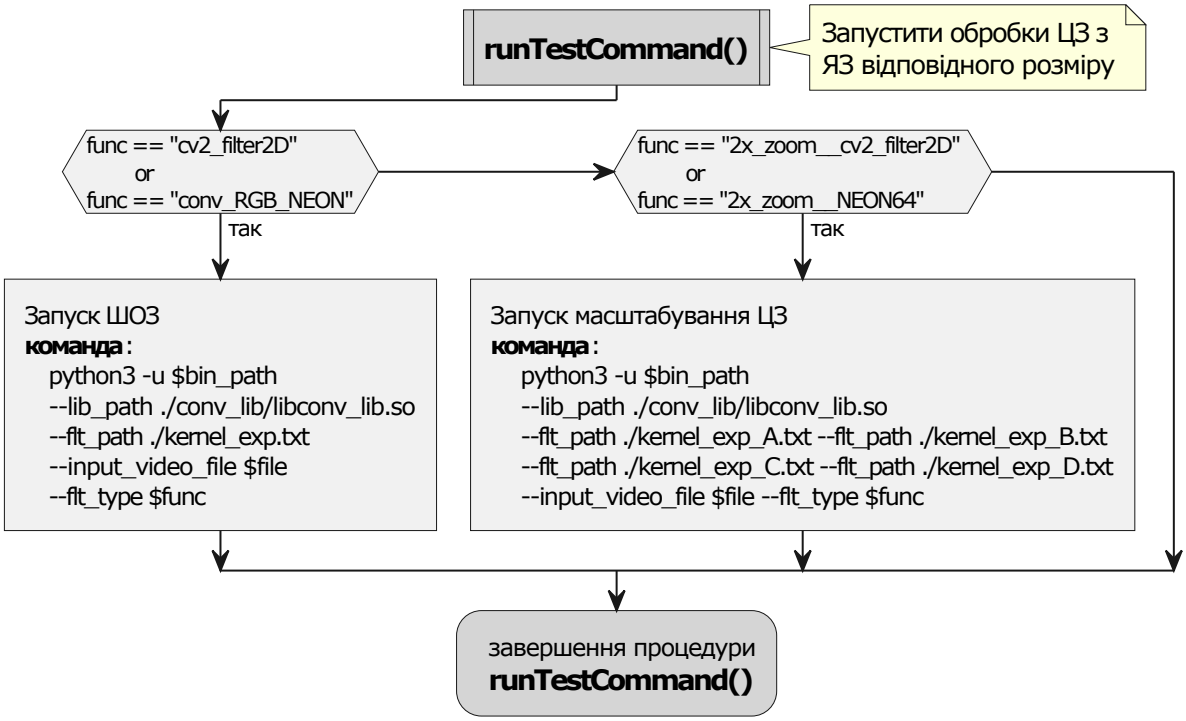


Рисунок 4.5 — UML/SDL Діаграма процедури запуску тесту швидкодії ОЗ для ЦЗ з відповідними параметрами.

Процедура runTestCommand() (рис. 4.5) відповідає за безпосередній запуск тестового програмного забезпечення з відповідними параметрами.

Залежно від типу функції, процедура реалізує два сценарії:

1. Для функцій cv2_filter2D(…) або conv_RGB_NEON(…) запускається команда з одним файлом параметрів ЯЗ.
2. Для функцій zoomUP_2__cv2_filter2D(…) або zoomUP_2__NEON64(…) запускається команда з чотирма файлами параметрів ЯЗ.

При запуску тестових команд використовується параметр Python (-u - unbuffered), що забезпечує виведення даних без буферизації. Це рішення критично важливе для тестування, оскільки забезпечує: миттєве відображення показників FPS у вихідному потоці; мінімізацію втрат даних при неочікуваному завершенні процесу; збереження хронологічної послідовності виведених даних. Такий підхід дозволяє точно контролювати процес накопичення експеримен-

тальних даних та забезпечує достовірність результатів тестування.

5. Процедура очікування завершення тесту - waitForTestCompletion():

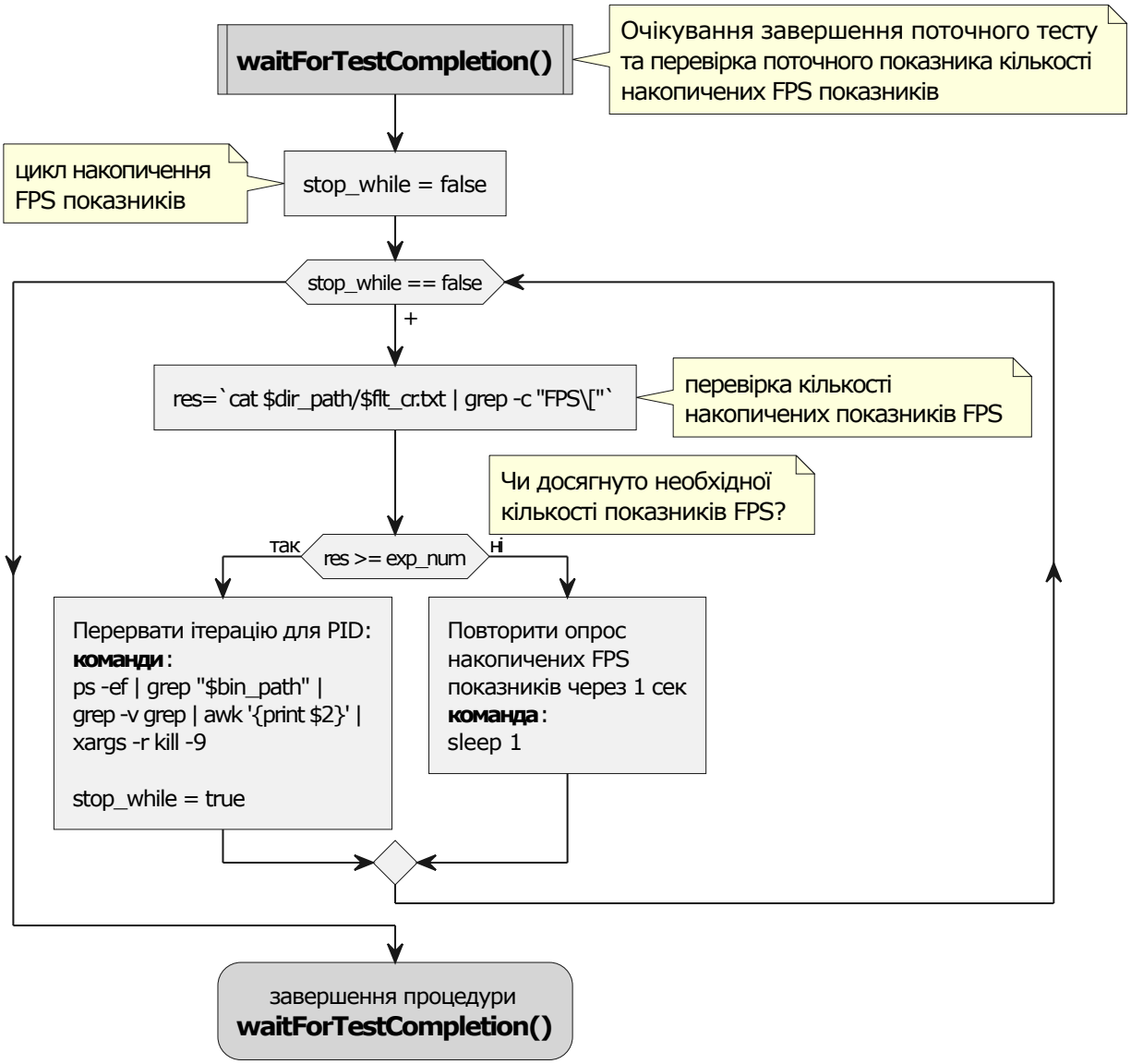


Рисунок 4.6 — UML/SDL Діаграма процедури очікування завершення накопичення тестових даних.

Процедура waitForTestCompletion() (рис. 4.6) реалізує механізм контролю за процесом накопичення даних. Основна функціональність процедури полягає у перевірці кількості накопичених показників FPS у файлі результатів.

Процедура включає два сценарії:

1. Якщо кількість накопичених показників перевищує задане значення (exp_num), процедура завершує всі процеси за вказаним шляхом.
2. Якщо кількість накопичених показників недостатня, процедура очікує протягом 1 секунди і продовжує цикл.

Така система забезпечує автоматичне завершення тестування після накопичення статистично значущої вибірки даних, підвищуючи ефективність процесу тестування.

Висновки щодо структури тестового середовища. Розроблена система процедур для тестування швидкодії ЯІТ демонструє системний підхід до організації експериментальних досліджень, ключовими особливостями котрої є:

1. Модульність – розбиття складного процесу тестування на окремі функціональні процедури, що спрощує розуміння, модифікацію та підтримку тестового середовища.
2. Ієрархічність – чітка структура взаємодії процедур, що забезпечує логічну послідовність виконання операцій.
3. Гнучкість – можливість налаштування параметрів тестування для різних сценаріїв використання.
4. Автоматизація – мінімізація необхідності ручного втручання в процес тестування.
5. Систематизація – організоване зберігання результатів тестування для подальшого аналізу.

Такий підхід дозволяє провести всебічне дослідження швидкодії розробленої інформаційної технології в різних умовах використання та для різних параметрів обробки ЦЗ.

4.6 Аналіз показників швидкодії векторизованої операції згортки цифрового зображення.

Візуалізація отриманих результатів у вигляді матриць 8×14 дозволяє наочно представити швидкодію ШОЗ ЦЗ запропонованого ЯІТ conv_RGB_NEON(…) у порівнянні з ШОЗ OpenCV cv::filter2D(...) для різних комбінацій роздільної здатності ЦЗ з відеопотоку та розмірів ЯЗ, враховуючи як прямі виклики функцій ВОЗ, так і їх Python-виклики. Це дає можливість швидко оцінити ефективність оптимізації та виявити умови, за яких запропонований алгоритм демонструє найбільшу перевагу [3]. Для забезпечення об’єктивного порівняння швидкодії розробленого ЯІТ - conv_RGB_NEON(…) та функції ШОЗ OpenCV - cv::filter2D(…), було застосовано методологію тестування, що описана в (підрозд. 4.5).

Тестування проводилось на апаратних платформах OrangePi 5 Pro та Rasp-

berry Pi 4. Для кожної комбінації параметрів (розмір ЯЗ та роздільна здатність ЦЗ) було зібрано статистично значущу вибірку показників швидкодії у кадрах на секунду (FPS), що дозволило об'єктивно оцінити ефективність обох реалізацій та їх порівняльні характеристики [8].

4.6.1 Аналіз швидкодії векторизованої операції згортки цифрового зображення

Таблиця 4.3 — Показники роботи ШОЗ ЯІТ та cv::filter2D(…) (C++ реалізація). Показник в FPS.

Orange Pi 5 Pro

conv_RGB_NEON(…)

	256	426	640	854	1280	1920	2560	3840
15	681.8	229.4	100.8	56.8	24.6	10.9	6.1	2.7
14	777.3	259.3	114.6	64.3	28.1	12.5	7.0	3.0
13	858.8	294.3	129.1	72.8	31.7	14.0	7.8	3.4
12	995.1	342.3	148.7	84.7	36.7	16.2	9.1	4.0
11	1168.9	424.2	180.5	103.5	42.5	18.7	10.4	4.5
10	1337.8	498.4	219.3	123.1	54.6	24.1	12.6	5.5
9	1616.1	548.0	238.9	144.4	59.0	25.8	14.4	6.4
8	1917.1	680.1	305.6	172.2	71.6	31.5	17.6	7.8
7	2296.7	782.1	337.8	213.8	83.3	37.2	20.7	9.2
6	2645.1	1002.7	463.0	236.8	106.2	45.8	25.1	11.2
5	3021.7	1251.3	575.6	323.0	128.0	57.9	30.8	14.0
4	3152.9	1600.8	740.5	450.8	179.6	83.9	39.6	17.7
3	2609.0	1527.6	793.0	479.6	201.8	89.3	48.1	21.4
2	3184.8	1717.4	880.6	483.9	230.7	113.7	60.0	26.5

3185 2831 2478 2124 1771 1417 1063 710 356 3

cv::filter2D(…)

	256	426	640	854	1280	1920	2560	3840
15	171.5	102.9	40.1	26.6	12.0	5.6	3.2	1.5
14	171.9	102.6	40.2	26.6	12.0	5.7	3.3	1.4
13	171.9	103.2	40.3	26.7	12.0	5.7	3.3	1.5
12	172.4	102.2	40.3	26.9	12.1	5.7	3.3	1.5
11	172.7	89.7	40.3	26.9	12.1	5.7	3.3	1.5
10	172.9	89.6	40.4	26.9	12.1	5.7	3.3	1.5
9	173.2	89.8	40.5	26.9	12.1	5.7	3.3	1.5
8	172.9	90.1	40.4	26.8	12.1	5.7	3.3	1.5
7	1009.3	360.8	154.2	86.7	38.4	16.8	9.4	4.2
6	1365.0	482.0	210.3	115.2	51.3	22.2	12.5	5.5
5	1830.7	645.9	277.1	156.3	67.3	28.5	16.0	7.0
4	2607.7	907.2	427.3	247.3	98.1	40.8	22.6	10.1
3	3205.8	1360.6	579.0	367.3	148.3	61.4	31.9	13.9
2	2984.5	1753.7	842.9	513.7	246.7	113.6	54.1	23.5

3206 2850 2494 2138 1782 1426 1070 714 357 1

Raspberry Pi 4

conv_RGB_NEON(…)

	256	426	640	854	1280	1920	2560	3840
15	308.3	105.2	45.5	25.4	11.0	4.8	2.6	1.2
14	348.6	116.1	51.5	28.3	12.3	5.4	2.9	1.3
13	386.6	132.1	57.2	31.8	13.6	6.0	3.3	1.4
12	435.8	151.0	65.4	37.2	15.8	6.9	3.9	1.7
11	517.6	185.5	76.7	43.2	18.7	8.2	4.5	2.0
10	602.9	219.6	94.1	53.8	23.4	10.1	5.3	2.4
9	730.4	251.1	113.0	60.7	25.7	11.2	6.3	2.8
8	867.9	293.8	127.2	73.1	32.6	13.9	7.6	3.3
7	1070.3	381.7	156.8	85.3	37.5	15.9	8.9	4.0
6	1348.1	452.1	196.2	116.7	47.0	20.5	11.6	5.1
5	1586.2	485.6	215.4	127.5	55.5	26.4	14.6	5.9
4	1609.8	662.5	278.9	165.1	70.7	32.4	17.5	7.1
3	1668.0	614.4	282.8	162.0	76.8	36.7	20.0	8.2
2	1702.8	654.8	305.5	173.6	82.2	36.7	19.8	8.9

1703 1514 1325 1136 947 757 568 379 190 1

cv::filter2D(…)

	256	426	640	854	1280	1920	2560	3840
15	83.8	50.9	19.7	13.0	5.8	2.8	1.6	0.7
14	83.8	50.8	19.8	13.1	5.8	2.7	1.6	0.7
13	84.1	50.8	19.8	13.0	5.8	2.8	1.6	0.7
12	84.1	50.5	19.7	13.0	5.8	2.8	1.6	0.7
11	84.5	45.2	19.9	13.0	5.9	2.7	1.6	0.7
10	84.3	45.2	19.7	13.1	5.9	2.8	1.6	0.7
9	84.6	45.4	19.9	13.0	5.8	2.8	1.6	0.7
8	84.7	45.5	20.0	13.2	5.9	2.8	1.6	0.7
7	456.9	162.1	70.9	39.7	17.2	7.5	4.3	1.9
6	621.5	218.5	95.1	54.8	23.7	10.4	5.8	2.5
5	823.7	281.9	124.7	70.4	30.3	14.4	7.4	3.1
4	1179.1	424.9	178.5	105.7	45.5	20.9	10.5	4.5
3	1519.3	576.1	234.9	143.8	57.9	24.6	13.7	5.8
2	1212.4	692.0	295.3	185.1	80.2	38.3	19.4	8.0

1519 1351 1182 1013 844 676 507 338 169 1

Варто зазначити, що на відміну від реалізації ШОЗ conv_RGB_NEON(…), функція cv::filter2D(…) буде демонструвати дещо інший характер розподілу

швидкодії. Саме падіння швидкодії пов'язане зі зміною алгоритму/функції ШОЗ cv::filter2D(…) [22].

Слід зазначити, що по осі Ox розташовано ширину фрейму/ЦЗ (повний розмір ЦЗ вказано в (табл. 4.2)), а по осі Oy розмір ЯЗ у форматі одного числа – N ($N \times N$).

Аналіз отриманих даних дозволяє виявити кілька ключових закономірностей для алгоритму conv_RGB_NEON(…):

1. **Різке падіння швидкодії функції cv::filter2D(…) на ЯЗ 8×8 і більше:** Неозброєним оком видно, що показники FPS (табл. 4.3) функції cv::filter2D різко знижуються при переході від ЯЗ 7×7 до 8×8 на обох платформах:
   (а) На Orange Pi 5 Pro: падіння з 1009,3 FPS (ЯЗ 7×7) до 172,9 FPS (ЯЗ 8×8) для зображення 256×144 - зниження в ∼5,8 раза.
   (б) На Raspberry Pi 4: падіння з 456,9 FPS (ЯЗ 7×7) до 84,7 FPS (ЯЗ 8×8) для зображення 256×144 - зниження в ∼5,4 раза.

   Причиною цього є виявлений у вихідному коді OpenCV механізм автоматичного вибору алгоритму ВОЗ. У файлі (filter.dispatch.cpp:1275) в функції dftFilter2D(...) присутня така умова – «if (kernel_width * kernel_height < dft_filter_size) { return false;}». Ця умова визначає, який алгоритм буде використано: при невиконанні умови (для ЯЗ до 7×7) використовується більш ефективна реалізація, що описана в (підрозд. 1.4), а при виконанні (для ЯЗ 8×8 і більше) – ОЗ на основі швидкого перетворення Фур'є (FFT). Хоча теоретично FFT-алгоритм має кращу асимптотичну складність для дуже великих ЯЗ, на практиці для ЯЗ середнього розміру він виявляється значно повільнішим через додаткові накладні витрати на перетворення Фур'є та менш ефективне використання кеш-пам'яті.
2. **Стабільна перевага запропонованого методу conv_RGB_NEON(…):** Показники швидкодії запропонованого методу ЯІТ/conv_RGB_NEON(…) демонструють перевагу над cv::filter2D(…) для всіх розмірів ЯЗ від 4×4, особливо значну для ядер 8×8 і більше. Навіть для малих ЯЗ (4×4 - 7×7), де OpenCV показує хорошу швидкодію, ЯІТ є швидшим завдяки використанню оптимізованого підходу (підрозд. А.4.6). Для найменших ЯЗ (2×2 та 3×3) в окремих випадках спостерігається паритет або незначне сповільнення (до 0,8 раза), що є наслідком обмеження підсистеми вводу-виводу, а не швидкодії самої ШОЗ — надалі це твердження буде детально

розглянуто у частині «Пряме вимірювання швидкодії ОЗ» та (табл. 4.5). Отже, на практиці підхід (підрозд. А.4.6) дозволяє:

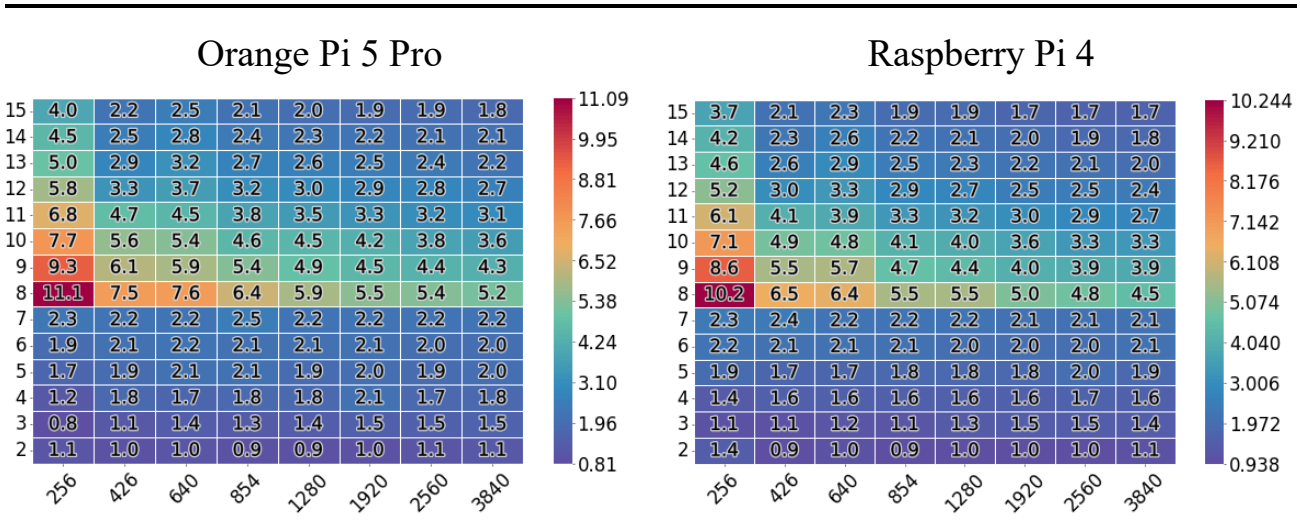
(а) Мінімізувати кількість звернень до пам'яті.

(б) Не потребує використання кеш-пам'яті, бо, натомість, максимально ефективно використовує векторні регістри NEON64.

(в) Уникнути накладних витрат на повторне завантаження даних зображення.

3. **Фундаментальна відмінність у реалізації обчислень:**
Окрім вибору алгоритму, на швидкодію суттєво впливає стратегія обробки даних. Функція cv::cpu_baseline::FilterVec_8u::operator() в OpenCV використовує систему Universal Intrinsics, що забезпечує кросплатформність, але водночас призводить до додаткових перетворень типів. Як зазначено у (підрозд. 1.4), OpenCV використовує послідовність перетворень 8u→16s→32s→32f для прорахунку ВСД. Це надає/забезпечує захист від переповнень, що, хоч і незначно, але пригнічує швидкодію, особливо на ARM64 платформах [81].
На противагу цьому, запропонований ЯІТ/conv_RGB_NEON(…) використовує оптимізований підхід з 16-бітними цілочисельними обчисленнями (використовується послідовність перетворень 8u→16s для прорахунку ВСД), що забезпечує баланс між точністю та захистом від переповнень та швидкодією.

Таблиця 4.4 — Порівняння швидкодії ШОЗ ЦЗ. Безрозмірний показник.



Після огляду ключових особливостей, слід розглянути схожі патерни при-

скорення ЯІТ/ШОЗ відносно cv::filter2D, а саме, те що для обох методів спостерігається закономірне зниження швидкодії зі збільшенням роздільної здатності ЦЗ, але ЯІТ демонструє кращу масштабованість. Особливо це помітно для ЦЗ високої роздільної здатності (1920×1080 і вище), де перевага ЯІТ залишається суттєвою навіть для великих ЯЗ.

Збір та проведення автоматичного порівняльного аналізу швидкодії (коефіцієнтів прискорення ЯІТ відносно cv::filter2D(…) описано в підрозд. 4.5.1) на двох різних платформах виявляє важливу закономірність - платформну незалежність обох ШОЗ (ЯІТ та cv::filter2D(…)) оптимізацій (табл. 4.4), що означає:

1. **Схожість патернів прискорення:** обидві тестові платформи (Orange Pi 5 Pro та Raspberry Pi 4) демонструють надзвичайно схожі патерни прискорення ЯІТ відносно cv::filter2D(…):
   (а) Максимальне прискорення для ЯЗ 8–11 (Orange Pi 5 Pro: 3,1–11,1 раза, Raspberry Pi 4: 2,7–10,2 раза).
   (б) Помірне прискорення для малих ЯЗ 2–7 (Orange Pi 5 Pro: 0,8–2,5 раза, Raspberry Pi 4: 0,9–2,4 раза). Надалі у (табл. 4.5) та описі до неї буде розглянуто причини сповільнення швидкодії ШОЗ відносно cv::filter2D(…) в окремих випадках для ЯЗ розміром 2×2 та 3×3;
   (в) Стабільно високе прискорення для великих ЯЗ 12–15 (Orange Pi 5 Pro: 1,8–5,8 раза, Raspberry Pi 4: 1,7–5,2 раза).

   Ця схожість переконливо доводить, що запропоновані/обидві оптимізації є платформно-незалежними в рамках архітектури ARM64 і ефективно працюють на різних імплементаціях процесорів ARM64, але ЯІТ показує значно швидші показники швидкодії відносно cv::filter2D(…).
2. **Стабільність переваги для всіх роздільних здатностей:** На обох платформах зберігається стабільна перевага ЯІТ для всіх роздільних здатностей ЦЗ. Хоча абсолютний коефіцієнт прискорення дещо знижується для зображень надвисокої роздільної здатності (3840×2160), відносна перевага залишається за ЯІТ і є значною (Orange Pi 5 Pro: 1,1–5,2 раза, Raspberry Pi 4: 1,1–4,5 раза на різних ядрах).
3. **Універсальність оптимізацій:** Незважаючи на різні характеристики процесорів (Orange Pi 5 Pro використовує Rockchip RK3588S з ARMv8.2-A, а Raspberry Pi 4 - Broadcom BCM2711 з ARMv8-A), запропоновані оптимізації демонструють схожу ефективність. Це підтверджує, що розроблене ЯІТ

ефективно використовує спільні можливості NEON64 інструкцій, доступні на обох платформах.

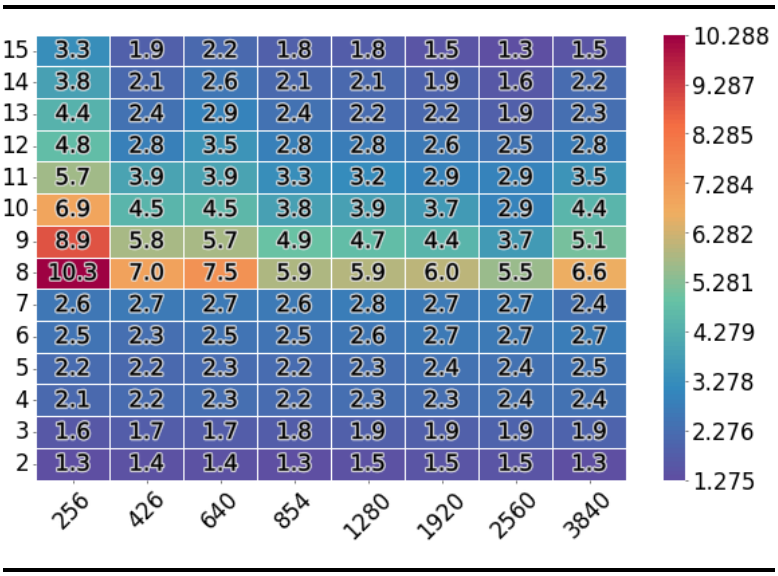
Таким чином, аналіз показників швидкодії підтверджує значну перевагу ЯІТ/conv_RGB_NEON(…) над стандартною реалізацією cv::filter2D(…) на різних ARM64 платформах. Особливо важливим є те, що найбільша перевага досягається для ЯЗ середнього та великого розміру, які часто використовуються в практичних задачах обробки ЦЗ.

Пряме вимірювання швидкодії ОЗ. Оскільки на малих ЯЗ та роздільностях виміряний показник FPS обмежується пропускною здатністю зчитування кадру — про що свідчить немонотонність показників у (табл. 4.3): наприклад, FPS для ЯЗ 3×3 подекуди нижчий, ніж для 4×4, що для самої ОЗ неможливо, — було проведено додаткове пряме вимірювання часу виконання обох реалізацій без участі вводу-виводу на платформі Orange Pi 5 Pro (з урахуванням платформно незалежного характеру прискорення). До особливостей прямого вимірювання швидкодії ОЗ слід віднести наступне:

1. Синтетичне RGB ЦЗ формується безпосередньо в пам’яті;
2. Обидві реалізації виконуються в одному тестовому процесі над однаковими даними в однопотоковому режимі (для OpenCV явно встановлено cv::setNumThreads(1)).
3. Час виконання кожної функції визначається інструментом профілювання perf (Linux) як частка процесорного часу (task-clock), віднесена до піддерева викликів відповідної функції, тож час генерації ЦЗ до вимірюваного часу не входить.

Також слід зазначити, що було використано розріджені ЯЗ зі змішаною знаковістю елементів — найскладніший для ЯІТ випадок, що активує гілку кластеризації (той самий тип ЯЗ, що й у решті експериментів цього розділу). Оскільки час виконання ШОЗ зростає приблизно квадратично від розміру ЯЗ, а час FFT-гілки cv::filter2D(…) практично не залежить від нього, екстраполяція виміряних трендів дає точку беззбитковості прямого методу на рівні ЯЗ ≈28×28 для 256×144 та ≈18×18 для 3840×2160 — тобто за межами дослідженого діапазону та практичних розмірів ЯЗ фільтрації.

Таблиця 4.5 — Пряме порівняння часу виконання ШОЗ ЯІТ та cv::filter2D(…) інструментом perf (OpenCV 4.13.0, один потік, Orange Pi 5 Pro). Безрозмірний показник.



За прямим порівнянням (табл. 4.5) ЯІТ виявився швидшим за cv::filter2D(…) в усіх 112 комбінаціях параметрів: для малих ЯЗ (2×2–7×7) — в 1,3–2,8 раза з монотонним зростанням переваги від розміру ЯЗ; для середніх (8×8–11×11) — у 2,9–10,3 раза; для великих (12×12–15×15) — в 1,3–4,8 раза. Додатково зафіксовано, що час виконання cv::filter2D(…) для ЯЗ від 8×8 до 15×15 за фіксованої роздільності практично не змінюється — незалежність вартості обчислень від розміру ЯЗ характерна саме для FFT-алгоритму, що незалежно підтверджує висновок п. 1 попереднього переліку. Запуск даного тесту на OpenCV 5.0.0 дав практично ідентичні результати (в усіх комбінаціях $\geq$1,3; час виконання cv::filter2D(…) між версіями збігся в межах 1%), тобто отримані співвідношення не зумовлені версією бібліотеки. Таким чином, близькі до паритету коефіцієнти для малих ЯЗ у (табл. 4.4) відображають обмеження підсистеми вводу-виводу, а не швидкодію самої ШОЗ, що слід розглядати як нижню оцінку переваги ЯІТ.

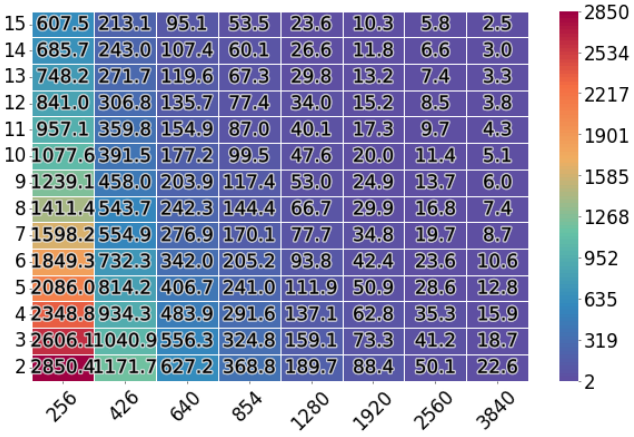
4.6.2 Аналіз швидкодії Python-інтерфейсу ШОЗ ЦЗ.

Аналіз показників швидкодії Python-інтерфейсу ЯІТ демонструє аналогічні закономірності з C++ реалізацією, що описано в (підрозд. 4.6.1), але з суттєво нижчими абсолютними значеннями FPS.

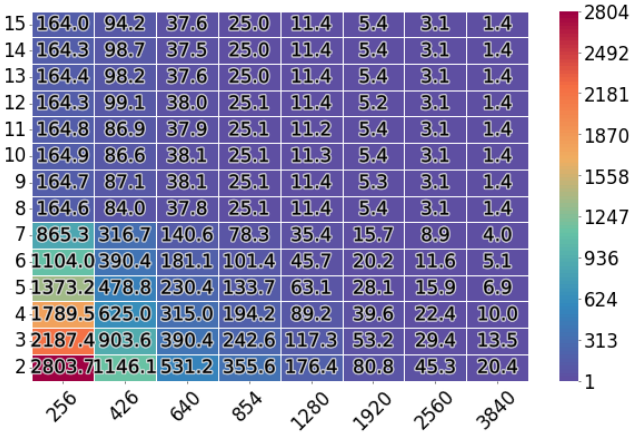
Таблиця 4.6 — Показники роботи ШОЗ ЦЗ в Python-додатку. Показник в FPS.

Orange Pi 5 Pro

conv_RGB_NEON

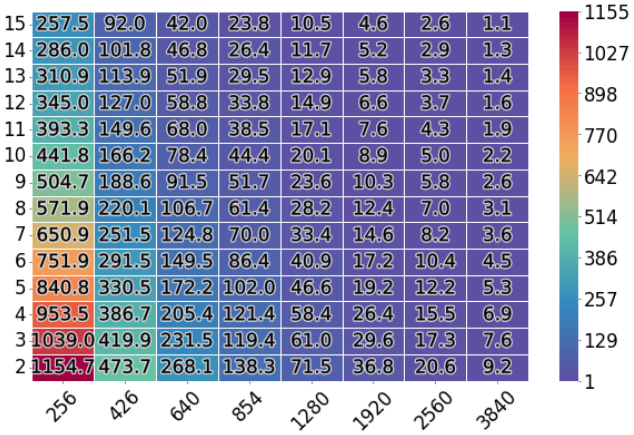


cv::filter2D

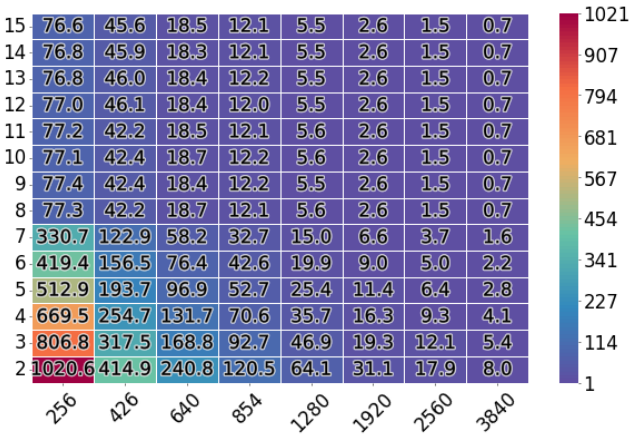


Raspberry Pi 4

conv_RGB_NEON



cv::filter2D



Показники FPS для Python-інтерфейсу обох ШОЗ (як ЯІТ, так cv2_filter 2D(…)) є значно нижчими (табл. 4.6) порівняно з прямими викликами C++ функцій (табл. 4.3) через два основних фактори:

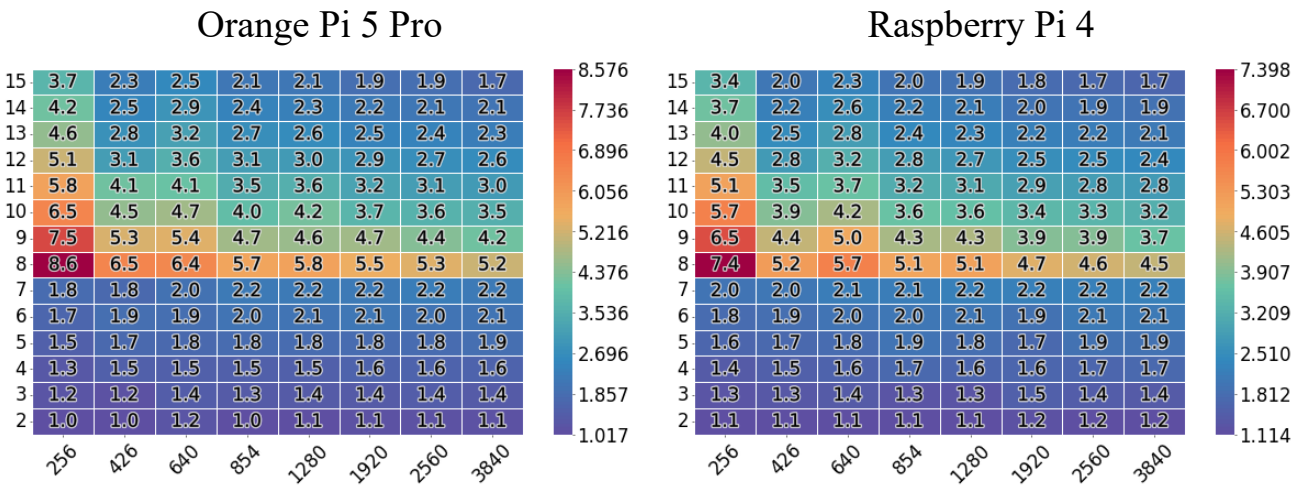
1. відсутність багатопоточності у зчитуванні ЦЗ з відеопотоку, на відміну від C++ реалізації;
2. Python-версія використовує однопоточний режим зчитування, що також створює додаткові затримки при обробці/зчитуванні кадрів;
3. Третім, прихованим, фактором є накладні витрати на виклик функцій через використання бібліотеки ctypes для взаємодії Python з C++ кодом/бібліотекою, що включає в себе: конвертацію типів даних (Python → C++); передачу вказівників через ctypes.POINTER(ctypes.c_ubyte); створення

С-масивів типу (ctypes.c_int * (flt_cols * flt_rows)).

Незважаючи на зниження абсолютних показників, основні патерни швидкодії обох ШОЗ залишаються незмінними:

1. Різке падіння швидкодії cv2_filter2D(…) при переході від ядра 7×7 до 8×8.
2. Плавне зниження швидкодії conv_RGB_NEON(…) зі збільшенням розміру ядра.
3. Експоненційне зниження швидкодії (FPS) зі збільшенням роздільної здатності для обох методів.

Таблиця 4.7 — Показники порівняння швидкодії ШОЗ ЦЗ в Python-додатку на різних платформах. Безрозмірний показник.



Порівняльний аналіз відносної швидкодії Python-інтерфейсу підтверджує висновки, зроблені у підрозд. 4.6.1, демонструючи платформну незалежність запропонованих оптимізацій (табл. 4.7):

1) Збереження патернів прискорення. Коефіцієнти прискорення залишаються практично ідентичними тим, що продемонстровано у (табл. 4.4):

1. Максимальне прискорення для ядер середнього розміру (8–11): 3,0–8,6 раза для Orange Pi 5 Pro; 2,8–7,4 раза для Raspberry Pi 4.
2. Помірне прискорення для великих ядер (12–15): 1,7–5,1 раза для Orange Pi 5 Pro; 1,7–4,5 раза для Raspberry Pi 4.
3. Близька до паритету швидкодії для малих ядер (2–7): 1,0–2,2 раза для Orange Pi 5 Pro; 1,1–2,2 раза для Raspberry Pi 4.

2) Схожість показників роботи ШОЗ на платформах Orange Pi 5 Pro та Raspberry Pi 4 в Python-інтерфейсі додатково підтверджує, що запропоновані ШОЗ ЯІТ

оптимізовано ефективно, незалежно від: конкретної апаратної платформи ARM64; способу виклику функцій (прямий C++ або через Python ctypes); наявності додаткових накладних витрат.

В цілому це свідчить про фундаментальну ефективність розробленого ЯІТ/conv_RGB_NEON(…) на рівні алгоритмічних оптимізацій NEON64, які зберігають свою перевагу навіть при використанні через високорівневі Python інтерфейси.

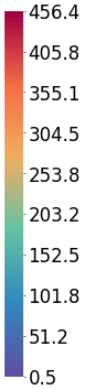
4.6.3 Аналіз швидкодії операцій масштабування ЦЗ.

Таблиця 4.8 — Показники роботи операцій масштабування ЦЗ в Python-додатку: Показник в FPS.

Orange Pi 5 Pro

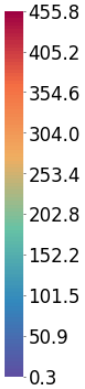
zoomUP_2__NEON64(…)

	256	426	640	854	1280	1920	2560	3840
15	132.7	46.5	20.5	11.6	5.1	2.2	1.2	0.5
14	147.7	51.0	22.7	12.8	5.6	2.5	1.3	0.6
13	159.4	56.3	24.8	14.0	6.2	2.7	1.5	0.7
12	175.0	62.0	27.6	15.7	6.9	3.1	1.7	0.7
11	196.2	71.4	30.7	17.3	7.7	3.4	1.8	0.8
10	216.0	77.2	34.1	19.2	8.6	3.8	2.0	0.9
9	240.8	84.9	38.0	21.5	9.6	4.3	2.3	1.0
8	251.2	95.9	41.2	24.2	10.8	4.8	2.5	1.1
7	294.1	105.3	46.8	26.7	12.1	5.3	2.8	1.2
6	331.4	117.1	52.1	30.4	13.9	6.1	3.1	1.4
5	360.9	129.3	58.6	34.0	15.4	6.9	3.4	1.6
4	399.6	142.5	65.6	38.5	17.1	7.7	4.0	1.8
3	425.5	152.2	70.6	41.1	18.5	8.2	4.2	1.9
2	456.4	164.2	78.2	44.3	20.0	8.9	4.5	2.0



zoomUP_2__cv2_filter2D(…)

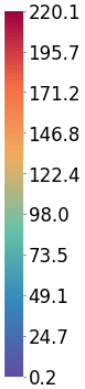
	256	426	640	854	1280	1920	2560	3840
15	39.5	23.0	8.9	5.9	2.6	1.2	0.7	0.3
14	39.5	23.0	8.9	5.8	2.6	1.2	0.7	0.3
13	39.6	22.6	9.0	5.9	2.6	1.2	0.7	0.3
12	39.6	23.1	8.9	5.9	2.6	1.2	0.7	0.3
11	39.7	20.4	9.0	5.9	2.7	1.2	0.7	0.3
10	39.8	20.4	8.9	5.9	2.6	1.2	0.7	0.3
9	39.9	20.1	9.0	5.9	2.7	1.2	0.7	0.3
8	39.8	20.4	8.9	5.9	2.6	1.2	0.7	0.3
7	180.5	63.8	28.5	15.9	7.1	3.2	1.7	0.8
6	221.9	78.3	34.9	19.6	8.7	3.8	2.0	0.9
5	261.7	92.6	41.3	23.2	9.8	4.6	2.4	1.1
4	323.2	113.9	50.8	28.7	13.5	5.7	3.0	1.3
3	376.6	131.9	60.8	34.2	15.8	7.0	3.5	1.6
2	455.8	159.6	76.2	43.3	19.4	8.6	4.4	2.0



Raspberry Pi 4

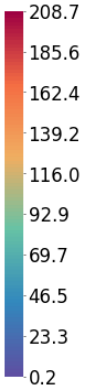
zoomUP_2__NEON64(…)

	256	426	640	854	1280	1920	2560	3840
15	60.1	21.5	9.4	5.3	2.3	1.0	0.6	0.2
14	66.1	23.5	10.4	5.8	2.6	1.1	0.6	0.3
13	71.5	26.0	11.4	6.4	2.8	1.2	0.7	0.3
12	78.4	28.8	12.7	7.2	3.1	1.4	0.8	0.3
11	88.4	33.2	14.2	8.0	3.5	1.6	0.9	0.4
10	97.7	36.7	15.9	9.0	4.0	1.8	1.0	0.4
9	110.0	41.1	18.2	10.3	4.6	2.0	1.1	0.5
8	119.0	46.7	20.6	11.7	5.2	2.3	1.2	0.5
7	136.5	52.2	23.2	13.1	5.9	2.6	1.4	0.6
6	155.0	58.4	26.5	14.9	6.8	3.0	1.6	0.7
5	168.2	65.9	29.4	16.8	7.4	3.3	1.8	0.8
4	188.5	71.5	32.0	19.0	8.5	3.8	2.0	0.9
3	201.4	78.7	35.7	20.3	8.7	4.1	2.1	1.0
2	220.1	86.3	39.4	22.5	10.2	4.6	2.4	1.0



zoomUP_2__cv2_filter2D(…)

	256	426	640	854	1280	1920	2560	3840
15	19.1	11.1	4.4	2.9	1.3	0.6	0.3	0.2
14	19.1	11.1	4.4	2.9	1.3	0.6	0.3	0.2
13	18.9	11.1	4.3	2.9	1.3	0.6	0.4	0.2
12	19.2	11.1	4.4	2.9	1.3	0.6	0.3	0.2
11	19.2	10.2	4.4	2.9	1.3	0.6	0.3	0.2
10	19.2	10.3	4.4	2.9	1.3	0.6	0.3	0.2
9	19.3	10.3	4.4	2.9	1.3	0.6	0.3	0.2
8	19.3	10.2	4.4	2.9	1.3	0.6	0.3	0.2
7	76.5	28.3	12.6	7.1	3.2	1.4	0.8	0.3
6	94.1	35.0	15.7	8.9	4.0	1.8	1.0	0.4
5	113.9	42.2	18.9	10.8	4.9	2.2	1.2	0.5
4	142.9	53.8	24.1	13.7	6.2	2.8	1.5	0.7
3	169.4	64.2	28.7	16.2	7.5	3.4	1.8	0.8
2	208.7	79.7	35.7	20.4	9.4	4.2	2.2	1.0



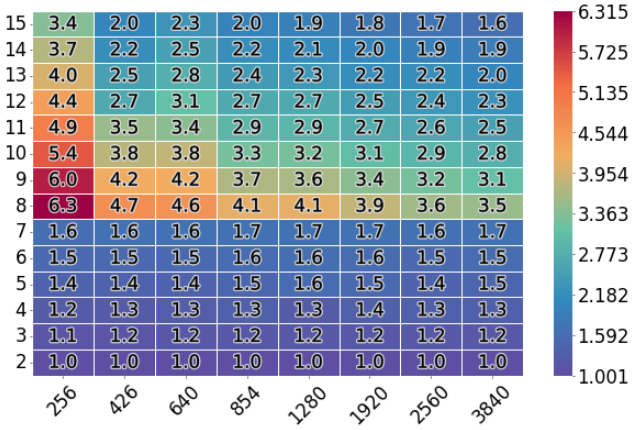
Аналіз показників швидкодії операцій масштабування (табл. 4.8) спирається на очікувані закономірності, які слідують з архітектури/коду цих комплексних функцій, та на «негативний» вплив Python інтерфейсних викликів.

Функції zoomUP_2__NEON64() та zoomUP_2__cv2_filter2D() реалізують алгоритм збільшення зображення в 2 рази (див. підрозд. 2.2), послідовно застосовуючи чотири різні ЯЗ до вхідного ЦЗ і об'єднуючи їх (чотири результуючі ЦЗ) за принципом шахової дошки, через функцію combine_frames_checkerboard().

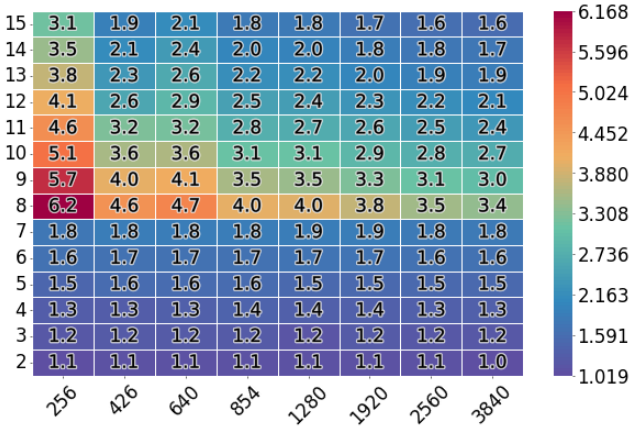
З вищесказаного виходить, що кожна операція масштабування фактично виконує чотири повні ШОЗ ЦЗ та додаткову операцію об'єднання результатів у вихідне ЦЗ. З цього виходить прогнозоване щонайменше чотириразове зниження показників FPS порівняно з базовою ШОЗ (табл. 4.6); фактично виміряне зниження становить приблизно 4–11 раза і зростає з роздільною здатністю ЦЗ. І, додатково, використання Python, при виклику ЯІТ, призводить до характерного пригнічення швидкодії через накладні витрати ctypes-інтерфейсу, що було продемонстровано у (підрозд. 4.6.2), та додаткові операції з пам'яттю (створення та обробка чотирьох проміжних ЦЗ і фінальне об'єднання у вихідне ЦЗ).

Таблиця 4.9 — Показники порівняння швидкодії ШОЗ ЦЗ операцій масштабування в Python-додатку. Безрозмірний показник.

Orange Pi 5 Pro



Raspberry Pi 4



З (табл. 4.8) слідує, що незважаючи на суттєве зниження абсолютних показників FPS, основні закономірності залишаються сталими:

1. Різке падіння швидкодії zoomUP_2__cv2_filter2D(…) при обробці ЯЗ, що відповідають переходу $7\times7\to8\times8$.
2. Плавне/закономірне зниження швидкодії zoomUP_2__NEON64(…) зі збіль-

шенням розміру ЯЗ.

3. Експоненційне зниження швидкодії зі збільшенням роздільної здатності.

Як видно з (табл. 4.8) та (табл. 4.9), порівняльний аналіз відносної швидкодії операцій масштабування ЦЗ підтверджує стабільність зростання швидкодії ЯІТ навіть у комплексних сценаріях використання. Відносні показники прискорення zoomUP_2__NEON64(…) порівняно з zoomUP_2__cv2_filter2D(…) демонструють схожі патерни з базовими ШОЗ, але з дещо меншими абсолютними значеннями:

1. Максимальне прискорення для ЯЗ середнього розміру (8–11): 2,5–6,3 раза для Orange Pi 5 Pro; 2,4–6,2 раза для Raspberry Pi 4.
2. Помірне прискорення для великих ядер (12–15): 1,6–4,4 раза для Orange Pi 5 Pro; 1,6–4,1 раза для Raspberry Pi 4.
3. Близька до паритету швидкодії для малих ядер (2–7): 1,0–1,7 раза для Orange Pi 5 Pro; 1,0–1,9 раза для Raspberry Pi 4.

Порівняння з показниками (табл. 4.7) виявляє зменшення коефіцієнтів прискорення приблизно в 1,0–1,5 раза, що пояснюється:

1. Пропорційним впливом накладних витрат - додаткові витрати Python-інтерфейсу впливають на обидві ВОЗ.
2. Накладними витратами на управління пам’яттю - створення та передача чотирьох масивів даних через ctypes.
3. Синхронізацією виконання та послідовне виконання чотирьох ВОЗ ЦЗ без можливості паралелізації.

Схожість показників на платформах Orange Pi 5 Pro та Raspberry Pi 4 для операцій масштабування додатково підтверджує універсальність запропонованих оптимізацій, що використовуються в основі розробленої ІТ.

Висновки щодо операцій масштабування:

1. Прогнозованість зниження швидкодії - зменшення FPS пропорційне збільшенню обчислювальної складності ($\times 4$ ОЗ).
2. Стабільність переваг – запропонований метод (на основі ЯІТ) зберігає значну перевагу навіть у комплексних сценаріях використання.
3. Ефективність для практичних застосувань - незважаючи на зниження абсолютних показників, операції масштабування залишаються працездатними для обробки відео в режимі, близькому до реального часу.

Це підтверджує практичну цінність розробленої ІТ для створення компле-

ксних систем обробки зображень на ARM64 платформах, де необхідно поєднувати високу швидкодію з гнучкістю програмного інтерфейсу [62, 63].

4.7 Висновки

У четвертому розділі дисертаційної роботи проведено експериментальне дослідження швидкодії розробленої ІТ обробки ЦЗ із застосуванням ШОЗ на базі технології NEON64, що використовує 16-бітні цілочисельні обчислення. На основі отриманих результатів зроблено такі висновки:

1. Розроблено та реалізовано комплексну методику експериментального дослідження швидкодії, що включає стандартизовані умови тестування на різних апаратних платформах (SBC Orange Pi 5 Pro та Raspberry Pi 4), використання широкого діапазону розмірів ЦЗ (від 256×144 до 3840×2160) та ЯЗ (від 2×2 до 15×15). Для автоматизації процесу тестування та збору даних було розроблено спеціалізовані Shell-скрипти та Python-інтерфейс, що забезпечило високу відтворюваність та достовірність результатів.
2. Проведено порівняльний аналіз швидкодії (в FPS) розробленого алгоритму conv_RGB_NEON(...) та функції cv::filter2D(...) бібліотеки OpenCV для C++ реалізацій. Встановлено, що розроблений алгоритм демонструє значну перевагу в швидкодії на обох тестових платформах:
    - Для ЯЗ середнього розміру (8×8 – 11×11) прискорення становить 3,1–11,1 раза для Orange Pi 5 Pro та 2,7–10,2 раза для Raspberry Pi 4.
    - Для великих ЯЗ (12×12 – 15×15), де cv::filter2D(...) переходить на використання FFT-алгоритмів (що підтверджується різким падінням її швидкодії для ядер розміром 8×8 та більше), запропонований метод прямої згортки з NEON64 оптимізацією показує прискорення в 1,8–5,8 раза для Orange Pi 5 Pro та 1,7–5,2 раза для Raspberry Pi 4.
    - Для малих ЯЗ (2×2 – 7×7) прискорення є помірним (0,8–2,5 раза для Orange Pi 5 Pro та 0,9–2,4 раза для Raspberry Pi 4), причому близькі до паритету значення для ЯЗ 2×2–3×3 зумовлені обмеженням пропускної здатності зчитування кадру, а не швидкодією самої ОЗ: пряме вимірювання часу виконання без вводу-виводу (табл. 4.5) показує перевагу ЯІТ у 1,3–2,8 раза для всіх малих ЯЗ.
3. Виявлено, що переваги розробленої ІТ зберігаються та масштабуються зі

збільшенням роздільної здатності оброблюваних ЦЗ, підтверджуючи ефективність застосованих оптимізацій для обробки великих обсягів даних.

4. Аналіз швидкодії Python-інтерфейсу для розробленої ІТ показав, що, незважаючи на накладні витрати, пов'язані з використанням ctypes та однопоточним виконанням Python-скриптів, відносні переваги запропонованого ЯІТ-conv_RGB_NEON(...) та cv::filter2D(...) зберігаються і демонструють схожі патерни прискорення, що й у C++ реалізаціях. Це підтверджує ефективність оптимізацій на рівні ЯІТ незалежно від способу його виклику.
5. Дослідження швидкодії операцій масштабування зображень (zoomUP_2__NEON64 та zoomUP_2__cv2_filter2D), які базуються на чотирикратному застосуванні ВОЗ, підтвердило стабільність переваг розробленого підходу. Запропонований метод масштабування (zoomUP_2__NEON64) продемонстрував прискорення до 6,3 раза (Orange Pi 5 Pro) та до 6,2 раза (Raspberry Pi 4) для ЯЗ середнього розміру порівняно з аналогічною операцією на базі cv::filter2D(...). Зменшення коефіцієнтів прискорення порівняно з одиночною ОЗ пояснюється пропорційним впливом накладних витрат Python-інтерфейсу та управлінням пам'яттю.
6. Підтверджено платформну незалежність та універсальність запропонованих оптимізацій: схожі патерни прискорення та переваги розробленої ІТ спостерігалися як на платформі Orange Pi 5 Pro (з SoC Rockchip RK3588S), так і на Raspberry Pi 4 (з SoC Broadcom BCM2711) — пікові прискорення становлять, відповідно, 11,1 та 10,2 раза (розбіжність менше 10%), що свідчить про ефективне використання спільних можливостей архітектури ARMv8-A та її SIMD-інструкцій NEON64.

Експериментальні дослідження повністю підтвердили високу ефективність розробленої інформаційної технології та оптимізованого алгоритму ВОЗ, що дозволяє суттєво підвищити швидкодію обробки ЦЗ на сучасних одноплатних комп'ютерах з архітектурою ЦП ARM64.

ВИСНОВКИ

У роботі вирішено науково–технічну задачу розробки ІТ обробки ЦЗ та відео з використанням операції згортки для ЦП архітектури ARM64 для підвищення швидкодії на сучасних одноплатних вбудованих системах (SBC). За результатами дисертаційних досліджень отримано такі загальні висновки:

1. Проведений аналіз існуючих програмно–апаратних та архітектурно-інформаційних засобів для забезпечення, підвищення та реалізації швидкодіючих обчислень, для ОЗ ЦЗ та ПЗ в цілому дозволив визначити потенційні напрями та підходи вдосконалення швидкодії для цілочисельних обчислень;
2. Отримано лінійні оператори для ШОЗ, котрі ефективно використовують 16-бітні обчислення, при обробці ЦЗ, а саме: субсмугова фільтрація (2.4, 2.5); контрастування (2.7, 2.8); цифрова стабілізація зображень (2.9, 2.10); рекурентне двократне збільшення масштабу (2.11, 2.12);
3. Запропоновано математичну модель ОЗ з використанням 16-бітних цілочисельних операцій для 8-бітних RGB ЦЗ та 8-бітних ядер згортки;
4. Запропоновано формалізовану нотацію (А.12–А.17) для опису векторизованих обчислень з використанням SIMD NEON64, яка дозволяє абстрагуватися від асемблерних інструкцій та представляти векторні SIMD операції у вигляді математичних/структурованих виразів, що сприяє кращій читабельності та знижує поріг входу, при засвоєнні технік експертної векторизації;
5. Розроблено та теоретично обґрунтовано метод швидкодіючої операції згортки для обробки 8-бітних RGB ЦЗ з використанням 16-бітних цілочисельних операцій, який забезпечує підвищення швидкодії на центральних процесорах архітектури ARM64 порівняно з аналогічним підходом на основі функції cv::filter2D(…) бібліотеки OpenCV:
    - для ядер згортки середнього розміру (8x8–11x11) досягнуто прискорення в діапазоні 3,1–11,1 раза для Orange Pi 5 Pro та 2,7–10,2 раза для Raspberry Pi 4;
    - для великих ядер згортки (12x12–15x15), де бібліотека OpenCV може використовувати FFT-алгоритми, запропонований метод демонструє прискорення 1,8–5,8 раза для Orange Pi 5 Pro та 1,7–5,2 раза для Raspberry Pi 4;

- для операцій масштабування зображень, що включають чотирикратне застосування операції згортки, розроблений підхід (zoomUP_2__NEON64) показує прискорення до 6,3 раза (Orange Pi 5 Pro) та до 6,2 раза (Raspberry Pi 4) для ядер середнього розміру порівняно з аналогічною операцією на базі cv::filter2D.

6. Розроблено та реалізовано алгоритм ШОЗ, проведено оптимізацію цього алгоритму використовуючи технологію NEON64 та експериментальні дослідження швидкодії розробленого алгоритму на різних платформах SBC (Raspberry Pi 4 та Orange Pi 5 Pro) для різних розмірів ЦЗ (від 256x144 до 3840x2160) та розмірів ЯЗ (від 2x2 до 15x15);
7. Розроблена ІТ (основою якої є ШОЗ) дозволяє в режимі реального часу проводити фільтрацію та масштабування потокового відео та ЦЗ на одноплатних комп'ютерах з ЦП ARM64, як–от: Raspberry Pi 4 та Orange Pi 5 Pro;
8. Запропонована методика експериментальної оцінки та порівняння швидкодії різних програмних реалізацій операції згортки на платформах з архітектурою ЦП ARM64 дозволила визначати переваги запропонованих алгоритмів на основі ШОЗ (4.1–4.9);
9. Проведено порівняльний аналіз швидкодії розробленого алгоритму з існуючими рішеннями, зокрема з функцією cv::filter2D бібліотеки OpenCV. У результаті дослідження встановлено, що запропонована інформаційна технологія демонструє значне прискорення виконання операції згортки порівняно з функцією cv::filter2D (бібліотеки OpenCV), досягаючи:
   - максимального прискорення для ядер згортки розміром 8–11 у 3,1–11,1 раза (Orange Pi 5 Pro) та 2,7–10,2 раза (Raspberry Pi 4);
   - помірного для ядер розміром 2–7 у 0,8–2,5 раза (Orange Pi 5 Pro) та 0,9–2,4 раза (Raspberry Pi 4) з урахуванням витрат вводу-виводу вимірювального конвеєра; за прямим вимірюванням самої операції згортки — у 1,3–2,8 раза;
   - стабільно високого для ядер розміром 12–15 у 1,8–5,8 раза (Orange Pi 5 Pro) та 1,7–5,2 раза (Raspberry Pi 4).

Розроблена технологія потенційно може мати кращу енергоефективність порівняно з реалізаціями на основі операцій з плаваючою комою завдяки використанню цілочисельних обчислень та скороченню часу виконання операцій.

Дослідження енергоспоживання розробленої ІТ на мобільних та вбудованих пристроях є перспективним напрямом подальших досліджень.

Особливо значущі переваги ІТ проявилися у сценаріях реального часу: наприклад, при обробці відеопотоку з частотою 25 FPS на платформі Orange Pi 5 Pro було досягнуто стабільної швидкодії 26–29 FPS для операції згортки для RGB ЦЗ.

10. Запропоновано наступні рекомендації щодо застосування оптимізованого алгоритму ОЗ в системах реального часу та вбудованих пристроях:
    - Використовувати цілочисельні 16-бітні обчислення як основний тип даних. Для забезпечення високої швидкодії рекомендовано використовувати саме 16-бітні цілочисельні типи (int16_t/uint16_t) замість 32-бітних або операцій з плаваючою комою. Це дозволяє повністю задіяти SIMD-регістр NEON64 та обробляти більше даних за один такт.
    - Застосовувати SIMD-інструкції NEON64 через intrinsics або Inline Assembly. Найбільший приріст швидкодії спостерігається при прямому використанні NEON-інструкцій, а не при покладанні на автоматичну векторизацію компілятора. Рекомендується використовувати інструкції vld1q_s16, vmlaq_s16, vst1q_s16 для реалізації згорткових обчислень.
    - Попередньо нормалізувати ядра згортки для запобігання переповненню. Під час виконання згортки на 16-бітних типах важливо гарантувати, що сума добутків не перевищить допустимий діапазон. Для цього варто нормалізувати коефіцієнти ядра або використовувати масштабування після згортки з насиченням (saturating arithmetic).
    - Оптимізувати доступ до пам’яті (cache-aware access). Рекомендується організовувати читання пікселів блоками (tile-based access), зберігаючи їх у локальних буферах, щоб зменшити кількість кеш-промахів. Це особливо важливо при згортці великих зображень або високочастотними ядрами.
    - Застосовувати фіксовані розміри ядер згортки (наприклад, 3×3, 5×5, 7×7). Для забезпечення стабільного розкладу SIMD-операцій рекомендовано заздалегідь фіксувати розміри ЯЗ. Це дозволяє уникнути розгалужень у коді й пришвидшує компіляцію та виконання.

- Застосовувати оптимізацію циклів: розгортання (loop unrolling) та блокування (blocking). Розгортання циклів (manual loop unrolling) дозволяє мінімізувати накладні витрати на керування циклами. Блокування забезпечує обробку кількох рядків за один прохід і зменшує звернення до пам’яті.
- Інтегрувати згорткове ядро через C-бібліотеку із зовнішніми застосунками (наприклад, Python). Для систем, де гнучкість вищого рівня є пріоритетною, доцільно застосовувати динамічні C-бібліотеки (.so/.dll) з обгортками для Python (через ctypes), що забезпечує баланс між продуктивністю та зручністю розгортання.
- У системах реального часу важливо уникати універсальних реалізацій із високим накладним навантаженням (overhead), якщо є можливість вбудувати власне ядро згортки з оптимізованим кодом. Бажано використовувати бібліотеки, які дозволяють контроль над SIMD-виконанням, або перевіряти, яку реалізацію (SIMD-оптимізовану чи стандартну) використовує обрана функція бібліотеки.
- Проводити профілювання та верифікацію швидкодії на цільових пристроях. Рекомендується використовувати інструменти профілювання (perf, gprof, Valgrind, або NEON intrinsics visualizers) безпосередньо на пристроях типу Raspberry Pi, Orange Pi або мобільних SoC, щоб виявити вузькі місця.
- Застосовувати розроблену ІТ для задач фільтрації, стабілізації, масштабування у режимі реального часу. Завдяки універсальності архітектури, реалізоване ядро можна використовувати як основний обчислювальний модуль у системах відеоспостереження, обробки даних з БПЛА, мобільної діагностики тощо.

Також слід відзначити, що матеріали дисертаційного дослідження використані для виконання науково-дослідних робіт на базі Національного університету «Київський авіаційний інститут», зокрема у таких темах подвійного призначення:

- НДР №1062–ДБ16 «Автоматизація розпізнавання та класифікації цільових об’єктів за відеоданими з камери безпілотного повітряного судна» (номер державної реєстрації 0116U004638);
- НДР №247–ДБ19 «Розроблення та виготовлення програмно-апаратних

засобів цільового навантаження для повітряного спостереження та альтернативної навігації літального апарату» (номер державної реєстрації 0119U100553);

- НДР №479–ДБ24 «Розроблення багатокритеріальної завадозахищеної навігаційної системи для безпілотних апаратів» (номер державної реєстрації 0124U000291).

Також слід зауважити, що матеріали дисертаційного дослідження використані для розробки навчальних курсів та лабораторних робіт з дисциплін (див. Додаток В):

- «Архітектура обчислювальних систем та операційні системи» для студентів спеціальності 113 «Прикладна математика»;
- «Системне програмування» для студентів спеціальності 113 «Прикладна математика»;
- «Програмування мовою асемблер» для студентів спеціальності 113 «Прикладна математика».

Також результати дисертаційного дослідження впроваджено у виробничу діяльність підприємств: ТОВ «ЕЛІСАТ» — у вбудованому програмному забезпеченні пристроїв збору й обробки даних з датчиків; ТОВ «ІНСПІРІТО ТІМ» — у програмно-апаратних засобах обробки відеоданих з бортових камер БПЛА, що підтверджено відповідними актами впровадження (див. Додаток Г та Додаток Д).

Запропоновані алгоритми швидкодіючої операції згортки можуть бути застосовані для прискорення ОЗ в згорткових нейронних мережах на мобільних та вбудованих пристроях, що є перспективним напрямом для розвитку мобільних додатків з елементами штучного інтелекту та систем постпроцесингу на платформах ARM64.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Rosen I.* Auto-vectorization in GCC. GCC, the GNU Compiler Collection. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html (date of access: 29.07.2025).
2. Auto-vectorization in LLVM. The LLVM Compiler Infrastructure Project. URL: https://llvm.org/docs/Vectorizers.html (date of access: 29.07.2025).
3. *Приставка П. О.*, *Шевченко А. К.* Дослідження реалізації лінійного оператора згортки цифрового зображення при 16-бітних обчисленнях. Актуальні проблеми автоматизації та інформаційних технологій. 2016. Т. 20: 1. С. 78–90. URL: https://doi.org/10.15421/431608 (дата звернення: 29.07.2025).
4. *Bryant R. E.*, *O'Hallaron D. R.* Computer systems: a programmer's perspective. 3rd ed. Boston : Pearson Education, 2015. 1120 p.
5. *Fog A.* Optimizing software in C++: an optimization guide for windows, linux and mac platforms. Copenhagen University College of Engineering, 2025. 179 p. URL: https://www.agner.org/optimize/optimizing_cpp.pdf (date of access: 29.07.2025).
6. Opencv: image filtering. OpenCV team. URL: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04 (date of access: 05.08.2025).
7. *Barbier A.* ARM-software/computelibrary: the compute library is a set of computer vision and machine learning functions optimised for both arm cpus and gpus using SIMD technologies. Arm-software. URL: https://github.com/ARM-software/ComputeLibrary (date of access: 30.07.2025).
8. *Shevchenko A.*, *Tymchyshyn V.* A SIMD-based approach to the enhancement of convolution operation performance. Proceedings of the international workshop on conflict management in global information networks (CMiGIN). 2019. Vol. 2588. P. 447–458. URL: https://ceur-ws.org/Vol-2588/paper37.pdf (date of access: 29.07.2025).

9. *Chellappa S.*, *Franchetti F.*, *Püschel M.* How to write fast numerical code: a small introduction. Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. Berlin, Heidelberg : Springer, 2008. P. 196–259. URL: https://doi.org/10.1007/978-3-540-88643-3_5 (date of access: 05.08.2025).
10. *Flynn M.* Very high-speed computing systems. Proceedings of the IEEE. 1966. Vol. 54: 12. P. 1901–1909. URL: https://doi.org/10.1109/PROC.1966.5273 (date of access: 29.07.2025).
11. *Flynn M. J.* Some computer organizations and their effectiveness. IEEE transactions on computers. 1972. Vol. C–21: 9. P. 948–960. URL: https://doi.org/10.1109/TC.1972.5009071 (date of access: 29.07.2025).
12. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and intel platforms. G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, J. Zhou. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. 2013. P. 1107–1116. URL: https://ieeexplore.ieee.org/abstract/document/6650996/ (date of access: 29.07.2025).
13. *Larsen S.*, *Amarasinghe S.* Exploiting superword level parallelism with multimedia instruction sets. Sigplan not. 2000. Vol. 35: 5. P. 145–156. URL: https://doi.org/10.1145/358438.349320 (date of access: 05.08.2025).
14. Arm architecture reference manual for a-profile architecture. Arm Developer. URL: https://developer.arm.com/documentation/ddi0487/latest (date of access: 29.07.2025).
15. ARM cortex-a series (armv7-a) programmer's guide. introducing NEON. Arm Developer. URL: https://developer.arm.com/documentation/den0013/0400/Introducing-NEON?lang=en (date of access: 12.09.2025).
16. Arm C language extensions. Arm Developer. URL: https://arm-software.github.io/acle/main/acle.html (date of access: 30.07.2025).

17. *V S. P.* Arm NEON optimization: a practical approach. ignitarium.com. 2021. URL: https://ignitarium.com/practical-approach-to-arm-neon-optimization/ (date of access: 05.08.2025).

18. *Pyeatt L. D.*, *Ughetta W.* ARM 64-bit assembly language. Cambridge : Newnes, 2019. 508 p.

19. *Pyeatt L. D.* Modern assembly language programming with the ARM processor. Amsterdam : Elsevier, 2024. 554 p.

20. Arm a-profile a64 instruction set architecture. a64 – SIMD and floating-point instructions (alphabetic order). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions (date of access: 29.07.2025).

21. Arm neon intrinsics reference. Arm Developer. URL: https://arm-software.github.io/acle/neon_intrinsics/advsimd.html (date of access: 30.07.2025).

22. *Shevchenko A.*, *Prystavka P.*, *Tymchyshyn V.* Research on possible convolution operation speed enhancement via AArch64 SIMD. Advances in computer science for engineering and education. Lecture Notes on Data Engineering and Communications Technologies. Cham : Springer International Publishing, 2022. Vol. 134. P. 61–75. URL: https://doi.org/10.1007/978-3-031-04812-8_6.

23. *Шевченко А. К.* Вибір програмно-апаратних засобів для реалізації операції згортки з підвищеною швидкістю. Наукоємні технології. 2018. Т. 37: 1. С. 49–54. URL: https://doi.org/10.18372/2310-5461.37.12369 (дата звернення: 29.07.2025).

24. Intel® 64 and IA-32. architectures optimization reference manual. Intel. URL: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html (date of access: 30.07.2025).

25. *Hoey J. V.* Beginning x64 assembly programming: from novice to AVX professional. Berkeley : Apress, 2019. 413 p.

26. AMD64 architecture programmer's manual. volume 5: 64-bit media and x87 floating-point instructions. Advanced Micro Devices. 2002. URL: https://kib.kiev.ua/x86docs/AMD/AMD64/26569_APM_v5-r3.15.pdf (date of access: 12.09.2025).

27. *Lomont C.* Introduction to intel advanced vector extensions. URL: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf (date of access: 29.07.2025).

28. Intel® intrinsics guide. Intel. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (date of access: 05.08.2025).

29. CUDA C++ programming guide. NVIDIA. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (date of access: 05.08.2025).

30. The OpenCL™ specification. Khronos Group. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_introduction (date of access: 05.08.2025).

31. *Kirk D. B.*, *Hwu W.-m. W.* Programming massively parallel processors: a hands-on approach. Waltham : Newnes, 2012. 519 p.

32. Hipacc: a domain-specific language and compiler for image processing. R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, W. Eckert. Ieee transactions on parallel and distributed systems. 2016. Vol. 27: 1. P. 210–224. URL: https://doi.org/10.1109/TPDS.2015.2394802 (date of access: 13.09.2025).

33. Convergence and scalarization for data-parallel architectures. Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, K. Asanović. Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2013. P. 1–11. URL: https://doi.org/10.1109/CGO.2013.6494995 (date of access: 05.08.2025).

34. The qualcomm® hexagon™ SDK. Qualcomm. URL: https://docs.qualcomm.com/bundle/publicresource/topics/80-77512-1/hexagon-dsp-sdk-collection-landing-page.html?product=1601111740010422 (date of access: 05.08.2025).

35. Hexagonal convolutional neural network for spatial transcriptomics classification. J. Gao, F. Zhang, K. Hu, X. Cui. 2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). 2022. P. 200–205. URL: https://doi.org/10.1109/BIBM55620.2022.9995701 (date of access: 12.09.2025).

36. Qualcomm FastCV SDK. Qualcomm. URL: https://www.qualcomm.com/developer/software/qualcomm-fastcv-sdk (date of access: 12.09.2025).

37. *Lattner C.*, *Adve V.* LLVM: a compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004. CGO 2004. 2004. P. 75–86. URL: https://doi.org/10.1109/CGO.2004.1281665 (date of access: 05.08.2025).

38. *Naishlos D.* Autovectorization in GCC. Proceedings of the 2004 GCC developers' summit. 2004. P. 105–118. URL: https://people.redhat.com/lockhart/.gcc2004/MasterGCC-2side.pdf#page=105 (date of access: 16.09.2025).

39. GCC, the GNU compiler collection - GNU project. GNU Project. URL: https://gcc.gnu.org/ (date of access: 13.09.2025).

40. Compilers: principles, techniques, and tools. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. Addison-Wesley, 2007. 1009 p.

41. Comparison of vectorization capabilities of different compilers for x86 and ARM cpus. N. Sakib, T. Prabhu, N. Santhi, J. Shalf, A.-H. A. Badawy. 2024 IEEE High Performance Extreme Computing Conference (HPEC). 2024. P. 1–7. URL: https://doi.org/10.1109/HPEC62836.2024.10938481 (date of access: 13.09.2025).

42. *Botor T.*, *Habiballa H.* Comparison of compilers efficiency in C++. Aip conference proceedings. 2018. Vol. 2040: 1. P. 030005. URL: https://doi.org/10.1063/1.5079068 (date of access: 24.09.2025).

43. Performance comparison of gcc and llvm on the eisc processor. C. Park, M. Han, H. Lee, S. W. Kim. 2014 international conference on electronics, information and communications (ICEIC). 2014. P. 1–2.

44. Comparison of LLVM and GCC on the ARM platform. J.-J. Kim, S.-Y. Lee, S.-M. Moon, S. Kim. 2010 5th international conference on embedded and multimedia computing. 2010. P. 1–6.

45. Opencv - open computer vision library. OpenCV team. URL: https://opencv.org/ (date of access: 05.08.2025).

46. *Bradski G.*, *Kaehler A.* Learning OpenCV: computer vision with the OpenCV library. O'Reilly Media, 2008. 579 p.

47. Valgrind. callgrind: a call-graph generating cache and branch prediction profiler. Valgrind Developers. URL: https://valgrind.org/docs/manual/cl-manual.html (date of access: 05.08.2025).

48. Valgrind. massif: a heap profiler. Valgrind Developers. URL: https://valgrind.org/docs/manual/ms-manual.html (date of access: 05.08.2025).

49. *Munro J.*, *Boldyreff C.*, *Capiluppi A.* Architectural studies of games engines—the quake series. 2009 international IEEE consumer electronics society's games innovations conference. 2009. P. 246–255.

50. *Trenholme D.*, *Smith S. P.* Computer game engines for developing first-person virtual environments. Virtual reality. 2008. Vol. 12: 3. P. 181–187.

51. *IEEE.* IEEE standard for floating-point arithmetic. IEEE std 754-2019 (revision of IEEE 754-2008). 2019. P. 1–84. URL: https://doi.org/10.1109/IEEESTD.2019.8766229.

52. Low-precision floating-point formats: from general-purpose to application-specific. A. Sabbagh Molahosseini, L. Sousa, A. A. Emrani Zarandi, H. Vandierendonck. Approximate computing. 2012. P. 77–98.

53. *Padgett W. T.*, *Anderson D. V.* Fixed-point signal processing. Vol. 9. Morgan & Claypool Publishers, 2009.

54. *Oberstar E. L.* Fixed-point representation & fractional math. Oberstar Consulting. 2007. URL: http://darcy.rsgc.on.ca/ACES/ICE4M/FixedPoint/FixedPointRepresentationFractionalMath.pdf (date of access: 13.09.2025).

55. *Dongarra J. J.*, *Hinds A. R.* Unrolling loops in fortran. Software: practice and experience. 1979. Vol. 9: 3. P. 219–226. URL: https://doi.org/10.1002/spe.4380090307 (date of access: 05.08.2025).

56. *Sarkar V.* Optimized unrolling of nested loops. Proceedings of the 14th international conference on Supercomputing. ICS '00. 2000. С. 153–166. URL: https://doi.org/10.1145/335231.335246 (дата звернення: 29.07.2025).

57. ARM cortex-a (ARMv7-a) series programmer's guide. compiler-optimizations: loop-unrolling. Arm Developer. URL: https://developer.arm.com/documentation/den0013/d/Optimizing-Code-to-Run-on-ARM-Processors/Compiler-optimizations/Loop-unrolling (date of access: 05.08.2025).

58. *Clark N.*, *Hormati A.*, *Mahlke S.* VEAL: virtualized execution accelerator for loops. Sigarch comput. archit. news. 2008. Vol. 36: 3. P. 389–400. URL: https://doi.org/10.1145/1394608.1382155 (date of access: 05.08.2025).

59. *Hennessy J. L.*, *Patterson D. A.*, *Asanović K.* Computer architecture: a quantitative approach. 5th ed. Waltham : Morgan Kaufmann/Elsevier, 2012. 858 p.

60. *Jacob B.*, *Wang D.*, *Ng S.* Memory systems: cache, DRAM, disk. Burlington : Morgan Kaufmann, 2010. 1017 p.

61. *Kelefouras V.*, *Keramidas G.* Design and implementation of deep learning 2d convolutions on modern cpus. IEEE trans. parallel distrib. syst. 2023. Vol. 34: 12. P. 3104–3116. URL: https://doi.org/10.1109/TPDS.2023.3322037 (date of access: 05.08.2025).

62. *Шевченко А. К.* Підвищення швидкодії операції згортки RGB зображень з використанням алгоритму кластеризації ядер згортки для архітектури процесорів ARM64. Problems of informatization and management. 2025. Т. 1: 81. С. 140–149. URL: https://doi.org/10.18372/2073-4751.81.20144 (дата звернення: 18.08.2025).

63. *Prystavka P.*, *Shevchenko A.*, *Rokitianska I.* Comparative analysis of detector-tracker architecture for object tracking based on SBC for UAV. 2024 IEEE 7th international conference on actual problems of unmanned aerial vehicles

development (APUAVD) (22–24 October 2024). 2024. P. 175–178. URL: https://doi.org/10.1109/APUAVD64488.2024.10765890 (date of access: 29.07.2025).

64. *Brkayy*. Optimizing matrix operations with NEON intrinsics and OpenMP: a practical guide. Medium. 2025. URL: https://medium.com/@brkayy987/optimizing-matrix-operations-with-neon-intrinsics-and-openmp-a-practical-guide-1c3088df5f21 (date of access: 12.09.2025).

65. *Jiang X.*, *Lin W.*, *Nagda A.* Optimizing matrix multiplication on NERSC's high performance computer cori. Proceedings of the Future Technologies Conference (FTC) 2023, Volume 3. 2023. P. 563–572. URL: https://doi.org/10.1007/978-3-031-47457-6_37.

66. *Kelefouras V.*, *Keramidas G.* Design and implementation of 2d convolution on x86/x64 processors. IEEE transactions on parallel and distributed systems. 2022. Vol. 33: 12. P. 3800–3815. URL: https://doi.org/10.1109/TPDS.2022.3171471 (date of access: 05.08.2025).

67. Opencv: vectorizing your code using universal intrinsics. OpenCV team. URL: https://docs.opencv.org/4.x/d6/dd1/tutorial_univ_intrin.html (date of access: 05.08.2025).

68. Opencv: vectorizing your code using universal intrinsics. 2-d convolution. OpenCV team. URL: https://docs.opencv.org/4.x/d6/dd1/tutorial_univ_intrin.html#autotoc_md367 (date of access: 13.09.2025).

69. *Gonzalez R. C.*, *Woods R. E.* Digital image processing (3rd edition). USA : Prentice-Hall, Inc., 2006.

70. TMS320C64x DSP library programmer's reference. Texas Instruments. 2003.

71. *Kormanyos C.* Real-time C++ : Efficient object-oriented and template microcontroller programming. 3rd ed. Berlin, Heidelberg : Springer, 2018. 426 p. URL: https://doi.org/10.1007/978-3-662-56718-0.

72. *Приставка П.* Побудова контрастних фільтрів за використанням поліноміальних сплайнів. Актуальні проблеми автоматизації та інформаційних технологій. 2007. Т. 11. С. 15–22.

73. *Приставка П.* Лінійні комбінації в-сплайнів, близькі до інтерполяційних у середньому, в задачі моделювання аналогових сигналів. Актуальні проблеми автоматизації та інформаційних технологій. 2011. Т. 15. С. 4–17.

74. *Приставка П.*, *Чолишкіна О.* Піраміда зображень на основі сплайн-моделі у вигляді лінійної комбінації в-сплайнів. Інформаційні технології та суспільство. 2021. 1 (1). С. 34–42. URL: https://doi.org/10.32689/maup.it.2021.1.4 (дата звернення: 04.09.2025).

75. *Castleman K. R.* Digital image processing. 2nd ed. Englewood Cliffs, N.J. : Prentice Hall, 1996. 667 p.

76. *Приставка П.* Застосування комбінованих фільтрів на основі поліноміальних сплайнів при обробці растрових зображень. Вісник нау. 2008. 4. С. 104–107.

77. *Приставка П.* Поповнення зі згладжуванням послідовностей відліків функцій двох змінних на основі сплайнів. Математичне моделювання. 2008. 1(18). С. 9–12.

78. *Приставка П.*, *Чолишкіна О.* Дослідження комбінованих фільтрів для підвищення різкості зображень. Актуальні проблеми автоматизації та інформаційних технологій. 2009. Т. 13. С. 39–53.

79. *Приставка П. О.* Поліноміальні сплайни в задачах обробки цифрових зображень. Дніпропетровськ : ДНУ, 2012. 264 с.

80. NEON is the new black: fast JPEG optimization on ARM server. The Cloudflare Blog. 2018. URL: https://blog.cloudflare.com/neon-is-the-new-black/ (date of access: 05.08.2025).

81. *Shevchenko A.*, *Prystavka P.* Enhancement of convolution operation performance using SIMD of aarch64. Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II. CEUR Workshop Proceedings. 2024. Vol. 3826. P. 385–391. URL: https://ceur-ws.org/Vol-3826/#short32 (date of access: 29.07.2025).

82. *Liu Z.-G.*, *Mattina M.* Efficient residue number system based winograd convolution. European conference on computer vision. 2020. P. 53–68.

83. Winograd convolution for deep neural networks: efficient point selection. S. A. Alam, A. Anderson, B. Barabasz, D. Gregg. Acm transactions on embedded computing systems. 2022. Vol. 21: 6. P. 1–28.

84. *Cardoso J. M. P.*, *Coutinho J. G. d. F.*, *Diniz P. C.* Embedded computing for high performance: efficient mapping of computations using customization, code transformations and compilation. Elsevier Science & Technology Books, 2017. 322 p.

85. Introduction to parallel computing tutorial. Lawrence Livermore National Laboratory. URL: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial (date of access: 13.09.2025).

86. Flynn's taxonomy and classification of parallel systems. Fiveable. URL: https://fiveable.me/parallel-and-distributed-computing/unit-2/flynns-taxonomy-classification-parallel-systems/study-guide/Ohzf44x4HCtFZRjK (date of access: 13.09.2025).

87. Ctypes — a foreign function library for python. Python documentation. URL: https://docs.python.org/3/library/ctypes.html (date of access: 05.08.2025).

88. Numpy documentation — NumPy v2.3 manual. NumPy Developers. URL: https://numpy.org/doc/stable/ (date of access: 13.09.2025).

89. *Pratt W. K.* Introduction to digital image processing. Boca Raton : CRC Press, 2013. 750 p.

90. Argparse — parser for command-line options, arguments and subcommands. Python documentation. URL: https://docs.python.org/3/library/argparse.html (date of access: 05.08.2025).

91. Scikit-image: image processing in python. S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, [et al.]. Peerj. 2014. Vol. 2. e453. URL: https://doi.org/10.7717/peerj.453.

92. *Kuzmiakova A.* Concurrent, parallel and distributed computing. Arcler Education Incorporated, 2022. 258 p.

93. Benchmarking object detection deep learning models in embedded devices. D. Cantero, I. Esnaola-Gonzalez, J. Miguel-Alonso, E. Jauregi. Sensors. 2022. Vol.

22: 11: 11. P. 4205. URL: https://doi.org/10.3390/s22114205 (date of access: 05.08.2025).

94. Orange pi - orangepi. Orange Pi. URL: http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-5-Pro.html (date of access: 05.08.2025).
95. Raspberry pi 4 model b. Raspberry Pi Ltd. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ (date of access: 29.07.2025).
96. BCM2711 ARM peripherals. Raspberry Pi Ltd. URL: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf (date of access: 05.08.2025).
97. Clang C language family frontend for LLVM. LLVM Project. URL: https://clang.llvm.org/ (date of access: 13.09.2025).
98. About QEMU — QEMU documentation. QEMU Project. URL: https://www.qemu.org/docs/master/about/index.html (date of access: 05.08.2025).
99. Mlperf power: benchmarking the energy efficiency of machine learning systems from μwatts to mwatts for sustainable AI. A. Tschand, A. T. R. Rajan, S. Idgunji, A. Ghosh, J. Holleman, C. Kiraly, P. Ambalkar, R. Borkar, R. Chukka, T. Cockrell, [et al.]. 2025 IEEE international symposium on high performance computer architecture (HPCA). 2025. P. 1201–1216.
100. Benchmarking energy and latency in tinyml: a novel method for resource-constrained ai. P. Bartoli, C. Veronesi, A. Giudici, D. Siorpaes, D. Trojaniello, F. Zappa. 2025 international joint conference on neural networks (IJCNN). 2025. P. 1–8.

ДОДАТОК А

ФОРМАЛІЗОВАНА НОТАЦІЯ ВЕКТОРИЗОВАНИХ ОБЧИСЛЕНЬ ТА АЛГОРИТМ ШВИДКОДІЮЧОЇ ОПЕРАЦІЇ ЗГОРТКИ НА ЇЇ ОСНОВІ

У цьому додатку формалізовано ОЗ для цілочисельних обчислень (підрозд. А.1) та описано організацію векторизованих і завершальних обчислень ОЗ (підрозд. А.2) для основного алгоритму ШОЗ, який було векторизовано за допомогою inline-вставок C/C++, використовуючи діалект асемблера ARM NEON64 SIMD. Відсутність формальної мови запису атомарних асемблерних операцій зумовила необхідність розробки спеціалізованої нотації, яку запропоновано в підрозд. А.3 для узагальнення, спрощення і формалізації ядра обчислень ІТ. На основі цієї нотації описано обчислення векторизованого скалярного добутку (підрозд. А.4), нормалізацію ВСД (підрозд. А.5) та надано детальну блок-схему обчислення ШОЗ для цілочисельних ЯЗ (підрозд. А.6). Описано варіант оптимізації ШОЗ за наявності нульових елементів у складі ЯЗ на основі кластеризації (підрозд. А.7). Для подальшого підвищення швидкодії ШОЗ описано кілька варіантів модифікації обчислення ВСД та роботи з ЯЗ: оптимізацію за рахунок використання пари switch–continue (підрозд. А.8), алгоритм конвертування від'ємних значень ЯЗ у беззнакові числа (підрозд. А.9) та кеш-френдлі підхід до обробки ЦЗ (підрозд. А.10). Наприкінці додатка сформульовано висновки (підрозд. А.11), що підсумовують запропоновані підходи до реалізації ЯІТ, яке спирається на поєднання ШОЗ із ЯЗ, що містять 8-бітні елементи (підрозд. А.4–А.6), з оптимізацією за наявності нульових елементів (підрозд. А.7) та використанням конструкції switch–continue (підрозд. А.8), за умови використання предикт-матриці.

А.1 Формалізація операції згортки для цілочисельних обчислень.

Базовим виразом даного дослідження є (2.1), котре описує ОЗ. Для фактичної реалізації ОЗ необхідно провести модифікацію рівняння (2.1), а саме переміщення орієнтації індексів рівняння до «0-індексованого вигляду», зміну імен та введення стилістики «самодокументованого» програмування у математичні вирази, через неможливість використовувати латинські літери та «сабскрипт» індексування. Без зміни змісту (2.1), лише із зміною назв та використанням «0-індексації», отримуються наступні твердження:

Твердження А.1: нехай src_{src_i,src_j} та dst_{dst_i,dst_j} будуть елементами відповідних двовимірних векторів (матриць) *SRC* та *DST*, де: $src_i = \overline{0, src_rows - 1}$ і $dst_i = \overline{0, dst_rows - 1}$, у котрих *src_rows* та *dst_rows* відповідно позначають кількість рядків у матрицях *SRC* та *DST*; $dst_j = \overline{0, dst_cols - 1}$ і $src_j = \overline{0, src_cols - 1}$, у котрих *src_cols* та *dst_cols* відповідно позначають кількість стовпців у матрицях *SRC* та *DST*.

Твердження А.2: нехай матриці *SRC* та *DST* є однаковими за розмірами, а кожен їх елемент src_{src_i,src_j} та dst_{dst_i,dst_j} містить трійку значень, що відповідають інтенсивностям кольорових каналів RGB:

$$\begin{aligned} src_{src_i,src_j} &= (R_{src_i,src_j}, G_{src_i,src_j}, B_{src_i,src_j}), \\ dst_{dst_i,dst_j} &= (R_{dst_i,dst_j}, G_{dst_i,dst_j}, B_{dst_i,dst_j}), \end{aligned} \tag{A.1}$$

де *R*, *G*, і *B* представляють інтенсивності червоного, зеленого, та блакитного каналів ЦЗ відповідно. Усі елементи src_{src_i,src_j} та dst_{dst_i,dst_j}, містять 8-бітні значення:

$$\begin{aligned} R_{dst_i,dst_j} &\in [0; 0\text{xFF}], \\ G_{dst_i,dst_j} &\in [0; 0\text{xFF}], \\ B_{dst_i,dst_j} &\in [0; 0\text{xFF}]. \end{aligned} \tag{A.2}$$

Твердження А.3: нехай flt_{flt_i,flt_j} є елементами відповідного двовимірного вектора (матриці) *flt*, де: $flt_i = \overline{0, flt_rows - 1}$, де *flt_rows* відповідно позначають кількість рядків у матриці *flt*; $flt_j = \overline{0, flt_cols - 1}$, де *flt_cols* відповідно позначають кількість стовпців у матриці *flt*. Отже можна переписати ОЗ (2.1) у вигляді виразу:

$$\begin{aligned} &dst_{src_i+\left\lfloor \frac{flt_rows}{2} \right\rfloor, src_j+\left\lfloor \frac{flt_cols}{2} \right\rfloor} = \\ &\sum_{flt_i=0}^{flt_rows-1} \sum_{flt_j=0}^{flt_cols-1} src_{src_i+flt_i,src_j+flt_j} \cdot flt_{flt_i,flt_j} \end{aligned} \tag{A.3}$$

де, матриця вхідного ЦЗ *SRC* має наступні діапазони значень індексів $src_i = \overline{0, src_rows - flt_rows}$ та $src_j = \overline{0, src_cols - flt_cols}$. Також у (А.3) використовується взаємопов'язаність індексів матриць *SRC* та *DST*. Таким чином, рівняння (А.3) є аналогом/продовженням основного рівняння (2.1) [1].

Для спрощення рівняння (А.3) до рівня цілочисельних операцій, котрі використовуються здебільшого для спрощення обчислювань та підвищують

швидкодію ОЗ, що було неодноразово продемонстровано рівняннями (2.4–2.7), можна перетворити (А.3) у наступне рівняння:

$$dst_{src_i+\left\lfloor\frac{flt_rows}{2}\right\rfloor,src_j+\left\lfloor\frac{flt_cols}{2}\right\rfloor} = flt_div \cdot \left(\sum_{flt_i=0}^{flt_rows-1} \sum_{flt_j=0}^{flt_cols-1} src_{src_i+flt_i,src_j+flt_j} \times flt_{flt_i,flt_j} \right) \quad (A.4)$$

де, flt_div має тип *float*. Дужки введені для позначення першочерговості виконання операцій [2–4].

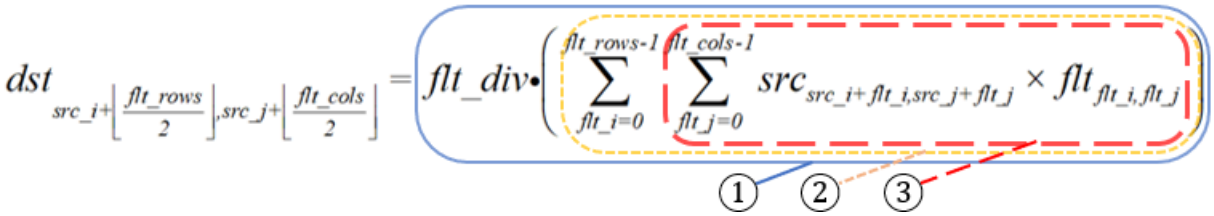


Рисунок А.1 — Схематичне представлення послідовності обрахування ОЗ (А.4) розбиття: (1) – червоний (внутрішня довга штрихова лінія); (2) – жовтий (коротка штрихова лінія) і (3) – синій (суцільна лінія).

Представлену форму ОЗ (А.4) та (рис. А.1) , з використанням цілочисельних операцій можна розділити на декілька операцій, що позначені (обведені) різними кольорами ліній:

1. Червоною лінією на (рис. А.1) позначено операцію розрахунку скалярного добутку для src_j-ї строки ЯЗ. Ця частина виразу (А.4) є найбільш навантаженою за швидкодією, що призводить до найбільш вагомих втрат швидкодії ОЗ ЦЗ на практиці. За умови підвищення швидкодії коду, який реалізує цю частину виразу, ОЗ буде значно пришвидшена. Саме тому ця частина зазнала значної NEON64 векторизації, розгортання циклу та максимізації локалізації пам’яті (через розміщення даних рядка ЦЗ у векторних регістрах);
2. Зовнішні цикли (частина виразу, що виділена оранжевою лінією) - фіналізують реалізацію обрахунку ОЗ, не враховуючи етап нормалізації. Для пришвидшення швидкодії даної частини коду рекомендується застосувати розгортання циклу, що було впроваджено на практиці;
3. Нормалізація обчислень (виділено синьою суцільною лінією) за допомо-

гою коефіцієнта нормалізації flt_div, що істотно не впливає на швидкодію коду.

Таким чином, виконавши/врахувавши всі ці кроки та здійснивши векторизацію з використанням технології NEON64, було реалізовано ОЗ ЦЗ (RGB – 8-біт). Усі подальші підрозділи будуть алгоритмічно описувати процес векторизації.

А.2 Організація векторизованих та завершальних обчислень операції згортки.

Для підвищення швидкодії ОЗ (А.4) є необхідним і логічним використання векторизованих SIMD-обчислень, оскільки традиційні скалярні обчислення не забезпечують достатньої швидкодії для обробки ЦЗ у режимі реального часу на платформах ARM64. Векторизовані SIMD-обчислення дозволяють зробити це за одну машинну інструкцію (умовно), що значно підвищує швидкодію ОЗ. Однак повна векторизація виразу (А.4) не завжди можлива через архітектурні обмеження SIMD NEON64, які полягають у фіксованій ширині векторних регістрів (128 біт) та необхідності вирівнювання даних. Розміри ЦЗ та ЯЗ можуть не бути кратними ширині векторних регістрів, що призводить до необхідності обробки ”залишкових”елементів ЦЗ. З цього слідує, що вираз (А.4) має бути розділено на дві частини: векторизовану частину, що виконує основний обсяг ОЗ ЦЗ за допомогою SIMD NEON64 обчислень з максимальною ефективністю, та завершальну частину, що обробляє залишкові елементи скалярними операціями. Така декомпозиція ОЗ забезпечує оптимальне використання обчислювальних ресурсів ЦП та гарантує коректність результатів для ЦЗ довільного розміру.

Процес векторизації коду складається з декількох кроків, але найважливішим є крок визначення кількості векторних регістрів, що використовуються при акті завантаження. Це є важливим, оскільки часто необхідно вийти за межі 8-бітних обчислень за рахунок збільшення бітового поля та використання замість одного векторного регістра двох (у випадку 16-бітних обчислень) чи чотирьох (у випадку 32-бітних обчислень). У даному дослідженні використовуються шість векторних регістрів для завантаження/збереження базової частини рядка ЦЗ розміром в 32 8-бітних трійок RGB. Це призводить, наприкінці, до стабільно зайнятих 24 векторних регістрів. З цього випливає, що **кожен канал ЦЗ займає**

на початку обчислень 2 векторних регістра, які згодом ”розкриваються”у 8 векторних регістрів для підтримки 32-бітних операцій. Таким чином, можна обрахувати кількість ітерацій обчислень ШОЗ ЦЗ, знаючи: розміри ЦЗ; розміри ЯЗ; базову кількість векторів для збереження одного каналу ЦЗ – 2, що вміщує в себе 32 елементи одного каналу ЦЗ. Кількість векторизованих ОЗ для одного рядка ЦЗ визначається наступним чином:

$$NC_vec = \left\lfloor \frac{src_cols - flt_cols + 1}{32} \right\rfloor, \tag{A.5}$$

де $\lfloor \bullet \rfloor$- округлення до цілого вниз, NC_vec - від англ. «number of columns that were vectorized». А діапазон таких обрахунків для одного рядка ЦЗ можна записати так:

$$st32N = \overline{0, NC_vec - 1}, \tag{A.6}$$

Знаючи NC_vec (А.5) можна обрахувати загальну кількість саме векторізованих ОЗ для всього ЦЗ:

$$N_vec = NC_vec \cdot NR_vec, \tag{A.7}$$

де $NR_vec = src_rows - flt_rows + 1$ (від англ. «number of rows»).

Використовуючи (А.5–А.7) можна записати (А.4) у вигляді, що описує обчислення тільки векторизованої частини ОЗ ЦЗ:

$$dst_{src_i+\left\lfloor \frac{flt_rows}{2} \right\rfloor, idx_{base}(st32N)+j+\left\lfloor \frac{flt_cols}{2} \right\rfloor} = flt_div \cdot \\ \cdot \left(\sum_{flt_i=0}^{flt_rows-1} \sum_{flt_j=0}^{flt_cols-1} src_{src_i+flt_i,\, idx_{base}(st32N)+j+flt_j} \cdot flt_{flt_i, flt_j} \right) \tag{A.8}$$

де: $idx_{base}(st32N) = 32 \cdot st32N$, $j = \overline{0, 31}$.

По суті, для даного дослідження вже сформульовано завдання для проведення векторізації (А.8), котре можна розбити на дві частини:

На (рис. А.2) схематично представлено обчислювальне ядро ШОЗ (А.8), що містить дві ключові складові: обрахунок ВСД для кожного рядка ЯЗ (позначено жовтою короткою штриховою лінією), та обрахунок ВСД для кожного стовпця ЯЗ (позначено червоною довгою штриховою лінією). Схема не відображає операцію

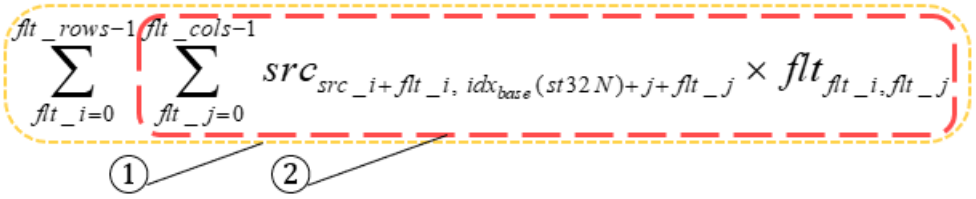


Рисунок А.2 — Схематичне представлення обчислювального ядра ВОЗ: (1) – ВСД по рядку ЯЗ (червона штрихова лінія); (2) – ВСД по стовпцю ЯЗ (жовта штрихова лінія).

нормалізації (множення на *flt_div*), оскільки ця операція має порівняно незначний вплив на загальну обчислювальну складність алгоритму, хоча й є необхідною для коректного завершення ВОЗ. Таким чином, основна частина ЯІТ - ШОЗ - має своє обчислювальне ядро, формалізоване рівнянням (А.8) та візуалізоване на (рис. А.2). Подальші описи ШОЗ будуть сфокусовані на ефективній векторизації цього ядра обчислень з використанням технології NEON64.

Щодо завершальних обчислень ОЗ ЦЗ, вони виконуються за допомогою стандартної мови C/C++. Базуючись на (А.5), формула кількості кроків дообрахування ОЗ ЦЗ для одного рядка, використовуючи мову C/C++ (без мануальної/штучної векторизації), записується наступним чином:

$$NC_rest = (src_cols - flt_cols + 1) - NC_vec \cdot 32. \tag{A.9}$$

На основі виразів (А.5) та (А.9), діапазон дообрахунку для одного рядка ЦЗ записується як:

$$src_jrest = \overline{NC_vec \cdot 32,\ src_cols - flt_cols}. \tag{A.10}$$

На основі (А.5) та (А.10) можна записати вираз дообрахування частини ОЗ ЦЗ, що не була векторизована:

$$dst_{src_i+\left\lfloor \frac{flt_rows}{2} \right\rfloor, src_jrest+\left\lfloor \frac{flt_cols}{2} \right\rfloor} = flt_div \cdot \left(\sum_{flt_i=0}^{flt_rows-1} \sum_{flt_j=0}^{flt_cols-1} src_{src_i+flt_i, src_jrest+flt_j} \times flt_{flt_i,flt_j} \right). \tag{A.11}$$

Таким чином можна попередньо описати алгоритм обчислення ОЗ ЦЗ у вигляді блок-схеми на основі (А.4) використовуючи кількість векторізованих

ОЗ (ВОЗ) для одного рядка ЦЗ (А.5) і знаючи кількість дообрахувань (А.9) ОЗ, використавши показник (А.7):

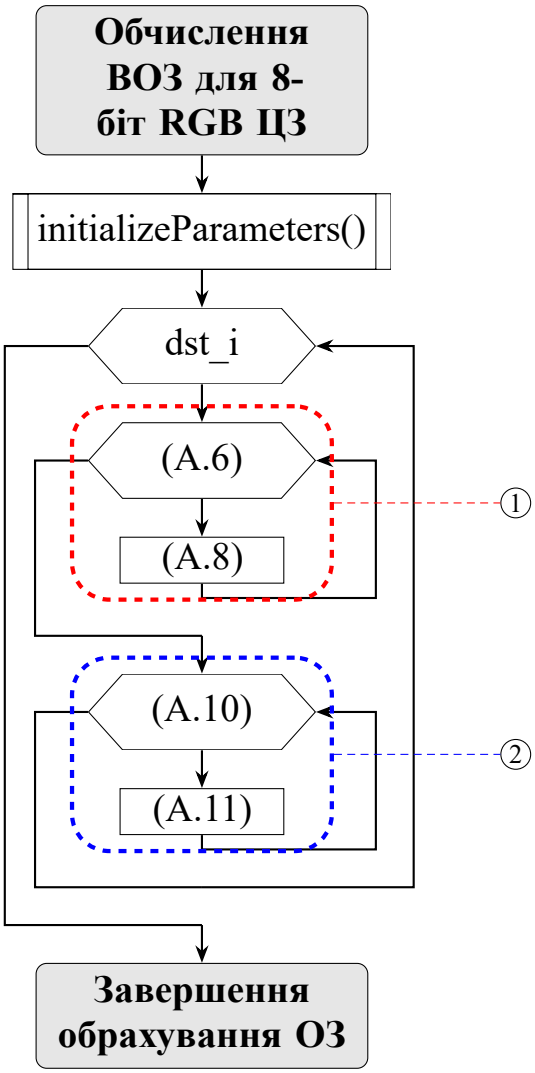


Рисунок А.3 — Блок-схема алгоритму обчислення векторізованої (NEON64) ОЗ для цілочисельного ЯЗ flt_{flt_i,flt_j} та RGB ЦЗ, де: (1) – векторізована частина обрахування ОЗ ЦЗ (найбільш навантажена частина ОЗ); (2) – дообрахування частини ОЗ ЦЗ мовою C++.

На (рис. А.3) зображено ШОЗ 8-біт RGB ЦЗ, яка складається з двох частин і пронумеровано відповідно: (1) - частина є найбільш ресурсозатратною у контексті швидкодії і описується рівнянням (А.8); (2) - це дообрахування частини ОЗ використовуючи суто C/C++ код, що не використовує спеціалізованих технологій окрім кеш-френдлі стилю написання коду для ще більшого підвищення швидкодії ОЗ ЦЗ.

Таким чином була сформована та описана база для опису ЯІТ – (А.5–А.11). ЯІТ теж має своє «ядро», яке описане рівнянням (А.8), і надалі весь опис ШОЗ спирається саме на вираз (А.8), точніше це можна назвати векторизуванням

рівняння виразу (А.8) за допомогою NEON64.

А.3 Базові назви векторів (NEON64 регістрів) та опис індексування їх елементів.

Для спрощення сприйняття подальшого опису ШОЗ ЦЗ, що спирається на векторні цілочисельні операції, необхідно зазначити, що опис ШОЗ (А.11) спирається на підхід ”самодокументованого”програмування [5], тобто на використання таких імен змінних та структур даних, котрі однозначно відображають їх призначення та зміст.

Основні змінні позначаються префіксом ”V_”або ”v_— це ім’я вектора або його елементів відповідно. Для формалізації запису векторних регістрів NEON64 вводиться наступна математична нотація:

$$VC_kI_e{}^{dtype} = \{vC_kI_{e,\,i}{}^{dtype} | i \in \mathcal{I}^{dtype}_{start,end}\}, \tag{А.12}$$

де:

1. $C_k \in C = \{r, g, b\}, k = \overline{0, 2}$ - позначення каналу кольору ЦЗ;
2. $I_e \in I = \{0007, 0815, 1623, 2431\}, e = \overline{0, 3}$ - впорядкована множина іменування діапазону індексів для вказівника, від котрого виконується завантаження елементів кольорових каналів ЦЗ (дана множина може змінювати свої значення в подальших описах рівнянь);
3. $dtype \in \{u8, u16, u32, s8, s16, s32, f32\}$ - тип даних елементів NEON64 регістрів;
4. $\mathcal{I}^{dtype}_{start,end} = \{i \mid i = \overline{start, end}\}$ - впорядкована множина індексів елементів NEON64 регістрів.

Так, представлена нотація дозволяє формалізувати іменування векторних регістрів NEON64, що використовуються для зберігання проміжних результатів обрахування ШОЗ для різних каналів кольору ЦЗ та різних типів даних. Вона також забезпечує гнучкість у визначенні діапазонів індексів елементів векторних регістрів, що є важливим для оптимізації алгоритму та адаптації його до різних архітектур ЦП в подальшому [5].

Для формалізації залежності кількості елементів від типу даних вводимо функцію *elems*, яка відображає тип даних у кількість елементів, що можуть

зберігатися у 128-бітному векторному регістрі NEON64:

$$elems : dtype \mapsto \mathbb{N}, \quad elems(dtype) = \frac{128}{size(dtype)}, \tag{A.13}$$

де: $size(dtype)$ - функція, що повертає розмір типу даних у бітах; 128 (- біт) - довжина NEON64 регістра в ЦП ARM64. Тож:

$$elems(dtype) = \begin{cases} 16, & dtype \in \{u8, s8\}, \\ 8, & dtype \in \{u16, s16\}, \\ 4, & dtype \in \{u32, s32, f32\}, \\ 2, & dtype \in \{u64, s64, f64\}. \end{cases} \tag{A.14}$$

Тоді множина індексів елементів NEON64 регістрів може бути визначена як:

$$\mathcal{I}^{dtype}_{start,end} = \{i \mid i = \overline{start, end}, end < elems(dtype)\}. \tag{A.15}$$

Для повного векторного регістра це буде:

$$\mathcal{I}^{dtype}_{0,elems(dtype)-1} = \{i \mid i = \overline{0, elems(dtype) - 1}\}. \tag{A.16}$$

Також для зручності опису діапазонів елементів векторного регістра введемо функції, що буде часто використано з:

$$\begin{aligned} i_all(dtype) &= \overline{0, elems(dtype) - 1} \\ i_1hlf(dtype) &= \overline{0, \left\lfloor \frac{elems(dtype)}{2} \right\rfloor - 1} \\ i_2hlf(dtype) &= \overline{\left\lfloor \frac{elems(dtype)}{2} \right\rfloor, elems(dtype) - 1} \end{aligned} \tag{A.17}$$

де: $i_all(dtype)$ - діапазон індексів усіх елементів вектора; $i_1hlf(dtype)$ - діапазон індексів першої половини вектора; $i_2hlf(dtype)$ - діапазон індексів другої половини вектора.

Таблиця А.1 — Значення діапазонів векторів (векторних регістрів) для різних типів даних.

Назва типу даних	$i_all(dtype)$	$i_1hlf(dtype)$	$i_2hlf(dtype)$
u8/s8	$\overline{0,15}$	$\overline{0,7}$	$\overline{8,15}$
u16/s16	$\overline{0,7}$	$\overline{0,3}$	$\overline{4,7}$
u32/s32/f32	$\overline{0,3}$	$\overline{0,1}$	$\overline{2,3}$
u64/s64/f64	$\overline{0,1}$	$\overline{0,0}$	$\overline{1,1}$

Для різних типів даних функції (А.17) дають такі діапазони елементів (табл. А.1). Таким чином, ці функції формулюють компактний запис діапазонів елементів векторів [6].

Надалі для позначення діапазонів індексів векторних елементів будуть використовуватись дві еквівалентні форми запису: 1) функціональна нотація $i = i_all(dtype)$, яка відповідає всім елементам вектора відповідного типу даних згідно з (табл. А.1); 2) класична математична нотація $\forall i \in \mathcal{I}^{dtype}_{0,elems(dtype)-1}$. Обидві нотації є взаємозамінними, і їх вибір у конкретному контексті визначається доцільністю представлення математичного апарату та його узгодженістю з програмною реалізацією векторизованих алгоритмів [7].

Для конкретного позначення, використовуючи самодокументований стиль запису імен векторів, буде використано більш простий варіант нотації V_R1623^{u16} (або $v_R1623^{u8}_{i_all(u8)}$), де усі множини з (А.12) розгорнуті. Так V_R1623^{u16} означає, що відбувається обробка R-каналу ЦЗ за індексами від поточної адреси завантаження у діапазоні «$\overline{16,23}$». Важливо підкреслити, що цифрове позначення «$\overline{16,23}$» є частиною іменування NEON64 вектора. Таким чином, «1623» вказує на відступ від поточної адреси вказівника, що використовується для завантаження байтів ЦЗ, і початковий відступ дорівнює 16, а кінцевий — 23 (включно). Тож проміжні значення обрахування ШОЗ для R-каналу ЦЗ зберігаються у векторі V_R1623^{u16}, який на практиці відповідає одному з 32-х векторних регістрів (v0-v31). Діапазон значень відповідає типу даних u16 (unsigned int 16-bit) і відповідно $v_r1623_i \in [0, 0xFFFF]$ для беззнакового діапазону (для знакового діапазону: $v_r1623_i \in [-0x8000, 0x7FFF]$) [8].

Довжина NEON64-регістра суттєво не впливає на формування алгоритму, проте більший розмір регістра забезпечує краще прискорення за умови наявності

аналогічної кількості векторних регістрів за умов адаптації/використання ЯІТ для інших архітектур ЦП. У випадку з ЦП ARM64 та його NEON64 довжина векторного регістра фіксована (128-біт/16 байт), що дозволяє маніпулювати різними типами даних (А.12).

А.4 Обчислення векторізованого скалярного добутку

Даний підрозділ буде повністю сконцентровано на описі обчислення ВСД (рис. А.2), що є ядром обрахунку ШОЗ і складається з трьох основних кроків: 1) обнуління проміжних векторів; 2) завантаження даних та обрахунок ВСД на основі даних з ЦЗ та однієї строки ЯЗ; 3) завершення обрахунку ВСД, що включає повторення пункту (2) *flt_rows* разів.

Вище наведені пункти є обов'язковими до виконання для завершення обрахунку ВСД та підготовки до процедури нормалізації ВСД і фіналізації обрахунку ВОЗ.

А.4.1 Процедура «обнуління» проміжних векторів.

Для проведення обчислення ШОЗ необхідно підготувати вектори/регістри, що зберігатимуть проміжні результати обчислення. Цей крок потребує виконати їх обнуління, тобто стерти попередні дані обчислень. Далі визначають назви та кількість векторів з урахуванням колірних каналів ЦЗ, і все згадане має відповідати нотації (А.12).

Отже, для кожного колірного каналу (R, G, B) виділяється по 4 векторних регістри. Число 4 відповідає тому, що кожен канал ЦЗ займає на початку обчислень (акту завантаження) 2 векторних регістра (32 байти). А після збільшення розрядності обчислень з 8 біт до 16 біт ці 2 векторних регістра займатимуть ще 2 додаткові векторні регістра – і отже у сумі 4 векторні регістра. Наприклад, для R-каналу: V_R0007, V_R0815, V_R1623, V_R2431 (для G та B-каналів та ж кількість – по 4 проміжних регістри відповідно). Загалом для зберігання проміжних значень використовується 12 векторів.

Операція обнуління проміжних векторів, використовуючи нотацію (А.12):

$$vC_kI_e{}_i^{u16} = vC_kI_e{}_i^{u16} \otimes vC_kI_e{}_i^{u16} \tag{А.18}$$

де: операція $\otimes$ є операцією «виключного або» (XOR) [9]; $\forall C_k \in C = \{r,\ g,\ b\}$, $k = \overline{0,2}$; $\forall I_e \in I = \{0007,\ 0815,\ 1623,\ 2431\}, e = \overline{0,3}; i = i_all(u16)$ (табл. А.1), або $i = \overline{0,7}$. Операція $\otimes$ / XOR задовольняє наступним умовам для кожного біту елементів векторів:

$$\begin{cases} 1 \otimes 1 = 0, \\ 1 \otimes 0 = 1, \\ 0 \otimes 1 = 1, \\ 0 \otimes 0 = 0. \end{cases},$$

У контексті обнуління використовуються властивості $\begin{cases} 1 \otimes 1 = 0, \\ 0 \otimes 0 = 0, \end{cases}$ що призводить до встановлення нульових значень для всіх елементів векторів. Саме ці вектори в подальшому зберігають проміжні значення скалярного добутку під час обчислення ОЗ.

Операція обнуління (А.18) при повному розгортанні по множинах *C* та *I* представляється таким чином:

$$\bigcup_{k=0}^{2} \bigcup_{e=0}^{3} vC_k I_e{}_{\text{i}}^{\text{u16}},$$

де $vC_k I_e{}_{\text{i}}^{\text{u16}}$ це множини операції (А.18).

Що еквівалентно послідовній підстановці всіх можливих комбінацій C_k та I_e:

1. Розгортання по $C_k \in C = \{r,\ g,\ b\},\ k = \overline{0,2}$:

$$\begin{aligned} vrI_e{}_{\text{i}}^{\text{u16}} &= vrI_e{}_{\text{i}}^{\text{u16}} \otimes vrI_e{}_{\text{i}}^{\text{u16}}, \\ vgI_e{}_{\text{i}}^{\text{u16}} &= vgI_e{}_{\text{i}}^{\text{u16}} \otimes vgI_e{}_{\text{i}}^{\text{u16}}, \\ vbI_e{}_{\text{i}}^{\text{u16}} &= vbI_e{}_{\text{i}}^{\text{u16}} \otimes vbI_e{}_{\text{i}}^{\text{u16}} \end{aligned} \qquad \text{(А.19)}$$

2. Повне розгортання по C_k та $I_e \in I = \{0007,\ 0815,\ 1623,\ 2431\},\ e = \overline{0,3}$:

$$
\begin{aligned}
vr0007\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vr0007\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vr0007\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vr0815\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vr0815\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vr0815\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vr1623\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vr1623\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vr1623\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vr2431\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vr2431\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vr2431\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
\\
vg0007\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vg0007\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vg0007\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vg0815\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vg0815\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vg0815\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vg1623\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vg1623\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vg1623\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vg2431\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vg2431\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vg2431\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
\\
vb0007\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vb0007\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vb0007\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vb0815\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vb0815\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vb0815\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vb1623\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vb1623\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vb1623\,{}^{\mathrm{u16}}_{\mathrm{i}},\\
vb2431\,{}^{\mathrm{u16}}_{\mathrm{i}} &= vb2431\,{}^{\mathrm{u16}}_{\mathrm{i}} \otimes vb2431\,{}^{\mathrm{u16}}_{\mathrm{i}}.
\end{aligned}
\tag{A.20}
$$

Зауваження. Важливо зазначити, що інколи у даному дослідженні буде використана форма запису, що відповідає (A.19), через неможливість об'єднання/групування множин I_e та $\mathcal{I}$, але у більшості випадків застосовується саме (A.12, A.18) нотація/форма запису операцій. При цьому в обох варіантах (повному (A.19) та скороченому (A.12, A.18)) явно або неявно зберігається порядок виконання операцій, що критично для коректного виконання SIMD-алгоритмів.

A.4.2 Опис векторів, що представляють кольорові складові растру ЦЗ, та базовий алгоритм їх завантаження

Для подальшого огляду алгоритму векторизації ОЗ ЦЗ необхідно описати розміщення колірних складових $p_{i,j}/src_{src_i,src_j}$ у окремі вектори. Перед завантаженням даних у вектори, що міститимуть окремі канали кольорів компонентів ЦЗ (A.8), доцільно ввести їх узагальнені назви: V_C0015, V_C1631, де $C \in \{r, g, b\}$. Додатково, для індексації в поточному рядку ЦЗ, буде використана попередньо визначена змінна (A.6) $st32N$, а для формалізації індексації визначається:

$$
idx(step, i) = \overline{32 \cdot step + i \cdot 16,\ 32 \cdot step + i \cdot 16 + 15},
\tag{A.21}
$$

де *step* – відступ від поточної адреси вказівника $idx_{base}(st32N) + j + flt_j$

з виразу (А.8); *i* – може бути як порядковий номер з векторів множини I_e, так і окремим числом; 32 – крок завантаження ВР описаний (А.5) (кількість трійок ВР є 2).

Отже, завантаження елементів ЦЗ у вектори NEON64 описується операцією **ld3** [10]:

$$ld3(ptr, pj, k) = ptr_{pj}[k] \tag{А.22}$$

де: *ptr* - назва матриці ЦЗ; *pi* – номер рядка матриці ЦЗ; *pj* - діапазон індексів для завантаження, який обчислюється за допомогою функції *idx(step,i)*; *k* - індекс каналу кольору.

Використовуючи нотацію (А.12), операцію завантаження даних можна записати для $VC_kI_e{}^{u8} = \{vC_kI_e{}_i^{u8} | i \in \mathcal{I}_{0,15}^{u8}\}$ (або $VC_kI_e{}^{u8} = \{vC_kI_e{}_i^{u8} | i = i_all(u8)\}$) так:

$$v_C_kI_e{}_i^{u8} = \text{ld3}(src_{src_i+flt_i},\ idx(st32N,\ e),\ k) \tag{А.23}$$

де: $\forall C_k \in C = \{r,\ g,\ b\}$, $k = \overline{0,2}$ - колірні канали ЦЗ; $\forall I_e \in I = \{0015,\ 1631\}$, $e = \overline{0,1}$ - впорядкована множина іменування діапазону індексів для вказівника, від котрого виконується завантаження елементів колірних каналів ЦЗ; $\forall i \in \mathcal{I}_{0,15}^{u8}$ або $i = i_all(u8)$ див (табл. А.1) - впорядкована множина індексів елементів NEON64 регістрів/векторів.

Таким чином, операція **ld3** (А.22) ініціалізує вектори даними із зображення. Кожен елемент множини *I* (А.23) вказує на завантаження окремих колірних компонентів для блоків по 16 пікселів. Індекс k при операції ld3 вказує на відповідний колірний канал RGB ЦЗ. Цей метод дозволяє ефективно використовувати можливості NEON64, завантажуючи та обробляючи декілька даних одночасно, що забезпечує підвищення швидкодії при обробці RGB ЦЗ. Базові адреси (А.21), що використовуються у (А.23), відрізняються на 16 RGB-компонентів (по 3 байти в кожному), що у 8-бітному ЦЗ дорівнює 16·3=48 байтів. Слід зазначити, що використання операції SIMD ”ld3”NEON64 дозволяє ефективно завантажувати дані у кожну трійку векторів.

А.4.3 Додаткове завантаження векторів кольорової складової растру ЦЗ

Далі розглядається частина алгоритму ВОЗ, що виконується циклічно для кожного рядка ЯЗ (А.4) flt_{flt_i,flt_j}. Слід зауважити, що далі представлено огляд випадку, коли у ЯЗ (flt_{flt_i,flt_j}) кількість елементів у рядках може бути більше ніж 16. Цей випадок потребує окремого розгляду і для цього буде використовується вираз (А.21).

Завантаження третьої трійки 16-ти RGB елементів [10], що надалі використовуватиметься для зміщення 16 разів вліво (необхідність цього буде детально пояснено далі), здійснюється таким чином:

$$v_C_kI\,{}_i^{u8} = \text{ld3}(src_{src_i+flt_i},\ idx(st32N,\ 2),\ k) \tag{А.24}$$

де як і в (А.23): $\forall C_k \in C = \{r,\ g,\ b\},\ k = \overline{0,2}$; $\forall i \in \mathcal{I}^{u8}_{0,15}$ або $i = i_all(u8)$ див (табл. А.1); $I = \{3247\}$. Єдине, що вимагає пояснення, це те що множина *I* містить лише один елемент $I = \{3247\}$ та те, що для довантаження 3-ї групи 16 елементів, використано константу 2. Таким чином, використавши нотацію (А.12), було визначено вектор, який відповідає завантаженню третьої трійки RGB елементів - $VC_k3247\,{}^{u8} = \{vC_k3247\,{}_i^{u8} | i \in \mathcal{I}^{u8}_{0,15}\}$.

Ця операція розширює діапазон даних, доступних для обробки ВОЗ, та дозволяє виконувати зміщення для ефективного обчислення згортки у випадках, коли розмір ЯЗ перевищує базовий розмір векторного блоку в 16 елементів.

Кількість ітерацій (А.24) буде описано змінною flt_st16N, та використано у подальших блок-схемах, для факту врахування таких випадків:

$$flt_st16N = \begin{cases} 0, & \text{якщо } flt_cols < 16, \\ 0,\ \left\lfloor \frac{flt_cols}{16} \right\rfloor - 1, & \text{інакше,} \end{cases} \tag{А.25}$$

Змінна flt_st16N представляє діапазон відступів із кроком у 16 елементів у поточному рядку ЯЗ flt_{flt_i,flt_j}. Таким чином, якщо довжина рядка ЯЗ flt_{flt_i,flt_j} буде менша за 16, даний діапазон буде порожнім і подальші дії не матимуть сенсу (що відображає верхня частина умови). Отже, на даному етапі опису векторизованого алгоритму ОЗ розглядається випадок, коли *flt_cols* $>$ *16*.

А.4.4 Опис процедури завантаження векторів, що представляють ядра згортки, і формування вектору маски «позитивності»

Завантаження даних з ЯЗ у вектор $v_flt0015$ для тих самих цілей (зсув ліворуч 16 разів) виглядає простіше, оскільки значення ЯЗ flt_{flt_i,flt_j} є однаковим для всіх RGB каналів ЦЗ:

$$ld1(ptr, pj) = ptr_{pj} \tag{А.26}$$

$$v_flt0015_j = \text{ld1}(flt_{flt_i}\ ,\ idx_flt(flt_st16N)) \tag{А.27}$$

де : $j = i_all(u8)$; $idx_flt(flt16s) = \overline{16 \cdot flt16s, 16 \cdot flt16s + 15}$ - відступ від поточної адреси вказівника строки ЯЗ - flt_{flt_i}. Далі формується вектор маски "позитивності"$V_nft0015 = \{v_nft0015_j | j \in i_all(u8)\}$, який використовується для швидкого визначення знаку елементів маски фільтра з (А.27):

$$v_nft0015_j = \begin{cases} 0xFF, & \text{якщо } v_flt0015_j < 0, \\ 0, & \text{у іншому випадку.} \end{cases} \tag{А.28}$$

Вектор $V_nft0015$ далі використовується для швидкого аналізу знаку елементів вектора $V_flt0015$. Такий підхід швидший (приблизно на 10%) порівняно з постійними порівняннями на більше або менше нуля. Слід зауважити, що складніший, але швидший код часто важче модифікувати та підтримувати. Однак у даному випадку пріоритетом є саме швидкодія.

А.4.5 Процедура обчислення векторізованого скалярного добутку, що зберігається у проміжних векторах

Для формалізації процедури обрахування ВСД необхідно описати механізм використання команд NEON64 smlal/smlal2 [11] та smlsl/smlsl2 [12]. Ці команди є основою для реалізації ВСД з одночасним накопиченням результату. Якщо

виконується обов'язкова умова $elems(dst_dt) \cdot 2 = elems(src_dt)$, тоді:

$$
\begin{gathered}
smlal(V0, dst_dt, V1, V2, src_dt) = \\
V0^{dst_dt}_{i_all(dst_dt)} + (V1^{src_dt}_{i_1hlf(src_dt)} \cdot V2^{src_dt}_{i_1hlf(src_dt)})^{dst_dt}_{i_all(dst_dt)}, \\
\\
smlsl(V0, dst_dt, V1, V2, src_dt) = \\
V0^{dst_dt}_{i_all(dst_dt)} - (V1^{src_dt}_{i_1hlf(src_dt)} \cdot V2^{src_dt}_{i_1hlf(src_dt)})^{dst_dt}_{i_all(dst_dt)}, \\
\\
smlal2(V0, dst_dt, V1, V2, src_dt) = \\
V0^{dst_dt}_{i_all(dst_dt)} + (V1^{src_dt}_{i_2hlf(src_dt)} \cdot V2^{src_dt}_{i_2hlf(src_dt)})^{dst_dt}_{i_all(dst_dt)}, \\
\\
smlsl2(V0, dst_dt, V1, V2, src_dt) = \\
V0^{dst_dt}_{i_all(dst_dt)} - (V1^{src_dt}_{i_2hlf(src_dt)} \cdot V2^{src_dt}_{i_2hlf(src_dt)})^{dst_dt}_{i_all(dst_dt)}.
\end{gathered}
\tag{A.29}
$$

де: *V0, V1, V2* - векторні регістри NEON64; dst_dt, src_dt - типи даних для векторів-приймачів та векторів-джерел відповідно; $i_1hlf(src_dt), i_2hlf(src_dt)$ - діапазони індексів першої та другої половини елементів вектора типу src_dt згідно з (А.17); $(...)^{dst_dt}_{i_all(dst_dt)}$ - розширення результату множення до типу dst_dt.

Наступним кроком є введення тимчасовий вектор *V_FLT0*, що зберігає значення $v_flt0015^{u8}_0$ у кожному зі своїх елементів, що відповідає операції dup у NEON64 [13] для:

$$v_flt0^{u8}_{i_all(u8)} = v_flt^{u8}_0. \tag{A.30}$$

Використовуючи (А.29) процедуру обрахування ВСД для кожного колірного каналу та кожного елемента ЯЗ (А.30) можна записати таким чином:

$$v_C_kA_{v\,i_all(s16)}^{s16} =$$
$$\begin{cases} smlsl(v_C_kA_v, s16, v_C_kB_v, v_flt0, u8), & v_nft0015_0 = 0xFF \\ smlal(v_C_kA_v, s16, v_C_kB_v, v_flt0, u8), & \text{інакше} \end{cases} \tag{A.31}$$

$$v_C_kD_{v\,i_all(s16)}^{s16} =$$
$$\begin{cases} smlsl2(v_C_kD_v, s16, v_C_kB_v, v_flt0, u8), & v_nft0015_0 = 0xFF \\ smlal2(v_C_kD_v, s16, v_C_kB_v, v_flt0, u8), & \text{інакше} \end{cases}$$

де: $\forall C_k \in C = \{r, g, b\}, k = \overline{0, 2}$; $\forall A_v \in A = \{0007, 1623\}, v = \overline{0, 1}$; $\forall D_v \in D = \{0815, 2431\}, v = \overline{0, 1}$; $\forall B_v \in B = \{0015, 1631\}, v = \overline{0, 1}$.

Вибір між операціями додавання (smlal/smlal2) та віднімання (smlsl/smlsl2) залежить від знаку поточного елемента ЯЗ, який визначається значенням $v_nft0015_0$. Коли цей елемент дорівнює 0xFF (що відповідає від'ємному значенню), використовуються операції віднімання, інакше - операції додавання.

Для повного опису процедури ВСД, з урахуванням всіх кольорових каналів та всіх комбінацій векторів, загальна кількість операцій smlal/smlsl та smlal2/smlsl2 на один елемент ЯЗ складає:

$$3\,(\text{кількість каналів}) \times 2\,(\text{кількість елементів множини A}) \times$$
$$\times 2\,(\text{кількість елементів множини B}) = 12$$

Таким чином, для одного елемента ЯЗ виконується 12 векторних операцій множення з акумуляцією або відніманням, що забезпечує високу ефективність ВСД порівняно зі скалярною реалізацією.

Вираз (А.31) повністю відображає розрахунок ВСД ОЗ для всіх кольорових каналів оброблюваної частини ЦЗ src_{src_i,src_j}. Отже, описано першу ітерацію обчислення ВСД для векторів, що зберігають проміжні результати з (А.18) (*V_CI*, де *C*∈{*r,g,b*}, *I*∈{*0007,0815,1623,2431*}), використовуючи елементи ЯЗ (flt_{flt_i,flt_j}), що містить у рядку більше ніж 16 елементів, як описано виразом.

А.4.6 Циклічний зсув значень векторів, що уособлюють RGB ЦЗ, для забезпечення роботи з наступним елементом ЯЗ без надлишкових завантажень векторів

Далі настає етап циклічного зсуву даних (А.23) та (А.24) вліво. Для цього використовується орієнтир (0-й елемент), котрий розташовано справа у векторах V_C0015, V_C1631 та V_FLT0015. Після даної операції 0-й елемент векторів групи з (А.23) матиме індекс 1. Саме для цього було попередньо завантажено вектори V_C3247 (А.24). Ці вектори (V_C3247 та V_FLT0015) є єдиними векторами, що будуть циклічно зміщуватися "самі з собою". У наборі інструкцій NEON64 це реалізує операція **"ext v0.16b, v1.16b, v2.16b"** [14], котру можна формалізувати таким чином:

$$CShL(n,\ dtype,\ \text{vres},\ \text{vsrc0},\ \text{vsrc1}) =$$
$$\begin{cases} vres^{dtype}_{0,elems(dtype)-1-n} = vsrc0^{dtype}_{n,elems(dtype)-1}, \\ vres^{dtype}_{elems(dtype)-n,elems(dtype)-1} = vsrc1^{dtype}_{0,n-1}, \end{cases} \tag{А.32}$$

де: *n* – параметр зсуву, що залежить від розміру *dtype*; *dtype* – тип даних, що буде використано для операції *CShL*; *vres* – вектор, що отримає результат зсуву; vsrc0 та *vsrc1* – вектори, з яких будуть взяті елементи для переформатування *vres*. Таким чином, зсув 8-бітних значень можна робити у будь-якому напрямку. Але у випадку даного дослідження зсув буде постійно виконуватися на 1 байт вліво для типу u8. Отже можна переписати (А.32):

$$\begin{gathered} \forall I_e \in I = \{0015,\ 1631,\ 3247\},\ e = \overline{0,2}\ ; \\ \forall C_k \in C, \forall l \in \{0,1\} : CShL(1,\ u8,\ VC_kI_l,\ VC_kI_l,\ VC_kI_{l+1}) \\ \forall C_k \in C : CShL(1,\ \text{u8},\ VC_kI_2,\ VC_kI_2,\ VC_kI_2) \end{gathered} \tag{А.33}$$

Останнім треба описати зсув на 1 байт вліво векторів що підвантажує частини фільтру - *V_flt* з ЯЗ (А.27):

$$\forall C_k \in C : CShL(1,\ \text{u8},\ V_FLT,\ V_FLT,\ V_FLT) \tag{А.34}$$

Це фактично завершує першу базову ітерацію ОЗ. Далі всі попередні операції ВСД (А.30, А.31, А.33, А.34) необхідно повторити ще 15 разів. Таким чином,

було завершено етап обробки ВСД для рядка ЯЗ flt_{flt_i,flt_j}, що має довжину більше ніж 16 елементів, один раз. Повну кількість ітерацій для виразів (А.30, А.31, А.33, А.34) описує вираз (А.25). Важливо відзначити, що завдяки такому підходу немає необхідності постійно підвантажувати дані, використовуючи (А.23) для векторів *V_C0015, V_C1631* (де $C \in \{r, g, b\}$). Більше того, завдяки наявності векторів *V_C3247* та постійним зсувам вліво на 1 байт (А.33, А.34), дані ЯЗ та ЦЗ постійно оновлюються і **операції завантаження не використовуються**, що є **суттєвою оптимізацією як алгоритмічною, так і з точки зору реалізації ВСД для алгоритму ШОЗ в цілому**.

А.4.7 Завершення процедури обчислення значень векторізованого скалярного добутку

Далі необхідно завершити ВСД ОЗ для ЦЗ $p_{i,j}/src_{src_i,src_j}$ з елементами ЯЗ flt_{flt_i,flt_j}. Цей обрахунок ВСД спирається на остачу від ділення *flt_cols* на 16, і може бути описаним як кількість елементів, що потребують додаткової обробки - *flt_rest*:

$$flt_rest = \overline{0, mod\left(\frac{flt_cols}{16}\right) - 1}, \tag{А.35}$$

де операція *mod* - це остача від ділення.

Отже, опис алгоритму дообрахунку спирається на усі попередні вирази та може бути поданий як показано на (рис. А.4).

Щодо алгоритму дообрахунку ВСД для елементів ЯЗ, що залишилися необробленими (після обробки блоків ЯЗ по 16 елементів), то він (алгоритм) спирається на усі попередні вирази та складається з наступних кроків:

1. Додаткове завантаження даних, що описане виразом (А.24);
2. Завантаження остачі елементів рядка ЯЗ flt_{flt_i,flt_j} у $v_flt0015$, згідно з (А.27) та формування вектору *V_NFLT0015* на його основі згідно з (А.28);
3. Повторення операцій ВСД з виразів (А.30, А.31, А.33, А.34) *flt_rest* разів.

Таким чином, буде завершено розрахунок ВСД для ЯЗ, що має кількість елементів рядка більшу ніж 16. Повний алгоритм ВСД для ЯЗ з довжиною рядка більше 16 елементів може бути візуалізовано у вигляді блок-схеми (рис. А.4) для повного алгоритму ВСД, що є найважливішою частиною ШОЗ RGB ЦЗ. Як видно з блок-схеми, алгоритм складається з трьох частин: (1) - підготовчі

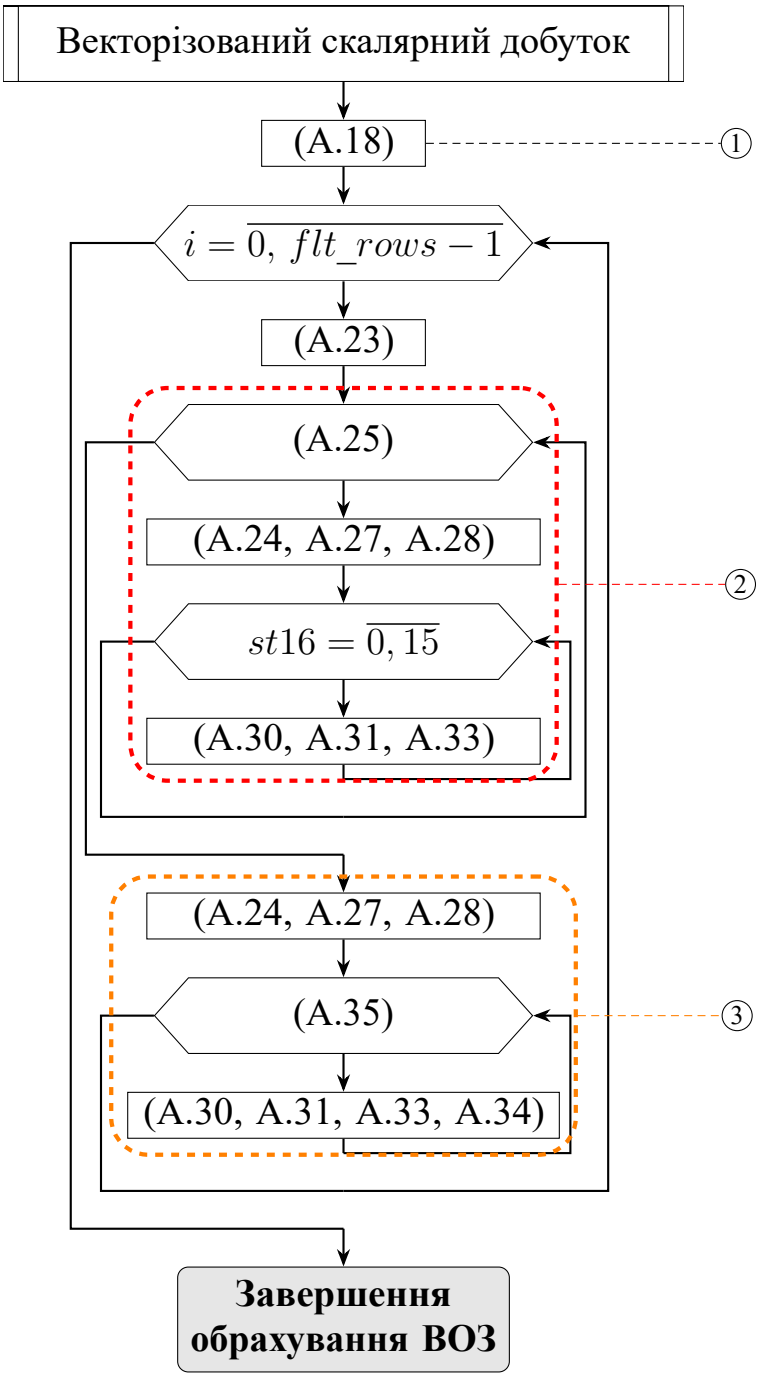


Рисунок А.4 — Алгоритм обрахування ВСД для ЯЗ, що має кількість елементів рядка більшу ніж 16, за допомогою операцій NEON64, що складається з наступних етапів: (1) – підготовчий; (2) – обчислення ВСД для ЯЗ з *flt_cols* > 16; (3) – завершення обрахунку ВСД.

дії, описані виразом (А.18) (ініціалізація/обнуління проміжних векторів); ② - обчислення ВСД, при умові, що перша частина виразу (А.25) істинна (тоді ② буде основною частиною, що буде найбільш обчислювально навантаженою); ③ - якщо перша частина виразу (А.25) не виконується, тоді ③ буде основною частиною обчислення ВСД та ВОЗ.

Таким чином, за допомогою даної блок-схеми (рис. А.4) було сформовано та описано найбільш ресурсоємну та основну частину ЯІТ - ШОЗ (рис. А.2) (без операції нормалізації).

А.5 Нормалізація векторізованого скалярного добутку

У попередньому підрозділі було надано повну інформацію щодо обчислення ВСД. Цей підрозділ описує фінальний етап обчислень ШОЗ - операцію нормалізації ВСД.

Таким чином, після обчислення ВСД (рис. А.4) (А.8) , було заповнено/сформовано $vC_kI_e\,{}_i^{s16}$, де $\forall C_k \in C = \{r,\ g,\ b\}, k = \overline{0,2}$; $\forall I_e \in I = \{0007,\ 0815,\ 1623,\ 243$ $e = \overline{0,3}$; $i = i_all(s16)$ (табл. А.1). Для завершення обчислення ШОЗ необхідно виконати нормалізацію значень $vC_kI_e\,{}_i^{s16}$ із використанням коефіцієнта *flt_div*. Математично ця операція виглядає просто, але для її виконання необхідно врахувати різницю типів *s16* та *f32*. Щоб виконати операцію нормалізації, треба або квантувати *flt_div*, або конвертувати значення у проміжних векторах до типу f32. У даному дослідженні було обрано другий варіант (конвертація значень з *s16* у тип *f32*). Ця стратегія була обрана для збереження точності обчислень.

А.5.1 Процедура конвертування значень проміжних векторів з 16-біт у 32-бітні значення з плаваючою комою (s16⇒f32).

Початковою операцією буде конвертування значень проміжних векторів з 16-біт у 32-бітні цілочисельні значення:

$$\begin{aligned} vC_kB_{e\cdot 2}\ {}_{i_all(s32)}^{s32} &= vC_kI_e\,{}_{i_1hlf(u16)}^{s16}, \\ vC_kB_{e\cdot 2+1}\,{}_{i_all(s32)}^{s32} &= vC_kI_e\,{}_{i_2hlf(u16)}^{s16}, \end{aligned} \tag{А.36}$$

де: $\forall C_k \in C = \{r,\ g,\ b\}$, $k = \overline{0,2}$; $\forall I_e \in I = \{0007,\ 0815,\ 1623,\ 2431\}$, $e = \overline{0,3}$; $B = \{0003,\ 0407,\ 0811,\ 1215,\ 1619,\ 2023,\ 2427,\ 2831\}$ - впорядкована множина іменування діапазону індексів проміжних векторів з (А.18). Таким

чином конвертування (А.36) з *s16* у *s32* буде виконано і оновлені значення множини *I* буде зберігатися у множині *B*. Слід зауважити, що вираз (А.36) виконується за допомогою команд sxtl (Signed Extend Low) та sxtl2 (Signed Extend High) [15], котрі розширюють 16-бітні значення до 32-бітних (з набору команд NEON64). Причому sxtl працює з нижньою половиною вектора (*i_1hlf*), а sxtl2 - з верхньою половиною (*i_2hlf*) векторних регістрів.

Наступний крок - це конвертування типу даних *s32* з значень векторів множини *B* у *f32*:

$$vC_kB_e\,{}^{f32}_{i_all(f32)} = scvtf\left(vC_kB_e\,{}^{s32}_{i_all(s32)}\right), \tag{А.37}$$

де *scvtf* є функцією перетворення знакового цілого 32-бітного числа у відповідне число з рухомою комою одинарної точності згідно зі стандартом IEEE 754. За це відповідає команда SCVTF (Signed Convert to Floating-Point) [16] у наборі інструкцій NEON64. Отже, операція (А.37) використовує SCVTF, що перетворює 4 знакових цілих 32-бітних числа (*s32*) у 4 32-бітні числа з плаваючою комою (*f32*).

А.5.2 Процедура нормалізації значень векторізованого скалярного добутку.

Наступним кроком буде множення елементів векторів на значення КН - flt_div. Для цього необхідно створити тимчасовий вектор *V_DIV*, усі 4 елементи якого будуть містити значення flt_div:

$$\begin{gathered} v_div\,{}^{f32}_{i_all(f32)} = flt_div, \\ vC_kB_e\,{}^{f32}_{i_all(f32)} = vC_kB_e\,{}^{f32}_{i_all(f32)} \cdot v_div\,{}^{f32}_{i_all(f32)}, \end{gathered}, \tag{А.38}$$

де: $\forall B_e \in B = \{0003,\ 0407,\ 0811,\ 1215,\ 1619,\ 2023,\ 2427,\ 2831\}, e = \overline{0,7}; \forall C_k \in C = \{r,\ g,\ b\}, k = \overline{0,2}$.

Операція (А.38) виконується за допомогою команди FMUL (Floating-point Multiply) з набору інструкцій NEON64, яка здійснює поелементне множення векторів з плаваючою комою [17]. Визначення значення результату ШОЗ (А.38), що зберігається у векторах множини *B* у форматі, що відповідає IEEE 754 (32-бітне число з рухомою комою одинарної точності, яке є базовим для математичних

обчислень при використанні ЦП ARM64), надало можливість для подальшої конвертації значень з f32 у u8. Ці конвертовані значення надалі будуть збережені у вихідному ЦЗ $dst_{src_i+\left\lfloor \frac{flt_rows}{2} \right\rfloor, idx_{base}(st32N)+j+\left\lfloor \frac{flt_cols}{2} \right\rfloor}$ (А.8).

А.5.3 Процедура конвертування значень проміжних векторів з 32-бітних значень з плаваючою комою у 32-бітні цілочисельні знакові значення (f32⇒s32).

Після обрахунку та нормалізації ВСД виникають декілька нюансів: 1) у векторах множини В можуть бути від'ємні значення; 2) у векторах також можуть бути значення, більші за максимально можливе, яке може вмістити в себе 8-бітне число (максимальне значення 0xFF).

Ці проблеми пропонується вирішувати так: елементи векторів, що мають від'ємні значення, буде зведено до 0; елементи векторів, що мають значення $>$ *0xFF*, буде приведено до значення *0xFF* (*255*).

Але почати необхідно з конвертації f32 у s32 - зворотного перетворення чисел з рухомою комою одинарної точності (згідно зі стандартом IEEE 754) у знакові цілі 32-бітні числа (є зворотною до (А.37)):

$$vC_kB_{e\ i_all(s32)}^{s32} = fcvtas\left(vC_kB_{e\ i_all(f32)}^{f32}\right) \quad (A.39)$$

де *fcvtas* є функцією, що перетворює числа з плаваючою комою (стандарт IEEE 754) у знакові цілі 32-бітні числа. У наборі інструкцій NEON64 роль цієї функції виконує команда FCVTAS (Floating-point Convert to Signed integer, with rounding to nearest, ties away from zero) [18].

Команда FCVTAS виконує перетворення з округленням до найближчого цілого, а при рівній відстані (наприклад, 1.5) - округляє у напрямку від нуля (тобто 1.5 буде округлено до 2, а -1.5 до -2). Таке округлення забезпечує більш точний результат при конвертації значень після нормалізації.

А.5.4 Процедура конвертування значень проміжних векторів з 32-біт знаковими цілочисельними значеннями у 8-бітні беззнакові числа (s32⇒u8).

Після процедури (А.39) можна перейти до другої частини підготовчого питання, а саме до конвертації значень векторів зі знакових 32-бітних цілочисельних у відповідні 16-бітні, протилежно операції, що описана (А.36). Для забезпечення коректного перетворення застосовується функція обмеження значень у діапазоні цільового типу:

$$maxmin_vdt\left(ftp,\ ttp,\ v\right) = \begin{cases} max\left(ttp\right), & \text{якщо } vec^{ftp}_{i_all(ftp)} \geq max\left(ttp\right), \\ min\left(ttp\right), & \text{якщо } vec^{ftp}_{i_all(ftp)} \leq min\left(ttp\right), \\ v^{\,ftp}_{i_all(ftp)}, & \text{інакше}, \end{cases} \tag{А.40}$$

де: ftp – тип, з якого виконується конвертування; ttp – тип, у який виконується конвертування; v – вектор, що використовується для цього. Обчислення максимального та мінімального значення було показано в (2.15). Отже, конвертація значень векторів зі знакових 32-бітних цілочисельних у відповідні 16-бітні значення виконується так:

$$\begin{aligned} vC_kI_e{}^{\,s16}_{i_1hlf(s16)} &= maxmin_vdt\left(s32,\ s16,\ vC_kB_{e\cdot 2}\right), \\ vC_kI_e{}^{\,s16}_{i_2hlf(s16)} &= maxmin_vdt\left(s32,\ s16,\ vC_kB_{e\cdot 2+1}\right) \end{aligned} \tag{А.41}$$

де множини *I* та *B* описані раніше (А.36). Дана операція (А.41) має своє відображення у наборі векторних інструкцій NEON64 - команди SQXTN (Signed saturating extract Narrow) та SQXTN2 [19], які виконують звуження знакових цілих чисел з насиченням.

Наступний крок є важливим, оскільки необхідно виконати відсів значень, що є меншими за 0:

$$vC_kI_e{}^{\,s16}_{i_all(s16)} = \begin{cases} vC_kI_e{}^{\,s16}_{i_all(s16)}, & O : I > \ vC_kI_e{}^{\,s16}_{i_all(s16)} \geq 0, \\ 0 & \text{інакше}. \end{cases} \tag{А.42}$$

Після операції (А.42) елементи векторів множини *I* далі вважаються без-

знаковими 16-бітними, тобто:

$$vC_kI_e\,{}_i^{u16} \in [0x0000,\ 0xFFFF]\,,$$

де: $\forall C_k \in C = \{r,\ g,\ b\}$, $k = \overline{0,2}$; $\forall I_e \in I = \{0007,\ 0815,\ 1623,\ 2431\}$, $e = \overline{0,3}$;$i = i_all(u16)$.

Операція (А.42) у NEON64 реалізується за допомогою команди UMAX з нульовим операндом, яка встановлює нуль для від'ємних значень.

Фінальна процедура – це конвертування значень початкових проміжних векторів $vC_kI_e\,{}_i^{u16}$ множини *I*, у беззнакові 8-бітні елементи множини L - $VC_kL_j\,{}^{u8}$, що було описано в (А.23). Тут $\forall C_k \in C = \{r,\ g,\ b\}\,,\ k = \overline{0,2}$, $\forall L_j \in L = \{0015,\ 1631\}, j = \overline{0,1}$. $VC_kL_j\,{}^{u8}$ є множиною іменування діапазону індексів для вказівника, по якому виконується збереження елементів колірних каналів вихідного ЦЗ:

$$\begin{aligned} vC_kL_j\,{}_{i_1hlf(u8)}^{u8} &= maxmin_vdt\,(u16,\ u8,\ vC_kI_{j\cdot 2})\,,\\ vC_kL_j\,{}_{i_2hlf(u8)}^{u8} &= maxmin_vdt\,(u16,\ u8,\ vC_kI_{j\cdot 2+1})\,. \end{aligned} \tag{A.43}$$

Ця операція (А.43) реалізується за допомогою команд UQXTN та UQXTN2 (Unsigned saturating extract Narrow), які виконують звуження беззнакових цілих чисел з насиченням. Крок (А.43) є фінальним у формуванні результату ВОЗ. Наступний крок - це збереження результату обчислення ШОЗ у вихідне ЦЗ, що представлене векторами $VC_kL_j\,{}^{u8}$.

А.5.5 Процедура збереження результату обчислення векторизованої операції згортки у вихідне цифрове зображення.

І на останньому кроці буде збереження результату одночасного обчислення 32 значень для $VC_kL_j\,{}^{u8}$ у ЦЗ:

$$dst_{src_i+\left\lfloor\frac{flt_rows}{2}\right\rfloor,idx(st32N,\,j)+\left\lfloor\frac{flt_cols}{2}\right\rfloor} = st3(v_C_kL_j\,{}_{i_all(u8)}^{u8},\ k) \tag{A.44}$$

де: $st3(V,k)$ - операція збереження колірних компонентів з вектора V у відповідний канал k вихідного ЦЗ; функція $idx(st32N,\ j)$ описується виразом (А.21); $\forall C_k \in C = \{r,\ g,\ b\}\,,\ k = \overline{0,2}$, $\forall L_j \in L = \{0015,\ 1631\}, j = \overline{0,1}$.

Операція **st3** є частиною набору інструкцій NEON64 і виконує збереже-

ння елементів векторів результату ШОЗ у колірні канали RGB вихідного ЦЗ $dst_{src_i+\left\lfloor \frac{flt_rows}{2} \right\rfloor, idx(st32N,\, j)+\left\lfloor \frac{flt_cols}{2} \right\rfloor}$ [20]. Зокрема, вираз (А.44) описує операцію, протилежну до операції завантаження (А.23), і виконує збереження елементів колірних каналів у вихідне ЦЗ відповідно до множини іменування діапазону індексів. Використання операції st3 дозволяє ефективно використовувати можливості NEON64, зберігаючи елементи трьох векторів (RGB-компоненти) одночасно для 16-ти пікселів, що забезпечує значне підвищення швидкодії обчислення ШОЗ порівняно з послідовним збереженням окремих пікселів.

Таким чином, завершується повний цикл обчислення ШОЗ для 32-х елементів вихідного RGB ЦЗ, котрий включає завантаження даних, обчислення ненормалізованого ВСД, нормалізацію обчислення ВСД та обчислення результату ШОЗ з наступним конвертуванням типів та збереженням 32-х оброблених елементів у вихідне RGB ЦЗ.

А.6 Деталізований опис алгоритму у вигляді блок-схеми векторизованої операції згортки.

Даний алгоритм базується на (рис. А.3) як основі обчислень ВОЗ, та на (рис. А.4) як на основі обчислення ВСД.

Для завершення повного алгоритму ШОЗ необхідно додати серію операцій:

1. конвертування значень проміжних векторів з 16-біт у 32-бітні значення з плаваючою комою $s16 \Rightarrow f32$, що було описано у (підрозд. А.5.1) та виразами (А.36, А.37);
2. процедура нормалізації значень ВСД, що було описано у (підрозд. А.5.1) та виразом (А.38);
3. зворотна процедура конвертування 32-бітних значень з плаваючою комою у 8-бітні беззнакові числа, що було описано у (підрозд. А.5.3, А.5.4) та виразами (А.39–А.43);
4. збереження результату ШОЗ у результуюче 8-бітне RGB ЦЗ, що було описано у (підрозд. А.5.5) та виразом (А.44).

Також слід зазначити, що даний алгоритм спирається на значення діапазонів st32N, що було описано виразом (А.6), та задано діапазон як завантаження, так і збереження ШОЗ у 32 елементи RGB, і що було описано $idx_{base}(st32N)$ у (А.8). Отже, сам алгоритм можна описати у вигляді такої блок-схеми:

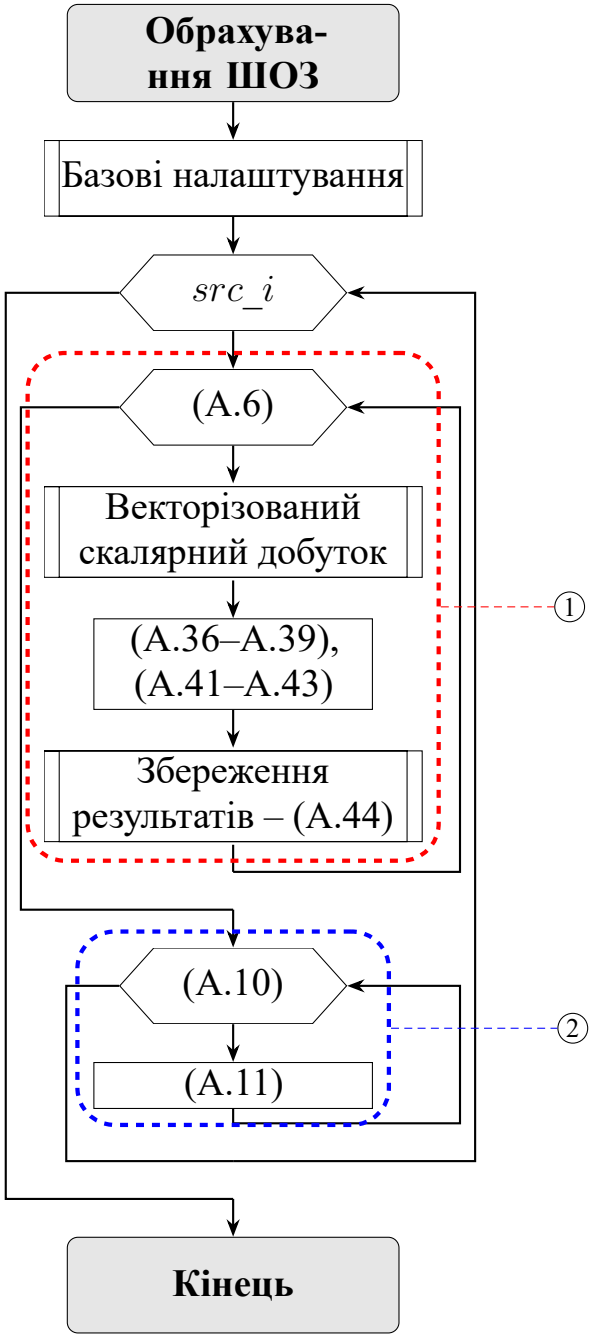


Рисунок А.5 — Блок-схема алгоритму обчислення ШОЗ для цілочисельних ЯЗ flt_{flt_i,flt_j}, де: (1) – Обрахунок ВСД, нормалізація значень ВСД і збереження результату у RGB ЦЗ; (2) – дообрахування частини ОЗ RGB ЦЗ мовою C++.

Представлений алгоритм (рис. А.5) складається з двох основних частин: перша реалізує ШОЗ та збереження результату у вихідне RGB ЦЗ (позначена ①); друга виконує дообрахування ОЗ мовою C++ і також зберігає результат у вихідне RGB ЦЗ (позначена ②). Ці дві частини були попередньо представлені на (рис. А.3) та описані виразами (А.8) та (А.11). Обидва кроки необхідно повторити src_i разів, як було описано у виразі (А.3). Блок-схема наочно демонструє структуру всього алгоритму ВОЗ, включаючи етапи обчислення ВСД, нормалізації та збереження результатів. Важливо відзначити, що в частині ① (векторизована частина) виконується одночасна обробка 32-х елементів вихідного RGB ЦЗ, а в частині ② (невекторизована частина) - обробка залишкових елементів, кількість яких залежить від розмірів вхідного зображення та ядра згортки.

А.7 Обчислення ШОЗ з ЯЗ, що має нульові елементи (у розмірі більше ніж 25% від загальної кількості)

Даний підрозділ присвячено випадкам, коли ЯЗ має у своєму складі як від'ємні значення, так і значну кількість нульових елементів. Такі випадки призводять до надлишкових обчислень, оскільки множення на 0 можна і необхідно пропускати. Таким чином, доцільно формувати кластери ненульових елементів, що є базовою ідеєю даного розділу.

А.7.1 Процедура кластеризації ЯЗ, що має деяку кількість нульових елементів

Розглянемо ЯЗ flt_{flt_i,flt_j}, що описане виразом (А.3), елементи якого приймають значення $flt_{flt_i,flt_j} \in [-0x80000000..0x7FFFFFFF]$. Введемо матрицю *index_map* відступів, що має таку ж розмірність *ЯЗ*. Елементи матриці *index_map* є парами чисел: перше число - відступ від нульового елемента рядка матриці *flt*; друге число - кількість ненульових елементів матриці flt у групі, де елементи не розташовані далі ніж на 1 позицію один від одного. Отже, матриця index_map описує кластери ненульових, однакових за знаком елементів ЯЗ [21].

Введемо змінну *k*, що буде ітератором для елементів у поточному рядку матриці *index_map* ($k \in [0; flt_cols - 1]$), яка змінюватиме свої значення за певних

умов, що будуть описані далі. Матриця *index_map* складається з пар елементів/значень $index_map_{flt_i,flt_j} = (offset_{flt_i,flt_j}, count_{flt_i,flt_j})$ і на початку обробки ініціалізована як $index_map_{flt_i,flt_j} = (-1, -1)$.

Нехай *signum(x)* – функція знаку числа, результатом роботи якої є:

$$signum(x) = \begin{cases} -1 & \text{якщо } x < 0, \\ 0 & \text{якщо } x = 0, \\ 1 & \text{якщо } x > 0. \end{cases}$$

Введемо змінну $prev_sign$, яка зберігає знак попереднього елемента в рядку *flt_i* матриці *flt*. Початкове значення $prev_sign = signum(flt_{flt_i,0})$.

Тоді умову формування елементів матриці $index_map_{flt_i,flt_j}$ можна записати таким чином:
$\forall flt_i \in \overline{0, flt_rows - 1}, \forall flt_j \in \overline{1, flt_cols - 1}$ та $k \in [0; flt_cols - 1]$

$$index_map_{flt_i,k} = \begin{cases} (offset_{flt_i,k}, count_{flt_i,k} + 1), \text{ якщо :} \\ sign(flt_{flt_i,flt_j}) = prev_sign \text{ і } sign(flt_{flt_i,flt_j}) \neq 0. \\ \\ (offset_{flt_i,k}, -1 \cdot count_{flt_i,k}), \text{ якщо :} \\ prev_sign < 0 \text{ і } sign(flt_{flt_i,flt_j}) \neq prev_sign. \\ \\ (offset_{flt_i,k}, count_{flt_i,k}), \text{ якщо :} \\ prev_sign > 0 \text{ і } sign(flt_{flt_i,flt_j}) \neq prev_sign. \end{cases}$$

$$k = \begin{cases} k + 1, & \text{якщо } sign(flt_{flt_i,flt_j}) \neq prev_sign \text{ і } sign(flt_{flt_i,flt_j}) \neq 0 \\ k, & \text{інакше} \end{cases}$$

$$index_map_{flt_i,k} = \begin{cases} (flt_j, 1), \text{ якщо } sign(flt_{flt_i,flt_j}) \neq prev_sign \\ index_map_{flt_i,k}, \text{ інакше} \end{cases}$$

$$prev_sign = signum(flt_{flt_i,flt_j})$$

(A.45)

Процес формування матриці *index_map* можна описати наступним чином:

1. Якщо знак поточного елемента flt_{flt_i,flt_j} співпадає зі знаком попереднього елемента ($sign(flt_{flt_i,flt_j}) = prev_sign$) та не дорівнює нулю ($sign(flt_{flt_i,flt_j}) \neq 0$), то збільшуємо лічильник $count_{flt_i,k}$ на 1 і зберіга-

ємо оновлений елемент $(offset_{flt_i,k}, count_{flt_i,k} + 1)$ в $index_map_{flt_i,k}$;

2. Якщо знак поточного елемента flt_{flt_i,flt_j} відрізняється від знаку попереднього елемента ($sign(flt_{flt_i,flt_j}) \neq prev_sign$) та: $prev_sign < 0$, то поточний елемент $count_{flt_i,k}$ в $index_map_{flt_i,k}$ буде помножено на -1, тобто $(offset_{flt_i,k}, -1 \cdot count_{flt_i,k})$; якщо $prev_sign > 0$ то значення $index_map_{flt_i,k}$ не станеться;
3. Якщо знак поточного елемента flt_{flt_i,flt_j} відрізняється від знаку попереднього елемента ($sign(flt_{flt_i,flt_j}) \neq prev_sign$) та не дорівнює нулю ($sign(flt_{flt_i,flt_j}) \neq 0$), то збільшуємо значення *k* на 1 ($k = k + 1$), в іншому випадку *k* не змінюється;
4. Якщо знак поточного елемента flt_{flt_i,flt_j} відрізняється від знаку попереднього елемента ($sign(flt_{flt_i,flt_j}) \neq prev_sign$) та не дорівнює нулю ($sign(flt_{flt_i,flt_j}) \neq 0$), то оновлюємо $index_map_{flt_i,k} = (flt_j, 1)$;
5. Оновлюємо значення *prev_sign* знаком поточного елемента flt_{flt_i,flt_j}: $prev_sign = signum(flt_{flt_i,flt_j})$.

Сформована матриця $index_map_{flt_i,k}$ служить орієнтиром по кластерах позитивних та негативних елементів flt_{flt_i,flt_j} в умовах, коли ЯЗ має вигляд розрідженої матриці та/або коли її елементи групуються за знаком. Таким чином, алгоритм, запропонований у (підрозд. А.5, А.6) та зображений на (рис. А.5), можна модифікувати із використанням матриці $index_map_{flt_i,k}$ для підвищення ефективності/швидкодії обчислень.

А.7.2 Процедура обчислення значень ВСД для проміжних векторів ОЗ, що використовує предобчислену предикт-матрицю

Використання матриці $index_map_{flt_i,k}$ передбачає оновлення виразів множення проміжних векторів (А.29, А.31) на елемент ЯЗ, а саме – їх необхідно розділити на дві категорії (для позитивних та негативних груп елементів $index_map_{flt_i,k}$). Таким чином, використавши групування/кластеризацію векторів ЯЗ як за знаком, так і за кількістю, для позитивних елементів ЯЗ обчислення

будуть виконуватись за умови $index_map_{flt_i,k}[1]>0$:

$$smlal(V0, dst_dt, V1, V2, src_dt) = V0^{dst_dt}_{i_all(dst_dt)} + \\ + \left(V1^{src_dt}_{i_1hlf(src_dt)} \cdot V2^{src_dt}_{i_1hlf(src_dt)}\right)^{dst_dt}_{i_all(dst_dt)},$$

$$smlal2(V0, dst_dt, V1, V2, src_dt) = V0^{dst_dt}_{i_all(dst_dt)} + \\ + \left(V1^{src_dt}_{i_2hlf(src_dt)} \cdot V2^{src_dt}_{i_2hlf(src_dt)}\right)^{dst_dt}_{i_all(dst_dt)}. \quad (A.46)$$

А для негативних елементів ЯЗ ($index_map_{flt_i,k}[1]<0$) будуть виконуватись:

$$smlsl(V0, dst_dt, V1, V2, src_dt) = V0^{dst_dt}_{i_all(dst_dt)} - \\ - \left(V1^{src_dt}_{i_1hlf(src_dt)} \cdot V2^{src_dt}_{i_1hlf(src_dt)}\right)^{dst_dt}_{i_all(dst_dt)},$$

$$smlsl2(V0, dst_dt, V1, V2, src_dt) = V0^{dst_dt}_{i_all(dst_dt)} - \\ - \left(V1^{src_dt}_{i_2hlf(src_dt)} \cdot V2^{src_dt}_{i_2hlf(src_dt)}\right)^{dst_dt}_{i_all(dst_dt)}, \quad (A.47)$$

де всі позначення та операції описані у виразі (А.29).

Після виконання підготовчої операції, описаної виразом (А.30), та спираючись на значення $index_map_{flt_i,k}$, формуються дві окремі операції обчислення ВСД для елементів ЯЗ з позитивними та негативними значеннями, що також залежать від довжини та неперервності зміни знаку в ЯЗ. Вираз (А.31) можна переписати так при $index_map_{flt_i,k}[1] > 0$:

$$\begin{aligned} v_C_kA_{v\cdot 2}\ {}^{s16}_{i_all(s16)} &= smlal(v_C_kA_{v\cdot 2}, \quad s16, v_C_kB_v, v_flt0, u8), \\ v_C_kA_{v\cdot 2+1}\ {}^{s16}_{i_all(s16)} &= smlal2(v_C_kA_{v\cdot 2+1}, s16, v_C_kB_v, v_flt0, u8), \end{aligned} \quad (A.48)$$

І навпаки, вираз (А.31) можна переписати за умови $index_map_{flt_i,k}[1] < 0$ так:

$$\begin{aligned} v_C_kA_{v\cdot 2}\ {}^{s16}_{i_all(s16)} &= smlsl(v_C_kA_{v\cdot 2}, \quad s16, v_C_kB_v, v_flt0, u8), \\ v_C_kA_{v\cdot 2+1}\ {}^{s16}_{i_all(s16)} &= smlsl2(v_C_kA_{v\cdot 2+1}, s16, v_C_kB_v, v_flt0, u8), \end{aligned} \quad (A.49)$$

де в обох виразах: $\forall C_k \in C = \{r, g, b\}, k = \overline{0, 2}; \forall A_j \in A = \{0007, 0815, 1623, 2431\}, j$

$\overline{0,3}; \forall B_v \in B = \{0015, 1631\}, v = \overline{0,1}$.

Слід зазначити, що кількість однакових за знаком елементів ЯЗ суттєво впливає на швидкодію ОЗ в реальних ситуаціях, оскільки групування обчислень по 1, 2 та 3 елементам за одну ітерацію циклу операцій (тут йдеться про вирази (А.48) або (А.49)) призводить до природного розгортання циклу програми [22–24]. Таким чином, якщо не використовувати аналіз знаку множення за допомогою вектора $V_NFT0015$ (А.28), ШОЗ може бути значно прискорена [6, 25, 26]. Це призводить до того, що кожна операція з векторами $v_C_kA_v$ ділиться на дві категорії при обчисленні скалярного добутку: на множення з додаванням (А.48) та множення з відніманням (А.49). Тепер все залежатиме від кількості таких дій у поточному елементі $index_map_{flt_i,k}[1]$.

А.7.3 Векторизована процедура обчислення значень скалярного добутку для проміжних векторів ОЗ у випадку, коли $index_map_{flt_i,k}[1] > 16$

Для подальшого опису алгоритму також необхідно розглянути випадок, коли значення $index_map_{flt_i,k}[1] > 16$. Для цього вводимо змінні, які будуть це враховувати - flt_pr16N та flt_pr16N_rest:

$$flt_pr16N = \overline{0, \left\lfloor \frac{|index_map_{flt_i,k}[1]|}{16} \right\rfloor - 1} \tag{А.50}$$

$$flt_pr16N_rest = \overline{0, |index_map_{flt_i,k}[1]| \ mod\, 16 - 1} \tag{А.51}$$

де flt_pr16N визначає кількість повних блоків по 16 елементів, а flt_pr16N_rest - кількість елементів, що залишились у неповному блоці.

Отже, оновлений алгоритм обчислення ВСД, що зображено на (рис. А.4), можна модифікувати, врахувавши $index_map_{flt_i,k}$:

Даний алгоритм є комбінацією двох підваріантів алгоритму (рис. А.6):

1. Коли ЯЗ та предикт-матриця мають усі елементи позитивні (і як варіант, вони добре згруповані), тобто виконується ліва частина (рис. А.6).
2. Коли ЯЗ та предикт-матриця мають усі елементи негативні і вони добре згруповані (права частина рис. А.6).

Найгіршим варіантом, який потребує окремого розгляду, є випадок, коли елементи ЯЗ згруповані по 1 на кластер, і виконання алгоритму постійно перемикається між лівою та правою гілками.

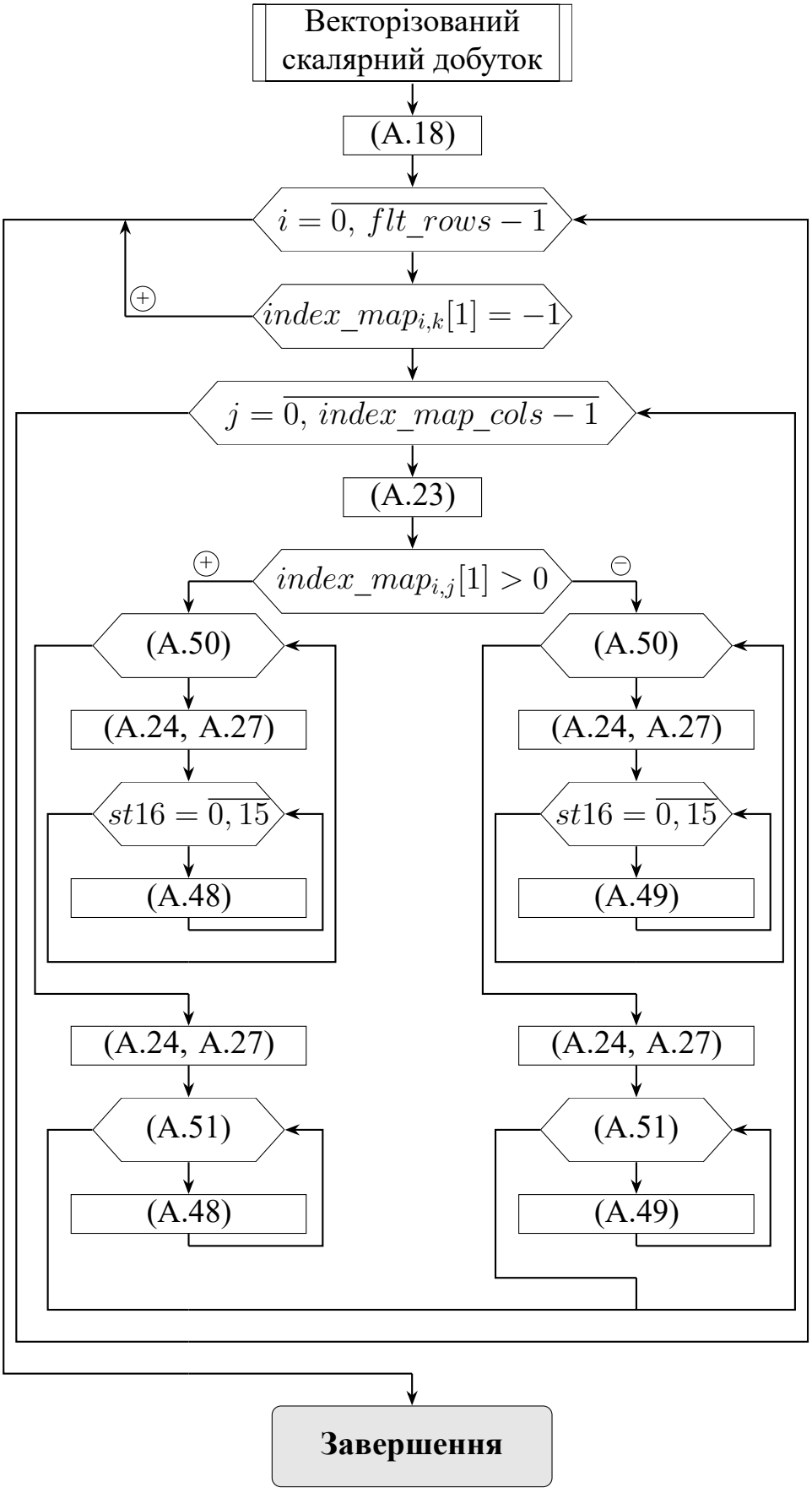


Рисунок А.6 — Алгоритм обчислення ВСД для ЯЗ, що має кількість елементів рядка більшу ніж 16, за допомогою операцій NEON64 та із використанням матриці *index_map*.

Результуючий алгоритм ВОЗ, представлений на (рис. А.5), що використовує оптимізований ВСД з матрицею index_map, формується методом заміни процедури з (рис. А.4) на (рис. А.6).

Таким чином, було описано оптимізовану ВОЗ, що є комбінацією алгоритму (рис. А.5) та оптимізованого ВСД (рис. А.6). Запропонована оптимізація дозволяє значно підвищити швидкодію обчислень для розріджених ядер згортки з великою кількістю нульових елементів, уникаючи надлишкових операцій множення.

А.8 Оптимізація обчислення векторизованого скалярного добутку за рахунок використання пари switch–continue мовою C/C++.

Описані математичні основи та підхід (підрозд. 2.1, 2.2) до реалізації ОЗ передбачають, що ЯЗ може містити як додатні, так і від'ємні елементи. Однак, якби всі елементи ЯЗ, описаного виразом (2.1), були додатними, швидкодія обчислень могла б бути вищою. Це пояснюється невеликим дослідженням у вигляді проведеного експерименту, в рамках якого отримано показники, що свідчать про те, що ОЗ над додатними величинами ЯЗ виконується незначно, але швидше (приблизно на 10–15%). Таке прискорення не є суттєвим, однак воно спостерігається за умови, що всі елементи ЯЗ є додатними і відсутні додаткові попередні обчислення. Натомість, якщо елементи ЯЗ мають від'ємні значення, у реалізації ШОЗ використовується оператор switch мови C/C++:

```
switch (…) {
case 1: {…} continue;
case 2: {…} continue;
case 3: {…} continue;
case -1: {…} continue;
case -2: {…} continue;
case -3: {…} continue;
default: {…} continue;
}
```

та

```
if (…) {… continue;} else {… continue;}
```

Таке використання оператора continue у конструкціях switch та if не є стандартним і обумовлене необхідністю якнайшвидшого переходу до наступної ітерації циклу. Як наслідок, оператор switch спричиняє зниження швидкодії ШОЗ для обробки ЦЗ на 10–15%. Основною причиною цього є те, що умовні оператори та цикли в мовах C/C++ на практиці транслюються в асемблерні команди умовного переходу. У кращому випадку використовується порівняння значення Z-біту/прапорця (прапорця нульового значення попередньої операції) стану регістра з нулем, що хоч і є оптимальнішим, але все одно призводить до пригнічення швидкодії. Це дослідження базується на припущенні, що ЯЗ містить від'ємні значення, через що в основі ШОЗ застосовуються оператор switch та цикли над ним. Для підвищення швидкодії було використано стратегію, описану в (підрозд. 1.3.6), яка суттєво покращила швидкодію ШОЗ ЦЗ (навіть за наявності від'ємних значень у ЯЗ), порівняно зі стандартним ітеративним підходом. Останній передбачає виконання лише однієї операції за ітерацію, зокрема обчислення скалярного добутку для одного значення ЯЗ [27, 28].

У (підрозд. 2.4) було описано алгоритм конвертування ЯЗ з від'ємними значеннями у ЯЗ з додатними значеннями. Це було зроблено для загострення уваги до того факту, що якщо усі елементи звести до діапазону значень $\gamma_{ii-i,jj-j} \in [0; 255]$, то необхідності у використанні конструкції switch–continue не буде. Більше того, векторизацію для ОЗ можна звести до стану, коли виконується перевірка лише на ненульові значення ЯЗ, або у більш просунутому варіанті, використовуючи $index_map_{flt_i,k}$ (А.45), що показано на (рис. А.6).

Важливим також є те, що інваріантність/вірність модифікованої ОЗ для 8-бітного RGB ЦЗ була підтверджена для NEON64 розробленою ІТ та розрахунками до цього (усі попередні пункти та опис основного алгоритму дослідження) [3, 4].

Таким чином, ЯЗ можна конвертувати у новий вигляд (2.18), що матиме діапазон значень $\gamma_{ii-i,jj-j} \in [0; 255]$, але після операції пошуку скалярного добутку треба зробити додаткові обчислення, що спираються на (2.19). Ці додаткові обчислення також включають обчислення значення Бокс-фільтра. Таким чином дана ОЗ (2.19) включатиме як пошук скалярного добутку, описаного на (рис. А.6), так і додатково частину обчислення Бокс-фільтра (чого немає на (рис. А.6)). З цього виходить, що навіть за умови повторного використання векторних регістрів (мається на увазі використання тих самих векторних регістрів, що використовуються

в операціях (А.18–А.34)), очікуване зменшення швидкодії порівняно з поточним наявним алгоритмом імовірно складатиме ті самі ($\sim$ 10–15%). Таким чином, це призведе або до зниження швидкодії ШОЗ ЦЗ (у гіршому випадку, що є більш імовірно), або не змінить істотно ці показники (за умови використання підходу розгортання циклу та техніки плитковості, згаданих у підрозд. 1.3.6). Саме тому даний опис алгоритму буде завершено і залишено для подальших досліджень [29].

А.9 Алгоритм конвертування від'ємних значень ЯЗ у беззнакові числа

У (підрозд. 2.5) було описано алгоритм конвертування ЯЗ із значеннями, що не відповідають умові виразу (2.2), і елементи ЯЗ є додатними. Хоча слід зазначити, що підхід у (підрозд. 2.5) — це лише приклад, і основним критерієм алгоритму є те, що результат обчислення скалярного добутку (що є складовою частиною ОЗ) на будь-якому етапі не має перевищувати значення 0xFFFFFFFF або діапазон значень [−0x80000000; 0x7FFFFFFF]. У цьому підрозділі буде проведено огляд робастності даного підходу та перевірено діапазони значень, яких може набувати ЯЗ, що містить значення, більші за 8-бітний діапазон [2, 3, 30].

Для обчислення ОЗ, що має справу з 32-бітним беззнаковим ЯЗ, логічною є така умова (для ЯЗ):

$$\sum_{flt_i=0}^{flt_rows-1}\sum_{flt_j=0}^{flt_cols-1} flt_{flt_i,flt_j} \leq 0x1010101, \tag{А.52}$$

де $0x1010101$ значення враховано так $\frac{0xFFFFFFFF}{0xFF} = 0x1010101$.

Тож на практиці сума елементів беззнакового ЯЗ не має перевищувати значення $0x1010101$. Якщо умова (А.52) не виконується, то потрібно залучати 64-бітні цілочисельні обчислення ОЗ.

Для ОЗ, в основі котрого є 32-бітне ЯЗ з від'ємними елементами, умова обчислення має такий вигляд (для ЯЗ):

$$-0x808080 \leq \sum_{flt_i=0}^{flt_rows-1}\sum_{flt_j=0}^{flt_cols-1} flt_{flt_i,flt_j} \leq 0x808080 \tag{А.53}$$

де $0x808080$ значення визначено так: $\frac{0x7FFFFFFF}{0xFF} = \frac{|-0x80000000|}{0xFF} = 0x808080$.

На практиці сума елементів беззнакового ЯЗ не має перевищувати значення $0x1010101$. Якщо умова (А.53) не виконується, то потрібно залучати 64-бітні цілочисельні беззнакові обчислення ОЗ [31].

З цього випливає таке логічне твердження — середнє значення елемента ЯЗ flt_{flt_i,flt_j} для ОЗ, що використовує 32-бітні беззнакові обчислення:

$$0 \leq flt_{flt_i,flt_j} \approx \frac{0x1010101}{flt_cols \cdot flt_rows} \tag{А.54}$$

А для знакового обчислення ОЗ, що використовує 32-бітні обчислення, середнє значення елемента ЯЗ flt_{flt_i,flt_j} можна описати такою умовою:

$$\frac{-0x808080}{flt_cols \cdot flt_rows} \leq flt_{flt_i,flt_j} \leq \frac{0x808080}{flt_cols \cdot flt_rows} \tag{А.55}$$

Тож (А.54) та (А.55) напряму залежать від розміру ЯЗ (та, звісно, наявності в ньому нульових елементів). Якщо всі елементи ЯЗ не нульові, то пару «розмір ЯЗ (від 2 до 15) — середнє значення для елемента ЯЗ» можна записати так: 2×2: ±0x202020, 3×3: ±0xE472A, 4×4: ±0x80808, 5×5: ±0x523DC, 6×6: ±0x391CA, 7×7: ±0x29F5B, 8×8: ±0x20202, 9×9: ±0x19621, 10×10: ±0x148F7, 11×11: ±0x10FDF, 12×12: ±0xE472, 13×13: ±0xC2A7, 14×14: ±0xA7D6, 15×15: ±0x9234. Ці пари (розмір ЯЗ та середнє значення) вірні для знакових елементів ЯЗ (А.55). Щодо виразу (А.54), відповідні пари матимуть такий вигляд: 2×2: 0x404040, 3×3: 0x1C8E55, 4×4: 0x101010, 5×5: 0xA47B8, 6×6: 0x72395, 7×7: 0x53EB6, 8×8: 0x40404, 9×9: 0x32C42, 10×10: 0x291EE, 11×11: 0x21FBE, 12×12: 0x1C8E5, 13×13: 0x1854E, 14×14: 0x14FAD, 15×15: 0x12469.

Нескладно побачити, що розмірність середнього значення (А.54) для великого ЯЗ становить 0x12469 — тобто 3 байти. Отже, складання всіх додаткових ЯЗ на основі (2.26) (на кшталт (2.25)) не має жодного сенсу. І це у випадку, якщо елементи ЯЗ взагалі міститимуть такі великі значення, як 0x12469. Зазвичай, добре квантизоване ЯЗ матиме 16-бітні елементи, і тоді вираз (2.28) матиме такий вигляд:

$$F\left(p^{i,j}\right) = \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma^{(0)}_{ii-i,jj-j} p_{ii,jj} + \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma^{(1)}_{ii-i,jj-j} p_{ii,jj} \ll 8. \tag{А.56}$$

Більше того, використовуючи матрицю $index_map_{flt_i,k}$ (А.45), можна прибрати нульові операції множення в обох матрицях ($\gamma^{(0)}_{ii-i,jj-j}$ та $\gamma^{(1)}_{ii-i,jj-j}$), що ще більше пришвидшить обчислення ОЗ. Тож, маючи 16-бітний алгоритм обчислення як основу, можна розширити можливості алгоритму на (рис. А.6) до 16-бітних обчислень, маючи 2 ЯЗ (або більше у рідкісних випадках).

Для формального обґрунтування ефективності запропонованого методу, можна додати кількісну оцінку швидкодії, що враховує особливості розбиття ядра на байтові компоненти та використання індексної матриці для пропуску нульових множень.

Для ядра згортки розміром N×N з елементами, що мають типові значення в межах 16 біт, очікуване прискорення використання запропонованого підходу (А.56) з оптимізацією пропуску нульових множень через матрицю $index_map_{flt_i,k}$ порівняно зі стандартним 32-бітним обчисленням можна оцінити як:

$$S = \frac{T_{\text{standard}}}{T_{\text{optimized}}} \approx \frac{N^2 \cdot C_{\text{32-bit}}}{N^{(0)}_{non\text{-}zero} \cdot C_{\text{8-bit}} + N^{(1)}_{non\text{-}zero} \cdot C_{\text{8-bit}} + C_{overhead} + C_{combine}} \tag{А.57}$$

де:

1. $N^{(i)}_{non\text{-}zero}$ – кількість ненульових елементів у відповідному байтовому ядрі $\gamma_{i,j}$
2. $C_{\text{32-bit}}$ – обчислювальна вартість 32-бітного множення та додавання
3. $C_{\text{8-bit}}$ – обчислювальна вартість 8-бітного множення та додавання
4. $C_{overhead}$ – накладні витрати на обробку індексної матриці $index_map_{flt_i,k}$
5. $C_{combine}$ – вартість комбінування результатів з різних байтових компонент

Враховуючи, що на більшості сучасних архітектур $C_{\text{8-bit}} \approx (0{,}25\text{–}0{,}5) \cdot C_{\text{32-bit}}$, а для типових розріджених ядер $N^{(1)}_{non\text{-}zero} < N^{(0)}_{non\text{-}zero} \ll N^2$, особливо для старших байтів, можна очікувати значне прискорення.

Для конкретного випадку згорткового ядра з розмірністю 5×5, де типові значення елементів не перевищують 0xA47B8 (як показано в дод. А.3), можна очікувати такий розподіл ненульових елементів:

1. У компоненті $\gamma^{(0)}$: від 60% до 100% ненульових елементів;
2. У компоненті $\gamma^{(1)}$: від 30% до 70% ненульових елементів;
3. У компоненті $\gamma^{(2)}$: від 0% до 30% ненульових елементів.

Це дає теоретичне прискорення:

$$S \approx \frac{25 \cdot C_{32\text{-}bit}}{(0{,}8 \cdot 25) \cdot C_{8\text{-}bit} + (0{,}5 \cdot 25) \cdot C_{8\text{-}bit} + C_{overhead} + C_{combine}}$$

Припускаючи, що $C_{8\text{-}bit} \approx 0{,}3 \cdot C_{32\text{-}bit}$, $C_{overhead} \approx 0{,}1 \cdot N^2 \cdot C_{32\text{-}bit}$ та $C_{combine} \approx 0{,}05 \cdot N^2 \cdot C_{32\text{-}bit}$, отримуємо:

$$S \approx \frac{25 \cdot C_{32\text{-}bit}}{(0{,}8 \cdot 25 + 0{,}5 \cdot 25) \cdot 0{,}3 \cdot C_{32\text{-}bit} + 0{,}15 \cdot 25 \cdot C_{32\text{-}bit}} \approx \frac{25}{9{,}75} \approx 2{,}56$$

Тобто, теоретично можна очікувати прискорення у 2,5–3 рази порівняно зі стандартним 32-бітним обчисленням для типових ядер розміром 5x5. Для більших ядер і більш розріджених матриць старших байтів прискорення може бути ще більшим.

Важливо зазначити, що реальне прискорення залежатиме від:

1. Конкретної апаратної платформи та її оптимізацій для 8-бітних операцій
2. Ступеня розрідженості компонентних матриць
3. Ефективності реалізації індексної матриці та механізму пропуску нульових множень

Таким чином, дана ОЗ (А.56) включатиме пошук скалярного добутку, описаного на (рис. А.6), для 2-х ЯЗ, що виглядатиме більш громіздким порівняно з обчисленням ОЗ, де ЯЗ має елементи з плаваючою комою. Але, як показано на прикладі значень (2.25), чим «ближче» ми до найбільш значущого байта оригінального ЯЗ, формуючи ЯЗ $\gamma_{i,j}^{(3)}$, тим більша вірогідність формування спарс-ЯЗ. З цього випливає логічний висновок — використання матриці $index_map_{flt_i,k}$ (А.45) значно пришвидшить обчислення ЯЗ з елементами значень понад 8-бітний діапазон. Тож і цей опис оптимізації обчислення ЯЗ з елементами значень понад 8-бітний діапазон буде завершено і залишено для подальших досліджень, хоча потенціал даного підходу дуже великий (у гіршому випадку прискорення становитиме 20–60%).

А.10 Загальний опис кеш-френдлі підходу та як він може прискорити ОЗ ЦЗ

Наївний підхід до реалізації ОЗ, використовуючи код на C/C++, базується на класичному обчисленні скалярного добутку між елементами рядка ЦЗ та відповідними елементами ЯЗ. У цьому випадку для кожного пікселя вихідного зображення виконується послідовний перебір елементів ядра та сусідніх пікселів у вхідному зображенні, після чого обчислюється їхня зважена сума. Такий метод є основою, добре відомою в теорії обробки цифрових сигналів та ЦЗ, і зазвичай реалізується через вкладені цикли. Хоча цей підхід простий і зрозумілий, він не враховує особливості апаратного забезпечення, зокрема організацію кеш-пам'яті центрального процесора (ЦП) [32].

При реалізації наївного підходу виникає проблема, пов'язана з неефективним використанням кеш-пам'яті ЦП. Якщо розмір хоча б двох рядків ЦЗ перевищує обсяг кешу (наприклад, L1 або L2 залежно від архітектури), то під час обробки виникає так званий "промах кешу"(cache miss). Це означає, що дані, необхідні для обчислень, не можуть одночасно перебувати в кеші, і ЦП змушений звертатися до оперативної пам'яті (RAM), що значно повільніше [33]. У реальних умовах, коли розмір зображення великий, ОС або апаратний контролер пам'яті намагається завантажити додаткові рядки зображення в кеш. Проте часте витіснення даних із кешу призводить до зниження швидкодії обчислень скалярного добутку, а отже, й усієї ОЗ в цілому.

Для підвищення швидкодії ОЗ необхідно врахувати кешованість даних, тобто забезпечити, щоб усі потрібні для обчислень дані тимчасово розміщувалися в неперервній пам'яті, яку можна умовно назвати "кешем". Такий підхід дозволяє зменшити ймовірність промахів кешу, оскільки ЦП може швидко отримувати доступ до компактно розташованих даних без звернень до RAM [34].

Основна ідея полягає в організації обробки ЦЗ блоками (або "кубами"), які вміщаються в кеш-пам'ять, і виконувати ОЗ у їхніх межах. Як показують експерименти з множенням матриць [35], подібна оптимізація може суттєво прискорити обчислення за рахунок покращення локальності даних.

Для реалізації кеш-френдлі підходу пропонується виділити буфер неперервної пам'яті, який враховує базовий крок векторизації (32 пари RGB) та

розміри ядра згортки. Розмір цього буфера визначається як (у байтах):

$$\left\lfloor \frac{32 - flt_cols + 1}{32} \right\rfloor \times (32 - flt_rows + 1) \tag{A.58}$$

де: 32 — базовий крок для векторизації (кількість пікселів, що обробляються за один цикл); flt_cols і flt_rows — ширина та висота ядра згортки відповідно; 3 — кількість байтів на піксель для RGB-формату (8 біт на канал); flt_cols - 1 і flt_rows - 1 — додаткові пікселі/рядки, що враховують ”обрамлення”для коректної роботи ядра.

Наприклад, для ядра розміром 3×3 (flt_cols = 3, flt_rows = 3) розмір буфера становить:

1. Ширина: (32 + 3 - 1) × 3 = 34 × 3 = 102 байти,
2. Висота: 32 + 3 - 1 = 34 рядки,
3. Загальний розмір: 102 × 34 = 3468 байтів.

Кількість таких блоків у зображенні залежить від розміру ЦЗ (src_rows — висота, src_cols — ширина в пікселях) і визначає, скільки разів потрібно копіювати шматки зображення в цей буфер. Точна кількість копіювань розраховується з урахуванням перекриття блоків і виражається як:

$$\left\lceil \frac{src_cols - flt_cols + 1}{32} \right\rceil \times \left\lceil \frac{src_rows - flt_rows + 1}{32} \right\rceil, \tag{A.59}$$

де: $\lceil \cdot \rceil$ — функція округлення вгору до найближчого цілого числа. Ця формула враховує, що кожен блок обробляє 32×32 пікселі результату, але через розмір ядра включає додаткові пікселі, а також забезпечує покриття всього зображення, навіть якщо його розміри не є кратними 32.

Модифікація коду полягає в переході від послідовної обробки рядків зображення до блочного підходу, де дані копіюються в буфер неперервної пам’яті перед обчисленням згортки [36]. Це необхідно з кількох причин:

1. Покращення локальності даних: Усі пікселі, потрібні для обчислень у межах блоку, знаходяться в компактному буфері, що зменшує промахи кешу.
2. Сумісність із векторизацією: Базовий крок 32 пікселі дозволяє ефективно використовувати інструкції ARM64 NEON (наприклад, ld3 для завантаження трійок RGB), що вже реалізовано в поточній векторизованій версії

коду.

3. Зменшення звернень до пам’яті: Одноразове копіювання блоку усуває повторні звернення до розкиданих даних у великому масиві ЦЗ.

Очікується, що такий підхід призведе до прискорення ОЗ в цілому за рахунок зменшення накладних витрат на доступ до пам’яті. Хоча прискорення навряд чи досягне 32 разів, як у випадку з множенням матриць (оскільки згортка має меншу обчислювальну щільність і більше залежить від розміру ядра), прогнозується підвищення швидкодії до 50% порівняно з поточною векторизованою реалізацією. Цей приріст базується на аналогії з експериментами множення матриць, де кеш-френдлі оптимізація суттєво скоротила час виконання, а також на врахуванні специфіки ARM64-архітектури, де локальність даних відіграє ключову роль.

А.11 Висновки

У цьому додатку представлено теоретичну базу нової ІТ для підвищення швидкодії лінійної ОЗ для обробки ЦЗ за умови 16-бітних цілочисельних обчислень. За результатами проведених досліджень можна зробити такі висновки:

1. Розроблено формалізовану нотацію для опису векторизованих обчислень з використанням SIMD NEON64. Запропонована математична нотація (А.12–А.17) дозволяє абстрагуватися від конкретних асемблерних інструкцій та представляти векторні операції у вигляді математичних виразів. Ця нотація є науковою новизною, оскільки аналогічні стандартизовані підходи у літературі практично відсутні.
2. Математично формалізовано ШОЗ для RGB-зображень. Модифіковано базове рівняння ОЗ (2.1) до вигляду (А.3, А.4), що враховує особливості векторизованих обчислень та 16-бітну арифметику. Показано, що використання 16-бітних цілочисельних обчислень забезпечує оптимальний баланс між точністю та швидкодією.
3. Розроблено комплекс алгоритмів ВСД для різних типів ЯЗ. Створено спеціалізовані реалізації для беззнакових (А.29) та знакових ЯЗ, що дозволяє оптимізувати обчислення залежно від характеристик конкретного ЯЗ. Розроблено механізм автоматичного вибору найефективнішого варіанту алгоритму.

4. Запропоновано метод кластеризації ЯЗ для оптимізації обробки розріджених ЯЗ. Розроблена процедура кластеризації (А.45) дозволяє групувати елементи ЯЗ за знаком та ненульовим значенням, що забезпечує значне прискорення для ЯЗ з великою кількістю нульових елементів (понад 25%). Це особливо актуально для практичних застосувань, де часто використовуються розріджені ЯЗ.
5. Детально описано процес нормалізації ВСД з урахуванням конвертації між різними типами даних (s16$\longleftrightarrow$f32$\longleftrightarrow$u8). Розроблено ефективні алгоритми перетворення типів (А.36–А.43), що мінімізують втрати точності та забезпечують коректне оброблення граничних значень.
6. Створено повні блок-схеми алгоритмів ШОЗ (рис. А.4-рис. А.6), що демонструють практичну реалізацію запропонованих методів. Схеми враховують різні сценарії використання: від базової ШОЗ до оптимізованої обробки розріджених ЯЗ з кластеризацією.
7. Розглянуто додаткові оптимізаційні стратегії, включаючи використання ”switch-continue”конструкцій (підрозд. А.8), конвертування від’ємних значень ядер у беззнакові числа (підрозд. А.9) та кеш-френдлі підходи (підрозд. А.10). Показано потенціал цих методів для подальшого підвищення швидкодії ВОЗ.

Розроблені у цьому додатку алгоритми та математичний апарат становлять теоретичну основу для створення високошвидкодійної ІТ обробки ЦЗ для процесорів сімейства ARM64. Запропонована формалізована нотація та алгоритми ШОЗ можуть бути використані для розробки широкого спектру додатків обробки ЦЗ, що потребують високої швидкодії, і використовуються на вбудованих системах та одноплатних комп’ютерах. Результати цього додатка створюють фундамент для практичної реалізації ІТ, яка розглядається у (підрозд. 4.1–4.6), та експериментального дослідження ефективності ВОЗ, представленого у (підрозд. 4.6).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Prystavka P.*, *Shevchenko A.*, *Rokitianska I.* Comparative analysis of detector-tracker architecture for object tracking based on SBC for UAV. 2024 IEEE 7th international conference on actual problems of unmanned aerial vehicles development (APUAVD) (22–24 October 2024). 2024. P. 175–178. URL: ht

tps://doi.org/10.1109/APUAVD64488.2024.10765890 (date of access: 29.07.2025).

2. *Shevchenko A.*, *Tymchyshyn V.* A SIMD-based approach to the enhancement of convolution operation performance. Proceedings of the international workshop on conflict management in global information networks (CMiGIN). 2019. Vol. 2588. P. 447–458. URL: https://ceur-ws.org/Vol-2588/paper37.pdf (date of access: 29.07.2025).
3. *Shevchenko A.*, *Prystavka P.*, *Tymchyshyn V.* Research on possible convolution operation speed enhancement via AArch64 SIMD. Advances in computer science for engineering and education. Lecture Notes on Data Engineering and Communications Technologies. Cham : Springer International Publishing, 2022. Vol. 134. P. 61–75. URL: https://doi.org/10.1007/978-3-031-04812-8_6.
4. *Shevchenko A.*, *Prystavka P.* Enhancement of convolution operation performance using SIMD of aarch64. Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II. CEUR Workshop Proceedings. 2024. Vol. 3826. P. 385–391. URL: https://ceur-ws.org/Vol-3826/#short32 (date of access: 29.07.2025).
5. *Martin R. C.* Clean code: a handbook of agile software craftsmanship. Upper Saddle River : Pearson Education, 2009.
6. *Pyeatt L. D.*, *Ughetta W.* ARM 64-bit assembly language. Cambridge : Newnes, 2019. 508 p.
7. *Muresano R.*, *Rexachs D.*, *Luque E.* Learning parallel programming: a challenge for university students. Procedia computer science. 2010. Vol. 1: 1. P. 875–883. (ICCS 2010). URL: https://doi.org/10.1016/j.procs.2010.04.096 (date of access: 13.09.2025).
8. *Brooks D. R.* C programming: the essentials for engineers and scientists. Springer Science & Business Media, 2012. 490 p.
9. Arm a-profile a64 instruction set architecture. EOR (bitwise exclusive-OR (vector)). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/EOR--vecto

r---Bitwise-exclusive-OR--vector--?lang=en (date of access: 13.09.2025).

10. Arm a-profile a64 instruction set architecture. LD3 (multiple structures). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/LD3--multiple-structures---Load-multiple-3-element-structures-to-three-registers-?lang=en (date of access: 29.07.2025).
11. Arm a-profile a64 instruction set architecture. SMLAL, SMLAL2 (vector). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/SMLAL--SMLAL2--vector---Signed-multiply-add-long--vector--?lang=en (date of access: 29.07.2025).
12. Arm a-profile a64 instruction set architecture. SMLSL, SMLSL2 (vector). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/SMLSL--SMLSL2--vector---Signed-multiply-subtract-long--vector--?lang=en (date of access: 29.07.2025).
13. Arm a-profile a64 instruction set architecture. DUP (duplicate vector element to vector or scalar (vector)). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/DUP--element---Duplicate-vector-element-to-vector-or-scalar-?lang=en (date of access: 13.09.2025).
14. Arm a-profile a64 instruction set architecture. EXT (extract vector from pair of vectors). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/EXT--Extract-vector-from-pair-of-vectors-?lang=en (date of access: 29.07.2025).
15. Arm a-profile a64 instruction set architecture. SXTL, SXTL2 (signed extend long). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/SXTL--SXTL2--Signed-extend-long--an-alias-of-SSHLL--SSHLL2-?lang=en (date of access: 13.09.2025).

16. Arm a-profile a64 instruction set architecture. SCVTF (signed integer convert to floating-point (scalar)). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/SCVTF--scalar--integer---Signed-integer-convert-to-floating-point--scalar--?lang=en (date of access: 29.07.2025).

17. Arm a-profile a64 instruction set architecture. FMUL (floating-point multiply (vector)). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/FMUL--vector---Floating-point-multiply--vector--?lang=en (date of access: 13.09.2025).

18. Arm a-profile a64 instruction set architecture. FCVTAS (floating-point convert to signed integer, rounding to nearest with ties to away (scalar)). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/FCVTAS--scalar---Floating-point-convert-to-signed-integer--rounding-to-nearest-with-ties-to-away--scalar--?lang=en (date of access: 29.07.2025).

19. Arm a-profile a64 instruction set architecture. SQXTN, SQXTN2 (signed saturating extract narrow). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/SQXTN--SQXTN2--Signed-saturating-extract-narrow-?lang=en (date of access: 29.07.2025).

20. Arm a-profile a64 instruction set architecture. ST3 (multiple structures). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/ST3--multiple-structures---Store-multiple-3-element-structures-from-three-registers-?lang=en (date of access: 29.07.2025).

21. Argparse — parser for command-line options, arguments and subcommands. Python documentation. URL: https://docs.python.org/3/library/argparse.html (date of access: 05.08.2025).

22. *Dongarra J. J.*, *Hinds A. R.* Unrolling loops in fortran. Software: practice and experience. 1979. Vol. 9: 3. P. 219–226. URL: https://doi.org/10.1002/spe.4380090307 (date of access: 05.08.2025).

23. ARM cortex-a (ARMv7-a) series programmer's guide. compiler-optimizations: loop-unrolling. Arm Developer. URL: https://developer.arm.com/documentation/den0013/d/Optimizing-Code-to-Run-on-ARM-Processors/Compiler-optimizations/Loop-unrolling (date of access: 05.08.2025).

24. *Sarkar V.* Optimized unrolling of nested loops. Proceedings of the 14th international conference on Supercomputing. ICS '00. 2000. С. 153–166. URL: https://doi.org/10.1145/335231.335246 (дата звернення: 29.07.2025).

25. *Pyeatt L. D.* Modern assembly language programming with the ARM processor. Amsterdam : Elsevier, 2024. 554 p.

26. *Thomas M.* Programmer's guide for ARMv8-a. Arm Community. 2015. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/programmer-s-guide-for-armv8-a (date of access: 13.09.2025).

27. *Pratt W. K.* Introduction to digital image processing. Boca Raton : CRC Press, 2013. 750 p.

28. *Cardoso J. M. P.*, *Coutinho J. G. d. F.*, *Diniz P. C.* Embedded computing for high performance: efficient mapping of computations using customization, code transformations and compilation. Elsevier Science & Technology Books, 2017. 322 p.

29. *Shin J.*, *Hall M.*, *Chame J.* Superword-level parallelism in the presence of control flow. International Symposium on Code Generation and Optimization. 2005. P. 165–175. URL: https://doi.org/10.1109/CGO.2005.33 (date of access: 05.08.2025).

30. *Шевченко А. К.* Підвищення швидкодії операції згортки RGB зображень з використанням алгоритму кластеризації ядер згортки для архітектури процесорів ARM64. Problems of informatization and management. 2025. Т. 1: 81. С. 140–149. URL: https://doi.org/10.18372/2073-4751.81.20144 (дата звернення: 18.08.2025).

31. *Kormanyos C.* Real-time C++ : Efficient object-oriented and template microcontroller programming. 3rd ed. Berlin, Heidelberg : Springer, 2018. 426 p. URL: https://doi.org/10.1007/978-3-662-56718-0.

32. *Bryant R. E.*, *O'Hallaron D. R.* Computer systems: a programmer's perspective. 3rd ed. Boston : Pearson Education, 2015. 1120 p.

33. *Hennessy J. L.*, *Patterson D. A.*, *Asanović K.* Computer architecture: a quantitative approach. 5th ed. Waltham : Morgan Kaufmann/Elsevier, 2012. 858 p.

34. *Jacob B.*, *Wang D.*, *Ng S.* Memory systems: cache, DRAM, disk. Burlington : Morgan Kaufmann, 2010. 1017 p.

35. *Kelefouras V.*, *Keramidas G.* Design and implementation of deep learning 2d convolutions on modern cpus. IEEE trans. parallel distrib. syst. 2023. Vol. 34: 12. P. 3104–3116. URL: https://doi.org/10.1109/TPDS.2023.3322037 (date of access: 05.08.2025).

36. P-Im2col: simple yet efficient convolution algorithm with flexibly controlled memory overhead. A. V. Trusov, E. E. Limonova, D. P. Nikolaev, V. V. Arlazarov. Ieee access. 2021. Vol. 9. P. 168162–168184. URL: https://doi.org/10.1109/ACCESS.2021.3135690 (date of access: 13.09.2025).

ДОДАТОК Б

МАТРИЦЯ ДОКАЗОВОСТІ ПУНКТІВ НАУКОВОЇ НОВИЗНИ

Б.1 Призначення додатку та структура матриці

У цьому додатку подано систематизоване відображення кожного з пунктів наукової новизни дисертаційної роботи на конкретні докази у тексті: формули, алгоритми (блок-схеми), таблиці та результати експериментальних досліджень. Мета додатку — забезпечити перехресні посилання, що дозволяють для кожного пункту наукової новизни швидко локалізувати у тексті дисертації відповідний теоретичний апарат, алгоритмічну реалізацію та експериментальне підтвердження.

Кожен науковий результат оформлено за п’ятичастинною схемою:

1. **Що саме отримано** — стисле формулювання сутності отриманого наукового результату.
2. **Аналогічні рішення та відомі підходи** — огляд споріднених підходів у науковій літературі, що використовуються як відправна точка для зіставлення.
3. **Відмінність від відомих підходів** — аналіз ключових відмінностей запропонованого підходу від вже відомих.
4. **Розмежування методу та реалізації** — виокремлення того, що є новим на рівні методу або технології (загальні принципи, формули, алгоритми), а що — реалізаційним наслідком (конкретні маски, бібліотеки, програмні продукти).
5. **Локалізація доказової бази у тексті** — таблиця перехресних посилань на формули, алгоритми (блок-схеми), таблиці, експерименти та програмні реалізації, що доводять відповідний пункт.

Перед розгорнутим описом за п’ятичастинною схемою для кожного пункту наводиться зведена матриця (підрозд. Б.2), що дозволяє оглянути усі результати на одній сторінці.

Відповідно до структури наукової новизни роботи, наведені нижче результати згруповано за трьома категоріями:

- **Уперше** — два результати: (1) розроблено та теоретично обґрунтовано

метод швидкодіючої ОЗ (ШОЗ) для обробки 8-бітних RGB ЦЗ з використанням 16-бітних цілочисельних операцій (п. 3), теоретичним підґрунтям якого є математична модель ОЗ для 16-бітних цілочисельних обчислень (п. 5); (2) запропоновано формалізовану нотацію для опису векторизованих обчислень SIMD NEON64 (п. 2).

- **Удосконалено** — технологію обробки ЦЗ на комп'ютерах, що підтримують паралелізм обчислень, шляхом запропонованої ІТ обробки зображень та відео з використанням ШОЗ для архітектури ARM64 (інтеграція запропонованих методу, нотації й операторів у цілісну технологію), доказовою базою чого є методика експериментальної оцінки швидкодії (п. 4).
- **Набули подальшого розвитку** — лінійні оператори на основі лінійних комбінацій *B*-сплайнів, що є близькими до інтерполяційних у середньому, — отримані для 16-бітних цілочисельних обчислень із гарантією відсутності переповнення (п. 1).

Таким чином, категорія «уперше» містить *два* результати — метод ШОЗ та формалізовану нотацію, — що дослівно відповідає формулюванням наукової новизни у вступі та переліку загальних висновків дисертації. Пункти 4 і 5 окремими категоріями новизни не є: п. 5 (математична модель) входить до складу «уперше» як теоретичне обґрунтування методу ШОЗ (п. 3), а п. 4 (методика експериментальної оцінки швидкодії) є доказовою базою категорії «удосконалено».

Для збереження наскрізної нумерації доказової бази нижче результати подано у наскрізному порядку (1–5), а їх належність до категорій новизни зазначено у заголовку кожного пункту та у зведеній матриці.

Б.2 Зведена матриця доказовості

У зведеній матриці (табл. Б.1) подано стисле відображення усіх пунктів наукової новизни на ключові докази у тексті дисертації — формули, алгоритми (блок-схеми), таблиці та результати експериментальних досліджень. Деталізоване розкриття кожного пункту наведено у наступних підрозділах (Б.3–Б.7).

Таблиця Б.1 — Зведена матриця доказовості пунктів наукової новизни

№	Результат (стисло)	Ключові формули	Алгоритми / блок-схеми	Експеримент / реалізація
1	*(Набули подальшого розвитку.)* Лінійні оператори 16-бітних цілочисельних обчислень для субсмугової фільтрації, контрастування, цифрової стабілізації та рекурентного 2× масштабування ЦЗ, у т.ч. адаптовані для беззнакових обчислень.	(2.1–2.3); (2.4–2.13); (2.16–2.23)	інстанціювання масок	табл. 4.8, 4.9 (zoom 2×); функції субсмугова фільтрація (2.4, 2.5); контрастування (2.7, 2.8); цифрова стабілізація зображень (2.9, 2.10); рекурентне двократне збільшення масштабу (zoomUP_2__NEON64(…)) (2.11, 2.12)
2	*(Уперше.)* Формалізована (структурована) нотація для опису векторизованих обчислень SIMD NEON64: формалізація іменування векторних регістрів, типів даних, діапазонів індексів, ключових векторних операцій.	(А.12–А.17); (А.18, А.22, А.29, А.32, А.40, А.45)	рис. А.4–А.6 (записані за допомогою нотації)	табл. А.1 (діапазони векторів); реалізація `libconv_lib.so` (повністю на основі нотації)
3	*(Уперше.)* Метод швидкодіючої ОЗ (ШОЗ) для обробки 8-бітних RGB ЦЗ з використанням 16-бітних цілочисельних обчислень NEON64, у т.ч. з оптимізаціями для розріджених ЯЗ (через предикт-матрицю *index_map*).	(А.1–А.11) (формалізація ШОЗ); (А.18–А.35) (ВСД); (А.36–А.44) (нормалізація); (А.45–А.51) (кластеризація)	рис. А.3–А.6	табл. 4.3, 4.4 (C++ – до 11,1×); табл. 4.6, 4.7 (Python); табл. 4.8, 4.9 (zoom 2×); `conv_RGB_NEON`
4	*(Доказова база категорії «Удосконалено».)* Методика експериментальної оцінки та порівняння швидкодії ВОЗ на платформах ARM64 (із автоматизацією тестування для 14 розмірів ЯЗ × 8 роздільностей ЦЗ).	(4.1–4.8); (4.9)	рис. 4.1–4.6 (UML/SDL діаграми)	табл. 4.1, 4.2; усі результати розд. 4.6
5	*(Теоретичне підґрунтя п. 3 категорії «Уперше».)* Математична модель ОЗ для 16-бітних цілочисельних обчислень при обробці 8-бітних RGB ЦЗ та 8-бітних ЯЗ — з умовою коректності, моделюванням діапазонів типів та методикою адаптації знакових ЯЗ до беззнакових ЦО.	(2.1–2.3); (2.15–2.19); (2.24–2.28); (А.1–А.4)	— (теоретична модель)	Висновки до розд. 2 (п.п. 1, 4, 5); основа для ШОЗ (табл. Б.5)

Б.3 Пункт 1 (Набули подальшого розвитку). Лінійні оператори 16-бітних цілочисельних обчислень

Б.3.1 Що саме отримано (А)

Отримано сукупність лінійних операторів згортки на основі цілочисельних ЯЗ, ядра яких задовольняють умові коректності 16-бітних цілочисельних обчислень ((2.3) — відсутність переповнень на проміжних етапах множення-акумуляції), для:

1. субсмугової фільтрації — виділення низько- та високочастотних складових на основі лінійних комбінацій *B*-сплайнів 2-го та 3-го порядків (маски 3×3 та 5×5, (2.4–2.6));
2. контрастування — з використанням псевдозворотного оператора до низькочастотного фільтра (2.7, 2.8);
3. цифрової стабілізації мікрорухів камери (підвищення різкості) — з можливістю обрати ступінь підвищення різкості (2.9, 2.10);
4. рекурентного двократного збільшення лінійних розмірів ЦЗ зі згладжуванням або без — через чотири спеціалізовані ЯЗ (A, B, C, D), що формують 4 проміжні ЦЗ з подальшим об’єднанням за принципом «шахової дошки» (2.11–2.13);
5. рекурентного двократного зменшення лінійних розмірів ЦЗ (узагальнений функціонал з можливістю обрати маску для попередньої фільтрації, (2.14));
6. адаптації ЯЗ зі змішаною знаковістю до беззнакових 16-бітних обчислень шляхом зсуву елементів маски на γ_{bias} з компенсаційною нормалізацією через бокс-фільтр ((2.17–2.19); конкретні беззнакові маски — (2.20–2.23)).

Сформульовано та обґрунтовано умову відповідності ЯЗ для 16-бітних цілочисельних обчислень (2.2, 2.3), а саме обмеження суми абсолютних значень елементів ЯЗ (до ділення на знаменник, винесений за маску). Запропоновано також підхід до обробки ЯЗ, елементи якого виходять за межі 8-бітного діапазону — через декомпозицію 32-бітного ЯЗ на чотири 8-бітні компоненти зі зсувами (2.24–2.28).

Б.3.2 Аналогічні рішення та відомі підходи (Б)

У науковій літературі та програмних продуктах, що адресують споріднені задачі обробки ЦЗ, можна виділити такі групи аналогів:

- **Класичні маски згортки** (фільтри Гауса, Лапласа, Собеля, Превітта тощо) — широко представлені у фундаментальних працях з обробки ЦЗ [1–3]. Зазвичай використовуються у форматі з плаваючою комою, без обмежень на діапазон значень елементів ЯЗ.
- **B-сплайнові підходи** до побудови контрастних фільтрів та операторів масштабування ЦЗ, розроблені Приставкою П.О. та співавторами [4–10] — є теоретичною базою побудови запропонованих у роботі масок, проте у згаданих працях розглядаються переважно обчислення з плаваючою комою.
- **Реалізація** `cv::filter2D(…)` **з бібліотеки OpenCV** [11–13] — універсальна функція згортки, що внутрішньо використовує послідовність перетворень типів 8u → 16s → 32s → 32f для уникнення переповнень (підрозд. 2.3, підрозд. 1.4).
- **Функції** `NEConvolution{N}x{N}::run(…)` **з ARM Compute Library** [14] — спеціалізовані SIMD-реалізації згортки для процесорів ARM, проте тільки для фіксованих розмірів ЯЗ та переважно для 32-бітних обчислень.
- **Згорткові ядра у форматі fixed-point** для застосувань у SIMD-обчисленнях, описані у [15–18] — розглядають загальні принципи цілочисельних обчислень, але без прив’язки до конкретних масок для ОЦЗ.

Б.3.3 Відмінність від відомих підходів (В)

Запропоновані лінійні оператори відрізняються від наявних у літературі за такими ключовими аспектами:

1. **Цілочисельність 16-бітної арифметики як проектне обмеження.** На відміну від класичних реалізацій, що використовують float-обчислення [1, 11], або від цілочисельних реалізацій без формальних обмежень на елементи ЯЗ, запропоновані маски сформовано з явно сформульованою умовою відповідності — сума абсолютних значень елементів ЯЗ (до ділення на знаменник) не перевищує 127 (2.3).
2. **Адаптація знакових масок до беззнакових обчислень з компенсацій-**

ною нормалізацією. Запропоновано методику зсуву елементів ЯЗ на γ_{bias} (2.17, 2.18) для зведення ОЗ до беззнакових 16-бітних обчислень з подальшою компенсацією через бокс-фільтр (2.19). Це не зустрічається у згаданих аналогах і дозволяє використовувати швидші інструкції UMLAL/UMLAL2 замість SMLAL/SMLAL2.

3. **Розкладання ЯЗ з елементами понад 8-бітний діапазон** (2.24–2.28) на чотири байтові компоненти зі зсувами — зберігає переваги 16-бітних проміжних обчислень навіть для широкосмугових ЯЗ.
4. **Спеціалізація під задачі ОЦЗ.** Кожна маска отримана з конкретною прикладною метою (субсмугова фільтрація, контрастування, стабілізація, масштабування) і пройшла перевірку на відповідність 16-бітним цілочисельним обчисленням (ЦО), що відрізняє цей результат від загальних принципів fixed-point обчислень [15, 16].

Б.3.4 Розмежування методу та реалізації (Г)

Що є новим на рівні методу:

- Умова відповідності маски згортки для 16-бітних цілочисельних обчислень — (2.3).
- Методика адаптації знакових ЯЗ до беззнакових обчислень із компенсаційною нормалізацією — (2.17–2.19).
- Підхід розкладання 32-бітних ЯЗ на байтові компоненти — (2.25–2.28).

Що є реалізаційним наслідком:

- Конкретні чисельні маски (2.4–2.13) (для знакових ЦО) та (2.20–2.23) (для беззнакових ЦО) — це інстанціювання загального методу для конкретних прикладних задач.
- Програмна реалізація операторів: субсмугова фільтрація, контрастування та цифрова стабілізація виконуються *універсальним* обчислювальним ядром `conv_RGB_NEON(...)` бібліотеки `libconv_lib.so`, якому відповідний оператор передається як ЯЗ (окремих функцій під кожен оператор не передбачено — оператор задається саме ЯЗ); рекурентне двократне збільшення масштабу оформлено процедурою `zoomUP_2__NEON64(...)` рівня Python-застосунку, що чотири рази викликає `conv_RGB_NEON(...)` з масками A, B, C, D — це програмний прояв методу для платформи ARM64.

Б.3.5 Локалізація доказової бази у тексті дисертації (Д)

Таблиця Б.2 — Локалізація доказової бази для пункту наукової новизни №1

Тип доказу	**Локація**	**Що саме**
Загальний оператор ОЗ	Розд. 2.1	(2.1)
Умова коректності 16-біт ЦО	Розд. 2.1	(2.2, 2.3)
Маски 3×3 субсмугової фільтрації (B-сплайни 2-го та 3-го порядку)	Розд. 2.2	(2.4, 2.5)
Маски 5×5 згладжування	Розд. 2.2	(2.6)
Маска 5×5 контрастування + псевдозворотне співвідношення	Розд. 2.2	(2.7, 2.8)
Маски 5×5 цифрової стабілізації / підвищення різкості	Розд. 2.2	(2.9, 2.10)
Оператори 2× збільшення масштабу (A, B, C, D) зі згладжуванням / без	Розд. 2.2	(2.11–2.13)
Узагальнений функціонал двократного зменшення масштабу	Розд. 2.2	(2.14)
Стандартні діапазони знакових / беззнакових типів	Розд. 2.3	(2.15)
Умова відповідності для беззнакових ЦО, методика зсуву γ_{bias} та компенсації	Розд. 2.4	(2.16–2.19)
Беззнакові маски (3×3, 5×5, 2× масштабування)	Розд. 2.4	(2.20–2.23)
Розкладання 32-бітних ЯЗ на байтові компоненти зі зсувами	Розд. 2.5	(2.24–2.28)
Висновки за розділом 2	Розд. 2.6	п.п. 1–5
Реалізація операторів (NEON64)	Розд. 3.4	`conv_RGB_NEON` (оператор = ЯЗ); масштабування — `zoomUP_2__NEON64` (Python); відповідність мнемонік ідентифікаторам коду — табл. 3.2
Інтеграція у Python-API	Розд. 3.6	табл. 3.4
Експериментальне підтвердження операції 2× масштабування	Розд. 4.6.3	табл. 4.8, 4.9 (прискорення до 6,3× (OPi5) та 6,2× (RPi4))
Підтвердження у висновках дисертації	Висновки	п. 2

Б.4 Пункт 4 (доказова база категорії «Удосконалено»). Методика експериментальної оцінки швидкодії на ARM64

Б.4.1 Що саме отримано (А)

Розроблено та апробовано методику експериментальної оцінки і порівняння швидкодії програмних реалізацій ВОЗ на платформах з архітектурою ЦП ARM64. Методика складається з таких компонентів:

1. **Матрична схема накопичення вибірки** — систематичний перебір 14 розмірів ЯЗ (від 2×2 до 15×15, (4.7)) на 8 роздільностях ЦЗ (від 256×144 до 3840×2160) (див. табл. 4.2), що дає 112 комбінацій параметрів і дозволяє побудувати матриці швидкодії розміром 8×14.
2. **Багатопотокова стратегія зчитування кадрів** зі вхідного відеопотоку для усунення ефекту «вузького місця» введення-виведення (з оптимальною кількістю 3 потоки за замовчуванням, (4.1)) — відповідає виявленому парадоксу, коли час обробки ЦЗ оптимізованою ВОЗ стає меншим, ніж час її зчитування з потоку.
3. **Адаптивне керування пам'яттю** з обмеженням 70 % доступної ОЗП, динамічним розподілом цього бюджету між обробленими та необробленими кадрами і розподілом черг між потоками-зчитувачами (4.2–4.4).
4. **Збереження цілісності відеопотоку** через супровід кожного кадру у черзі його порядковим номером, що гарантує впорядкованість результату обробки (4.5).
5. **Уніфікований показник швидкодії** (FPS, (4.6)), що дозволяє безпосередньо співставляти різні реалізації ВОЗ в умовах, наближених до режиму реального часу.
6. **Статистична обробка вибірки** — з обчисленням як середнього (4.8), так і медіани (4.9) для кожної комбінації параметрів (не менше 150 замірів FPS).
7. **Подвійне середовище вимірювання** — з прямого виклику C/C++ функцій (через виконуваний файл) та через Python-обгортку (через бібліотеку `ctypes`), що дозволяє оцінити вплив накладних витрат високорівневого інтерфейсу.
8. **Модульна автоматизація** тестового середовища у вигляді ієрархії про-

цедур (загальна процедура тестування, обробка ОЗ, генерація ЯЗ, підготовка директорії результатів, запуск тестової команди, очікування завершення) з відображенням алгоритмів через UML/SDL-діаграми активності рис. 4.1–4.6.

9. **Стандартизовані умови тестування** на двох поширених SBC-платформах (Orange Pi 5 Pro з SoC Rockchip RK3588S та Raspberry Pi 4 з SoC Broadcom BCM2711, табл. 4.1) з ідентичним налаштуванням ОС, відключенням GUI, фіксацією CPU governor.
10. **Формат результату** — три типи матриць (медіанних показників FPS для кожного алгоритму, відносного прискорення S та різниці D), що дозволяє багатовекторно інтерпретувати результати тестування.

Б.4.2 Аналогічні рішення та відомі підходи (Б)

У сфері оцінки швидкодії програмного забезпечення можна виділити такі групи аналогів:

- **Профілювальники загального призначення** — Valgrind / Callgrind [19] та Valgrind / Massif [20] — забезпечують детальний аналіз виконання, проте працюють у віртуальному середовищі і не дають точної картини швидкодії на цільовій ARM64-платформі (підрозд. 1.3.6).
- **Універсальні бенчмаркінгові фреймворки** для оцінки швидкодії алгоритмів машинного навчання та обробки сигналів — MLPerf [21] та споріднені бенчмаркінгові методики [22] — розраховані на широкий клас задач і не специфікують умови, важливі для ОЗ ЦЗ.
- **Спеціалізовані бенчмарки об'єктно-розпізнавальних рішень** на одноплатних комп'ютерах [23] — адресують дотичну, але іншу задачу (детектування об'єктів) і не зосереджуються на ізольованому вимірюванні ВОЗ.
- **Внутрішні бенчмарки бібліотеки OpenCV** (вбудовані тести `opencv_perf_imgproc`, `opencv_test_imgproc`) — орієнтовані на регресійне тестування власного коду OpenCV і не покривають порівняння з зовнішніми реалізаціями.
- **Бенчмарки бібліотек ARM Compute Library та inference-фреймворків** [14, 24, 25] — зазвичай вимірюють швидкодію CNN-операторів цілком, без ізольованого оцінювання ОЗ.

- **Загальні принципи інженерії бенчмарків** високошвидкодійних обчислень [26–30] — надають загальні рекомендації, але не пропонують конкретної методики для ВОЗ на ARM64.

Б.4.3 Відмінність від відомих підходів (В)

Запропонована методика відрізняється від наявних у літературі за такими ключовими аспектами:

1. **Спеціалізація під ВОЗ на ARM64.** На відміну від універсальних профілювальників та бенчмаркінгових фреймворків, методика проектована саме для оцінки швидкодії ВОЗ в умовах, наближених до реальних застосувань (обробка відеопотоку), з прив'язкою до архітектури ARM64 та її SIMD-розширення NEON64.
2. **Матричний характер вибірки.** Систематичний перебір $14 \times 8 = 112$ комбінацій параметрів (4.7) та (табл. 4.2) дозволяє виявити неочевидні залежності — наприклад, перехід `cv::filter2D(…)` на FFT-алгоритм для ЯЗ 8×8 і більше, що проявляється як різке падіння швидкодії (підтверджено експериментально, табл. 4.3). Універсальні бенчмарки таку картину виявити не можуть.
3. **Багатопотокова стратегія зчитування з оптимальною кількістю 3 потоки.** Виявлений парадокс «обробка швидша за зчитування» (підрозд. 4.5) і запропоноване його розв'язання (4.1) є оригінальним методичним внеском, що відрізняє цей підхід від однопотокових бенчмарків (зокрема, від поведінки Python-версії, де такий ефект не нівельовано і він явно проявляється у показниках, табл. 4.6, 4.7).
4. **Адаптивне керування пам'яттю на основі частки доступної ОЗП.** Підхід (4.2–4.4) дозволяє виконувати тестування на SBC з різним обсягом ОЗП (4 ГБ Raspberry Pi 4 vs 16 ГБ Orange Pi 5 Pro) без модифікації параметрів, що забезпечує переносимість методики.
5. **Подвійне середовище вимірювання** (C/C++ та Python через `ctypes`) дозволяє кількісно оцінити «штраф» високорівневого інтерфейсу окремо від алгоритмічних оптимізацій. У наявних бенчмарках такий розподіл, як правило, не виконується.
6. **Виключення впливу збереження результатів** (відмова від використання прапорця `-o`) та фокусування виключно на ефективності ВОЗ — мето-

дичне рішення, що ізолює вимірюваний показник від впливу драйверів запису та кодеків (підрозд. 4.5.2).

7. **Платформна валідація.** Тотожні методичні умови застосовуються до двох SBC різних виробників та різних поколінь архітектури ARMv8 (табл. 4.1), що дозволяє продемонструвати платформну незалежність як самих оптимізацій, так і методики оцінки.

Б.4.4 Розмежування методу та реалізації (Г)

Що є новим на рівні методу:

- Матрична схема перебору параметрів (14 розмірів ЯЗ $\times$ 8 роздільностей ЦЗ) як методичний фундамент — (4.7).
- Принцип багатопотокового зчитування з усуненням I/O bottleneck — (4.1).
- Принципи адаптивного керування пам'яттю — (4.2–4.4).
- Принцип збереження цілісності черги через впорядковану нумерацію — (4.5).
- Стандартизація показника швидкодії та його статистичної обробки — (4.6, 4.8, 4.9).
- Архітектура тестового середовища у вигляді ієрархії модульних процедур, формалізована UML/SDL-діаграмами активності — рис. 4.1–4.6.

Що є реалізаційним наслідком:

- Конкретні Shell-скрипти (`run.test.sh`, `run.test.py.sh`) з функцією `run_test(…)` — це програмний прояв методики.
- Python-клас `FpsCounter` та утилітарні функції збору метрик через `cv2.getT` — реалізаційні засоби обчислення показника FPS;
- Конкретні значення параметрів за замовчуванням (3 потоки, 70 % ОЗП, не менше 150 замірів) та конкретні набори тестових відеофайлів (табл. 4.2).
- Вибір SBC-платформ для апробації (Orange Pi 5 Pro, Raspberry Pi 4) — табл. 4.1.
- Вибір базового алгоритму-конкурента (`cv::filter2D(…)` з OpenCV) для порівняльного аналізу.

Б.4.5 Локалізація доказової бази у тексті дисертації (Д)

Таблиця Б.3 — Локалізація доказової бази для результату №4 (доказова база категорії «Удосконалено»)

Тип доказу	**Локація**	**Що саме**
Опис експерименту, мета та критерії	Розд. 4.1	14 розмірів ЯЗ, 8 роздільностей ЦЗ, 150 замірів FPS
Апаратна база (характеристики SBC)	Розд. 4.2	табл. 4.1
Програмне налаштування (ОС, GUI, governor)	Розд. 4.3	—
Інструментарій (компілятори, віртуалізація, профілювальники)	Розд. 4.4	—
Оптимізація багатопотокового зчитування	Розд. 4.5.1	(4.1)
Адаптивне керування пам’яттю	Розд. 4.5.1	(4.2–4.4)
Цілісність відеопотоку (нумерація кадрів)	Розд. 4.5.1	(4.5)
Показник FPS та статистична обробка	Розд. 4.5.1	(4.6, 4.8, 4.9)
Діапазон розмірів ЯЗ (матрична схема)	Розд. 4.5.1	(4.7)
Перелік тестових відеофайлів за роздільностями	Розд. 4.5.1	табл. 4.2
Тестування C/C++ функцій (опис прапорців CLI)	Розд. 4.5.2	—
Тестування Python-обгортки (через `ctypes`)	Розд. 4.5.2	—
Загальна процедура тестування швидкодії	Розд. 4.5.3	рис. 4.1
Процедура `processImgConvKernelOperation()`	Розд. 4.5.3	рис. 4.2
Процедура `generateKernel()`	Розд. 4.5.3	рис. 4.3
Процедура `prepareResultsDirectory()`	Розд. 4.5.3	рис. 4.4
Процедура `runTestCommand()`	Розд. 4.5.3	рис. 4.5
Процедура `waitForTestCompletion()`	Розд. 4.5.3	рис. 4.6
Застосування методики (C++ результати)	Розд. 4.6.1	табл. 4.3, 4.4
Застосування методики (Python результати)	Розд. 4.6.2	табл. 4.6, 4.7
Застосування методики (масштабування)	Розд. 4.6.3	табл. 4.8, 4.9
Висновки за розділом 4	Розд. 4.7	п.п. 1–6
Підтвердження у висновках дисертації	Висновки	п. 8

Б.5 Пункт 2 (Уперше). Формалізована нотація для опису SIMD NEON64 обчислень

Б.5.1 Що саме отримано (А)

Розроблено формалізовану (структуровану) математичну нотацію для опису векторизованих обчислень SIMD NEON64, яка дозволяє абстрагуватися від конкретних асемблерних інструкцій та представляти векторні операції у вигляді математичних виразів. Нотація включає такі компоненти:

1. **Базова формалізація іменування векторних регістрів** як параметризованої множини $VC_kI_e^{dtype} = \{vC_kI_{e,i}^{dtype} \mid i \in \mathcal{I}_{start,end}^{dtype}\}$ (А.12), що інтегрально враховує: позначення колірного каналу C_k, діапазон індексів вказівника I_e, тип даних $dtype \in \{u8, s8, u16, s16, u32, s32, f32\}$ та впорядковану множину індексів елементів. Ця базова конструкція є самодокументованою — ім'я типу V_R0007^{u16} несе у собі семантику (R-канал, біти 0–7, тип u16) безпосередньо у позначенні.
2. **Функція кількості елементів** $elems(dtype)$ (А.13, А.14), що відображає тип даних у кількість елементів, які поміщаються у 128-бітному регістрі ВР (16 для u8/s8, 8 для u16/s16, 4 для u32/s32/f32, 2 для u64/s64/f64); конкретні значення зведено у табл. А.1.
3. **Множини індексів елементів**: повна $\mathcal{I}_{0,elems(dtype)-1}^{dtype}$ (А.15, А.16) та допоміжні функції діапазону $i_all(dtype)$, $i_1hlf(dtype)$, $i_2hlf(dtype)$ (А.17) для компактного запису повного, нижнього та верхнього діапазонів.
4. **Дві еквівалентні форми запису** діапазонів — функціональна ($i = i_all(dtype)$) та класична математична ($\forall i \in \mathcal{I}_{0,elems(dtype)-1}^{dtype}$), вибір яких визначається доцільністю.
5. **Сукупність базових векторних операцій**, записаних у нотаційному вигляді:
   - обнуління через XOR (А.18);
   - завантаження з пам'яті: `ld3` (А.22–А.24), `ld1` (А.26, А.27);
   - формування вектора маски «позитивності» (А.28);
   - векторне множення-акумуляція `smlal/smlsl/smlal2/smlsl2` (А.29) — ядро обчислення ВСД;
   - функція циклічного зсуву $CShL$ (А.32–А.34);

- функція насичення $maxmin_vdt$ (А.40) — абстрагує родину інструкцій SQXTN/UQXTN;
- конвертація типів s16→s32→f32→s32→s16→u8 (А.36–А.43);
- збереження у пам'ять `st3` (А.44);
- кластеризація розрідженого ЯЗ через предикт-матрицю $index_map$ (А.45).

6. **Демонстрація потужності** нотації — усі блок-схеми ШОЗ (рис. А.3–А.6) повністю записано через елементи нотації без використання асемблерних мнемонік.

Концептуально нотація заповнює методологічну прогалину між математичним описом алгоритмів і його програмною реалізацією через SIMD-інтрінсики чи Inline Assembly, надаючи проміжний абстрактний рівень, придатний для формального аналізу та публікаційного викладення.

Б.5.2 Аналогічні рішення та відомі підходи (Б)

У літературі та програмних інструментах розрізняють такі групи аналогів:

- **Офіційна документація ARM A-profile Architecture Reference Manual** [31–33] — формальна, але описує окремі асемблерні мнемоніки (XOR, LD3, SMLAL тощо) на рівні псевдокоду інструкції; не пропонує узагальненої нотації для алгоритмів, що складаються з багатьох інструкцій.
- **ARM NEON Intrinsics** [34, 35] — C-рівневий API виду `vmlaq_s16(...)`, `vld3q_u8(...)`; програмна абстракція, не математична. Зручна для розробників, але не для наукового опису алгоритмів.
- **OpenCV Universal Intrinsics** [13, 36] — кросплатформений C++ шар (типи `v_uint8x16`, функції `v_load`, `v_mul`, `v_pack` тощо). Найближче за духом до запропонованої нотації, проте залишається імперативним і прив'язаним до C++.
- **Arm Compute Library (ACL)** [14] — C++ ядра для задач комп'ютерного зору. Реалізаційний, не нотаційний рівень.
- **Підручники з ARM64 асемблера та системного програмування** [37–41] — розглядають інструкції на низькому рівні без узагальненої формалізації для алгоритмів.
- **Наукова література з SIMD-парадигми** [30, 42–46] — описує паралелізм даних на абстрактному (таксономічному) рівні Флінна, але не формалізує

конкретні операції векторних регістрів.

- **Літературні джерела щодо реалізації згорткових алгоритмів** [47–50] — описують алгоритмічні підходи у псевдокоді або математичних виразах класичної форми, без формалізації векторних операцій.

Як експліцитно зазначено у самому тексті дисертації (Розд. 3.1.1), *«стандартизованих підходів до формалізації векторизованих обчислень у літературі практично немає»*. Це підкреслює прогалину, яку заповнює запропонована нотація.

Б.5.3 Відмінність від відомих підходів (В)

Запропонована нотація відрізняється від наявних аналогів за такими ключовими аспектами:

1. **Математичний (множинно-теоретичний), а не програмний характер.** На відміну від Intrinsics або Universal Intrinsics, що є C/C++ API, нотація формулюється у термінах множин $\mathcal{I}^{dtype}_{start,end}$ та функцій $elems$, i_all, i_1hlf, i_2hlf (А.13, А.15, А.17). Це дозволяє формальний аналіз коректності, складності та властивостей алгоритмів.
2. **Архітектурна незалежність.** Нотація не прив'язана до конкретних асемблерних інструкцій ARM64. Принципи (типізація, діапазони індексів, узагальнені операції $CShL$, $maxmin_vdt$) можуть бути перенесені на інші SIMD-архітектури (Intel SSE/AVX, RISC-V Vector Extension, ARM SVE), де вибудовується аналогічна структура регістрів та операцій.
3. **Параметризація типу даних як першокласного об'єкту.** У нотації $dtype$ присутній як змінна у формулах, що дозволяє описати поведінку для всіх 7 підтримуваних типів (u8, s8, u16, s16, u32, s32, f32) однією формулою (А.12). У C-Intrinsics для кожного типу існує окрема функція (`vmla_u8`, `vmla_s16`, `vmla_f32` тощо), що загромаджує опис алгоритму.
4. **Експліцитні діапазони індексів.** $\mathcal{I}^{dtype}_{start,end}$ та функції i_1hlf/i_2hlf дозволяють чітко відмічати, з якою половиною регістра працює інструкція. Це критично для коректного запису пар `smlal`/`smlal2` і подібних, що оперують різними половинами вхідного вектора (А.29).
5. **Самодокументований стиль іменування** [51]. Імена векторів виду V_R0007^{u16} безпосередньо несуть семантику (канал, відступ від адреси, тип), що недосяжно у класичних позначеннях $v0, v1, \ldots$.

6. **Узагальнення родин інструкцій.** Функції $CShL$ (A.32) та $maxmin_vdt$ (A.40) узагальнюють у єдиний абстрактний оператор сімейство споріднених інструкцій (відповідно: ext-варіанти; SQXTN/UQXTN/SQXTN2/UQXTN2). Це робить запис компактнішим і виразнішим.
7. **Дозволяє формальне доведення коректності** векторизованих алгоритмів, що недосяжно для імперативних описів. Як зазначено у Розд. 3.1.5, нотація створює міст між математичним описом алгоритмів та їх практичною реалізацією.
8. **Готова до автоматичної генерації коду.** Через формальність нотації її потенційно можна використати як проміжний шар компіляторів-генераторів для різних SIMD-архітектур (Розд. 3.1.4).

Б.5.4 Розмежування методу та реалізації (Г)

Що є новим на рівні методу/технології:

- Сама нотація-формалізм як науковий метод (A.12–A.17), табл. A.1.
- Принципи побудови нотації — множинно-теоретичний підхід до іменування регістрів, параметризація типу даних, експліцитна сегментація діапазонів індексів.
- Узагальнені оператори $CShL$ і $maxmin_vdt$ — абстрактні форми, що інкапсулюють родини конкретних інструкцій (A.32, A.40).
- Сукупність базових нотаційних операцій (XOR, ld3/ld1, smlal/smlsl/smlal2/smlsl2 dup, type conversions, st3, index_map clustering) — (A.18, A.22, A.29, A.45).
- Стиль самодокументованого іменування на основі нотації.
- Методологічний внесок: міст між математичним описом і реалізацією, придатний для формалізації, аналізу та верифікації векторизованих алгоритмів (Розд. 3.1.5, Розд. 3.1.5).

Що є реалізаційним наслідком:

- Конкретні асемблерні інструкції ARM64 NEON64, що відповідають елементам нотації: `eor` ($\otimes$), `ld3/ld1`, `smlal/smlsl/smlal2/smlsl2`, `ext` ($CShL$), `sxtl/scvtf/fcvtas/sqxtn/uqxtn` (конвертації), `st3`.
- Конкретні параметризації нотації для 8u $\to$ 16s обчислень — вибір $C \in \{r, g, b\}$, $I \in \{0007, 0815, 1623, 2431\}$, $L \in \{0015, 1631\}$.
- Реалізація обчислювального ядра `conv_RGB_NEON` через Inline Assembly у складі бібліотеки `libconv_lib.so`, що базується на нотації.

- Послідовність операцій (підготовка → ВСД → нормалізація → збереження), записана у блок-схемах рис. А.4–А.6 — це конкретний алгоритм, який *використовує* нотацію, але формалізм нотації існує і поза цим алгоритмом.

Б.5.5 Локалізація доказової бази у тексті дисертації (Д)

Таблиця Б.4 — Локалізація доказової бази для пункту наукової новизни №2

Тип доказу	**Локація**	**Що саме**
Формалізована нотація як універсальний науковий метод (загальний огляд)	Розд. 3.1.1	—
Загальний опис формалізованої нотації (підрозд.)	Розд. 3.1.1	Перелік 6 компонентів нотації
Огляд ШОЗ та її ядра ВСД через нотацію	Розд. 3.1.3	рис. А.4, А.6
Розширені можливості застосування нотації	Розд. 3.1.4	—
Теоретична значущість нотації	Розд. 3.1.5	—
Внесок у методологію досліджень	Розд. 3.1.5	—
Базова формалізація іменування векторних регістрів	Дод. А.3	(А.12)
Функція $elems(dtype)$ та її значення	Дод. А.3	(А.13, А.14)
Множини індексів елементів	Дод. А.3	(А.15, А.16)
Допоміжні функції діапазону i_all, i_1hlf, i_2hlf	Дод. А.3	(А.17); табл. А.1
Операція обнуління через XOR у нотації	Дод. А.4.1	(А.18–А.20)
Завантаження з пам’яті `ld3` у нотації	Дод. А.4.2, А.4.3	(А.22–А.24)
Завантаження ЯЗ `ld1` та маска позитивності	Дод. А.4.4	(А.26–А.28)
Векторне множення-акумуляція smlal/smlsl/smlal2/smlsl2	Дод. А.4.5	(А.29–А.31)
Функція циклічного зсуву $CShL$ як абстракція ext	Дод. А.4.6	(А.32–А.34)
Функція насичення $maxmin_vdt$ як абстракція SQXTN/UQXTN	Дод. А.5.4	(А.40, А.41, А.43)
Конвертації типів у нотації (s16 $\leftrightarrow$ s32 $\leftrightarrow$ f32)	Дод. А.5.1–А.5.3	(А.36–А.39)
Збереження у пам’ять `st3` у нотації	Дод. А.5.5	(А.44)
Кластеризація $index_map$ для розріджених ЯЗ	Дод. А.7.1	(А.45)
Демонстрація: блок-схеми ШОЗ записані через нотацію	Дод. А.2–А.7	рис. А.3–А.6
Реалізація обчислювального ядра на основі нотації	Розд. 3.3.2, 3.4.1	`libconv_lib.so` (Inline Assembly)
Висновки за Дод. А (підкреслює нотацію як новизну)	Дод. А.11	п. 1
Підтвердження у висновках дисертації	Висновки	п. 4

Б.6 Пункт 3 (Уперше). Метод швидкодіючої операції згортки (ШОЗ)

Б.6.1 Що саме отримано (А)

Розроблено та теоретично обґрунтовано метод швидкодіючої ОЗ (ШОЗ) для обробки 8-бітних RGB ЦЗ на платформі ARM64 з використанням 16-бітних цілочисельних обчислень та інструкцій SIMD NEON64. Метод складається з таких компонентів:

1. **Математична формалізація ОЗ для цілочисельних обчислень** — модифікація базового рівняння ОЗ (2.1) до 0-індексованого вигляду з самодокументованими іменами для 8-бітних RGB ЦЗ (А.1–А.4). Розкладання ОЗ на три компоненти: розрахунок скалярного добутку, зовнішні цикли, нормалізація через коефіцієнт flt_div (рис. А.1).
2. **Декомпозиція ОЗ на векторизовану та завершальну частини** — введення кількості векторизованих ОЗ NC_vec та діапазонів $st32N$, src_jrest (А.5–А.7); запис ядра обчислень як (А.8) (векторизована частина) та (А.11) (дообрахунок); візуалізація як рис. А.2, А.3.
3. **Алгоритм обчислення ВСД** як ядра ШОЗ, що складається з шести ключових кроків ((А.18–А.35), блок-схема рис. А.4):
   - обнуління 12 проміжних векторів (по 4 на канал R/G/B) через XOR (А.18–А.20);
   - базове завантаження 32 трійок RGB через `ld3` (А.22, А.23) та додаткове завантаження для довгих ЯЗ (А.24, А.25);
   - завантаження елементів ЯЗ через `ld1` (А.26, А.27) та формування вектора маски «позитивності» $V_NFT0015$ (А.28) — швидке визначення знаку елементів (на 10 % швидше за традиційне порівняння);
   - обчислення ВСД через пари інструкцій `smlal1/smlsl` та `smlal2/smlsl` (А.29–А.31) — 12 векторних операцій множення-акумуляції на один елемент ЯЗ;
   - **ключова оптимізація — циклічний зсув** векторних регістрів через $CShL$ (А.32–А.34) замість повторного завантаження даних ЦЗ, що усуває операції `load` у внутрішньому циклі;
   - фінальний дообрахунок остатку ЯЗ (А.35).

4. **Алгоритм нормалізації ВСД** (А.36–А.44) з повним ланцюгом конвертації типів:
   - s16 → s32 ((А.36), через `sxtl/sxtl2`);
   - s32 → f32 ((А.37), через `scvtf`);
   - множення на flt_div (А.38) через `fmul`);
   - f32 → s32 ((А.39), через `fcvtas`);
   - s32 → s16 → u8 із насиченням (А.40–А.43);
   - збереження результату через `st3` (А.44).
5. **Повна блок-схема ШОЗ** — інтеграція векторизованих обчислень, дообрахунку, нормалізації та збереження для всього ЦЗ (рис. А.5).
6. **Оптимізація для розріджених ЯЗ** через ПМ *index_map*, що групує елементи ЯЗ за знаком і ненульовим значенням (підрозд. А.7, (А.45)). Розширює ВСД двома гілками обчислень — для позитивних (А.46, А.48) та негативних (А.47, А.49) кластерів елементів. Допоміжні параметри flt_pr16N та flt_pr16N_rest (А.50, А.51) для випадків, коли довжина кластера перевищує 16 елементів. Алгоритм візуалізовано на рис. А.6.
7. **Чотири спеціалізовані варіанти ядра** `conv_RGB_NEON` з автоматичним вибором за параметрами ЯЗ (підрозд. 3.3.2):
   - `conv_RGB_NEON_U8_matrix` — стандартна для беззнакових 8-бітних ЯЗ, найшвидша;
   - `conv_RGB_NEON_S8_matrix_jumps` — для знакових 8-бітних ЯЗ з механізмом «стрибків» через CBNZ;
   - `conv_RGB_NEON_spars_matrix` — для розріджених ЯЗ ($>25\,\%$ нульових елементів);
   - `conv_RGB_NEON_scale_down` — для зменшення розміру ЦЗ з одночасною ОЗ.
8. **Програмна реалізація** у вигляді динамічної бібліотеки `libconv_lib.so` мовою C з Inline Assembly NEON64 (підрозд. 3.4.1, 3.4.2); інтерфейс ядра `conv_RGB_NEON(...)` наведено у табл. 3.3.
9. **Експериментально підтверджене прискорення** порівняно з `cv::filter2D` бібліотеки OpenCV на платформах ARM64:
   - для ЯЗ середнього розміру (8×8 – 11×11) — **3,1–11,1× (Orange Pi 5 Pro) та 2,7–10,2× (Raspberry Pi 4)**;
   - для великих ЯЗ (12×12 – 15×15) — **1,8–5,8× (Orange Pi 5 Pro) та**

1,7–5,2× (Raspberry Pi 4);

- для малих ЯЗ (2×2 – 7×7) — 0,8–2,5× (Orange Pi 5 Pro) та 0,9–2,4× (Raspberry Pi 4); за прямим вимірюванням без вводу-виводу — 1,3–2,8×;
- для операції 2× масштабування (4 ОЗ підряд) — до 6,3× (Orange Pi 5 Pro) та 6,2× (Raspberry Pi 4);
- у режимі обробки відеопотоку — стабільні 26–29 FPS на Orange Pi 5 Pro при роздільності HD (підрозд. 4.6).

10. **Перспективні розширення методу** (підрозділи А.8, А.9, А.10): використання конструкції switch-continue для оптимізації знакових ЯЗ; конвертування знакових ЯЗ у беззнакові з компенсаційною нормалізацією; кеш-френдлі обробка ЦЗ (А.58, А.59).

Б.6.2 Аналогічні рішення та відомі підходи (Б)

У літературі та програмних продуктах, що адресують пов'язану задачу прискорення ОЗ ЦЗ, можна виділити такі групи аналогів:

- **`cv::filter2D(...)` з OpenCV** [11–13, 36] — найпоширеніша універсальна реалізація ОЗ. Аналізується детально у Розд. 1.4: внутрішній стек 8 шарів виклику, конвеєр перетворень типів 8u → 16s → 32s → 32f, перехід на FFT-алгоритм для ЯЗ 8×8 і більше (умова в `filter.dispatch.cpp:1275`), що призводить до різкого падіння швидкодії на цьому розмірі (табл. 4.3).
- **ARM Compute Library (ACL) `NEConvolution{N}x{N}::run(...)`** [14] — спеціалізовані SIMD-реалізації ОЗ для ARM. Після коміту `7dcb9fadb9` ці класи видалено — бібліотека сфокусована на CNN-постобробці. Тільки фіксовані розміри ЯЗ та переважно 32-біт.
- **Автоматична векторизація компіляторів** [52–57] — GCC -O3, Clang -O3 / -Ofast. Ефективна для простих циклів, але недостатня для складних випадків з ЯЗ складної структури (підрозд. 1.3.6: експериментально доведено, що автовекторизація на ARM64 для розгорнутих циклів дає -3–5 % швидкодії порівняно з ручним кодом).
- **CUDA / OpenCL реалізації згортки на GPU** [58–62] — забезпечують прискорення до 48× для великих ЦЗ, але вимагають GPU, страждають від накладних витрат передачі даних CPU ↔ GPU, не підходять для одноплатних SBC без потужних GPU.
- **DSP Hexagon від Qualcomm** [24, 25, 63] — VLIW DSP-співпроцесор, до

4–18× прискорення кодування/декодування медіа. Доступний тільки на SoC Snapdragon, страждає від накладних витрат переривань.

- **Класичні алгоритми згортки**: im2col [64], Winograd [48], FFT-конволюція [36] – асимптотично кращі для дуже великих ЯЗ, але на практиці для ЯЗ середнього розміру (3×3 – 11×11) поступаються прямій ОЗ через накладні витрати трансформацій.
- **Ігровий двигун id Tech 3 та споріднені** (підрозд. 1.3.5) — піонерські SSE-оптимізації для 3D-графіки, але не вирішують специфічну задачу ОЗ ЦЗ.
- **Класичні маски та підходи Приставки П.О.** [4, 65, 66] — теоретичне підґрунтя побудови масок (B-сплайни), але на цілочисельних 16-біт обчисленнях NEON64 не реалізовано.
- **Підвищення швидкодії множення матриць та згорток через кеш-оптимізаці** 70] — релевантні принципи, що частково використано (підрозд. А.10).
- **Власні попередні роботи автора** [71–74] — попередні стадії розробки методу, що увійшли у поточний результат.

Б.6.3 Відмінність від відомих підходів (В)

Метод ШОЗ відрізняється від наявних аналогів за такими ключовими аспектами:

1. **16-бітні цілочисельні обчислення замість 32-біт float.** На відміну від `cv::filter2D(…)` (8u → 16s → 32s → 32f), запропонований метод використовує укорочений ланцюг 8u → 16s для розрахунку ВСД (А.29), що: (а) зменшує кількість конвертацій типів; (б) дозволяє утричі-увосьмеро більше елементів за один ВР; (в) обходить інструкції множення з плаваючою комою FMUL у внутрішньому циклі. Конвертація у f32 виконується тільки на етапі нормалізації (А.38).
2. **Циклічний зсув ВР замість повторного завантаження.** У `cv::filter2D(…` та аналогічних кросплатформних бібліотеках для кожного нового елементу ЯЗ виконується повторне завантаження частини ЦЗ. У запропонованому методі вектори V_C0015, V_C1631, V_C3247 (А.23, А.24) циклічно зсуваються «самі з собою» через $CShL$ (А.32, А.33) — це усуває операції `ld3` у внутрішньому циклі обчислення ВСД і є **суттєвою оптимізацією як алгоритмічною, так і реалізаційною**.

3. **Прямий метод без перемикання на FFT.** Метод працює прямою згорткою для всього діапазону розмірів ЯЗ (2×2 – 15×15), на відміну від `cv::filter2D(...)`, що для ЯЗ 8×8 і більше переходить на FFT-алгоритм через умову у файлі `filter.dispatch.cpp:1275`, що призводить до падіння швидкодії у $\sim$5,8 раза (Orange Pi 5 Pro) та $\sim$5,4 раза (Raspberry Pi 4) (експериментально підтверджено: табл. 4.3). Запропонований метод не має такого розриву (табл. 4.4).
4. **Беззнакова арифметика як проектний вибір.** Використання беззнакових 16-бітних обчислень, де можливо (`conv_RGB_NEON_U8_matrix`) уникнення додаткової логіки АЛП для обробки знаку, зниження кількості інструкцій на 50 % порівняно зі знаковим варіантом (підрозд. 2.3.5, (2.15)).
5. **Кластеризація розрідженого ЯЗ через ПМ.** Інтегрована оптимізація для ЯЗ з $>$25 % нульових елементів через *index_map* (А.45) рис. А.6) дозволяє пропускати множення на нуль і групувати ненульові елементи за знаком — цього аналога немає у `cv::filter2D(...)` та ACL.
6. **Інтеграція з ефективнішими інструкціями NEON64.** Метод явно використовує сімейство інструкцій `smlal/smlsl/smlal2/smlsl2` (А.29) — множення-акумуляцію з одночасним розширенням розрядності $8 \rightarrow 16$ біт за одну машинну інструкцію. У OpenCV Universal Intrinsics ця ефективність розмивається через шар абстракції.
7. **Ручна Inline Assembly замість intrinsics або автовекторизації.** Використання Inline Assembly NEON64 (підрозд. 4.4.3) дозволяє повний контроль над виділенням ВР, послідовністю команд і використанням конвеєра ЦП. Експериментально доведено, що це дає перевагу до 11,1$\times$ (Orange Pi 5 Pro) та до 10,2$\times$ (Raspberry Pi 4) проти `cv::filter2D(...)` — такий розрив недосяжний для автовекторизації.
8. **Адаптивне диспетчерування за параметрами ЯЗ.** Функція `conv_RGB_NEON` аналізує параметри ЯЗ (розмір, наявність нульових елементів, знаковість) та автоматично обирає найефективніший варіант реалізації (підрозд. 3.4.2). У `cv::filter2D(...)` такого диспетчерування за характеристиками ЯЗ немає (тільки за розміром, з переходом на FFT).
9. **Платформна незалежність у межах ARMv8.** Метод однаково ефективно працює як на ARMv8.2-A (Orange Pi 5 Pro з Cortex-A76), так і на ARMv8-A (Raspberry Pi 4 з Cortex-A72) — експериментально підтверджено схожі

патерни прискорення (табл. 4.4).

10. **Збережена точність обчислень.** Незважаючи на відмову від обчислень з плаваючою комою у ядрі, максимальна абсолютна похибка не перевищує 1 одиницю на канал 8-бітного RGB (підтверджено у [71–73]), що візуально не впливає на якість ЦЗ.

Б.6.4 Розмежування методу та реалізації (Г)

Що є новим на рівні методу/технології:

- Парадигма 16-бітних цілочисельних обчислень для ШОЗ 8-бітних RGB ЦЗ як проектне рішення.
- Математична формалізація ШОЗ — (А.1–А.4).
- Декомпозиція ОЗ на векторизовану та завершальну частини — (А.5–А.11).
- Алгоритм обчислення ВСД — (А.18–А.35), рис. А.4.
- **Принцип циклічного зсуву ВР замість повторного завантаження** як ключова оптимізація — (А.32–А.34).
- Алгоритм нормалізації ВСД з конвертацією типів — (А.36–А.44).
- Принцип кластеризації розрідженого ЯЗ через ПМ *index_map* — формула (А.45), рис. А.6.
- Повна блок-схема методу — рис. А.5.
- Підхід адаптивного диспетчерування за характеристиками ЯЗ (розмір, знаковість, розрідженість).

Що є реалізаційним наслідком:

- Бібліотека `libconv_lib.so` як С-реалізація методу.
- Конкретні Inline Assembly послідовності NEON64 для кожного етапу алгоритму (`eor`, `ld3`, `ld1`, `smlal`/`smlsl`/`smlal2`/`smlsl2`, `ext`, `sxtl`/`sxtl2`, `scvtf`, `fmul`, `fcvtas`, `sqxtn`/`sqxtn2`, `uqxtn`/`uqxtn2`, `st3`).
- Чотири спеціалізовані варіанти ядра (`conv_RGB_NEON_U8_matrix`, `conv_R` `conv_RGB_NEON_spars_matrix`, `conv_RGB_NEON_scale_down`) як інстанції методу для конкретних типів ЯЗ.
- Допоміжні функції `process_matrix_once` (формування *index_map*), `img_conv_RGB__flt_s8__contain0s`, `img_conv_RGB__flt_u8__co`
- Сигнатура та інтерфейс `conv_RGB_NEON(...)` (табл. 3.3).
- Python-обгортка через `ctypes` з підтримкою NumPy (підрозд. 3.6.1, 3.6.2).

Б.6.5 Локалізація доказової бази у тексті дисертації (Д)

Таблиця Б.5 — Локалізація доказової бази для пункту наукової новизни №3 (частина 1)

Тип доказу	**Локація**	**Що саме**
Математична формалізація ОЗ для цілочисельних обчислень	Дод. А.1–А.7	(А.1–А.4); рис. А.1
Декомпозиція на векторизовану + завершальну частини	Дод. А.2	(А.5–А.11); рис. А.2, А.3
Підготовка проміжних ВР (обнуління)	Дод. А.4.1	(А.18–А.20)
Завантаження RGB ЦЗ (`ld3`)	Дод. А.4.2, А.4.3	(А.21–А.25)
Завантаження ЯЗ (`ld1`) + маска позитивності	Дод. А.4.4	(А.26–А.28)
Ядро ВСД: smlal/smlsl/smlal2/smlsl2	Дод. А.4.5	(А.29–А.31)
Циклічний зсув ВР (ключова оптимізація)	Дод. А.4.6	(А.32–А.34)
Завершення обчислення ВСД	Дод. А.4.7	(А.35); рис. А.4
Конвертація s16 $\rightarrow$ s32 $\rightarrow$ f32	Дод. А.5.1	(А.36, А.37)
Нормалізація ВСД (множення на flt_div)	Дод. А.5.2	(А.38)
Конвертація f32 $\rightarrow$ s32	Дод. А.5.3	(А.39)
Конвертація s32 $\rightarrow$ u8 із насиченням	Дод. А.5.4	(А.40–А.43)
Збереження результату у вихідне ЦЗ (`st3`)	Дод. А.5.5	(А.44)
Повна блок-схема методу	Дод. А.6	рис. А.5
Кластеризація ЯЗ через *index_map*	Дод. А.7.1	(А.45)
ВСД з ПМ (позитивні / негативні гілки)	Дод. А.7.2	(А.46–А.49)
Обробка довгих кластерів ($>$16 елементів)	Дод. А.7.3	(А.50, А.51); рис. А.6
Перспективні розширення: switch-continue	Дод. А.8	—
Перспективні розширення: знакові $\rightarrow$ беззнакові	Дод. А.9	(А.52–А.57)
Перспективні розширення: кеш-френдлі підхід	Дод. А.10	(А.58, А.59)
Висновки за Дод. А	Дод. А.11	п.п. 1–7
Концепція ІТ та принципи (інтеграція методу в систему)	Розд. 3.2	—
Архітектура ІТ	Розд. 3.3	рис. 3.2
Програмна реалізація ядра ІТ	Розд. 3.4	`libconv_lib.so`
Інтерфейс функції `conv_RGB_NEON`	Розд. 3.4.2	табл. 3.3

Таблиця Б.6 — Локалізація доказової бази для пункту наукової новизни №3 (частина 2)

Тип доказу	**Локація**	**Що саме**
Спеціалізовані варіанти ядра `conv_RGB_NEON_U8_matrix`, `..._S8_matrix_jumps`, `..._spars_matrix`, `..._scale_down`	Розд. 3.3.2	—
Особливості реалізації (16-біт ЦО, NEON64, кеш, кластеризація)	Розд. 3.4.3	—
Методика застосування ІТ (різні типи ЦЗ)	Розд. 3.5.1	табл. 3.4
Експериментальні показники C++ (FPS, прискорення)	Розд. 4.6.1	табл. 4.3, 4.4 (до 11,1×)
Експериментальні показники Python (FPS, прискорення)	Розд. 4.6.2	табл. 4.6, 4.7
Експериментальні показники масштабування (zoom 2×)	Розд. 4.6.3	табл. 4.8, 4.9 (до 6,3× (OPi5) та 6,2× (RPi4))
Висновки розд. 3	Розд. 3.8	п.п. 1–7
Висновки розд. 4	Розд. 4.7	п.п. 2–6
Підтвердження у висновках дисертації	Висновки	п. 5 (метод ШОЗ)

Б.7 Пункт 5 (теоретичне підґрунтя п. 3 категорії «Уперше»). Математична модель операції згортки для 16-бітних цілочисельних обчислень

Б.7.1 Що саме отримано (А)

Розроблено математичну модель ОЗ, орієнтовану на використання 16-бітних цілочисельних обчислень для обробки 8-бітних RGB ЦЗ при 8-бітних ЯЗ. Ця модель є теоретичним фундаментом для методу ШОЗ (Пункт Б.6) та для конкретних лінійних операторів (Пункт Б.3). Модель включає такі компоненти:

1. **Узагальнене рівняння ОЗ** у класичній формі (2.1) з визначенням оператора згортки $F(p^{i,j})$, маски фільтра $\gamma_{ii-i,jj-j}$ та анкорів $(2r_i + 1) \times (2r_j + 1)$.
2. **Самодокументована 0-індексована форма для цілочисельних обчислень** (А.1–А.4) — модифікація рівняння до вигляду, придатного для практичної реалізації, з самодокументованими іменами індексів (src_i, src_j, dst_i, dst_j, flt_i, flt_j) та з винесенням коефіцієнта нормалізації flt_div

за межі суми.

3. **Умова коректності 16-бітних обчислень** — ключовий теоретичний результат у формі обмеження на ЯЗ:

$$\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} |\gamma_{ii-i,jj-j}| \leq 127$$

(2.2, 2.3). Виведено з умови, що $p_{ii,jj} \in [0; 255]$ і проміжний результат не повинен виходити за межі діапазону знакових 16-бітних цілих $[-32768; 32767]$. Це обмеження є визначальним для проектування ЯЗ під 16-бітну арифметику.

4. **Моделювання діапазонів типів даних** (2.15) — формальний опис множини значень, які можуть приймати знакові та беззнакові 8-/16-/32-бітні цілі типи: $max(u8) =$ 0xFF, $max(s8) =$ 0x7F, $min(s8) = -$0x80 та аналогічно для u16/s16/u32/s32. Цей апарат є основою для аналізу того, які операції SIMD NEON64 застосовні (UMLAL/UMLSL для беззнакових, SMLAL/SMLSL для знакових) та для порівняння очікуваних діапазонів проміжних результатів.
5. **Методика адаптації знакових ЯЗ до беззнакових 16-бітних обчислень з компенсаційною нормалізацією** (2.16–2.19):
   - умова коректності для беззнакового випадку (2.16): $\sum \gamma_{ii-i,jj-j} \leq 255$;
   - визначення зсуву $\gamma_{bias} = \gamma_{\min} + 1$ як найбільшого за модулем від'ємного елемента ЯЗ плюс одиниця (2.17);
   - перетворення ЯЗ $\gamma^{new}_{i,j} = \gamma_{i,j} + \gamma_{bias} > 0$ — усі елементи стають додатними (2.18);
   - компенсація через бокс-фільтр (2.19):

$$F(p^{i,j}) = F^{new}(p^{i,j}) - \gamma_{bias} \sum p_{ii,jj}$$

   Це дозволяє виконувати основні обчислення у беззнаковому 16-бітному діапазоні з використанням швидших операцій `umlal/umlal2`, при цьому обробляючи удвічі більше обсягу за 32-бітні обчислення.
6. **Підхід до розкладання ЯЗ з елементами понад 8-бітний діапазон** на

чотири байтові компоненти (2.24–2.28):

$$\gamma_{i,j} = \gamma_{i,j}^{(0)} + \gamma_{i,j}^{(1)} \cdot 2^8 + \gamma_{i,j}^{(2)} \cdot 2^{16} + \gamma_{i,j}^{(3)} \cdot 2^{24}$$

та переформулювання ОЗ через зсуви ($\ll 8$, $\ll 16$, $\ll 24$). Це дозволяє зберегти переваги 16-бітних проміжних обчислень навіть для широкосмугових ЯЗ (32-біт).

7. **Типізація вхідних та вихідних ЦЗ** — модель параметризована глибиною 8 біт на канал ($p_{ii,jj} \in [0; 255]$), що відображає типову RGB-обробку.
8. **Розширення на матричний рівень для оптимізації** — модель допускає введення допоміжної ПМ *index_map* (детально розкривається у Пункті Б.6, (А.45)), що дозволяє оптимізувати обробку розріджених ЯЗ.

Таким чином, модель є замкнутою теоретичною системою, що (а) описує структуру обчислень, (б) формулює умови коректності для 16-біт ЦО, (в) надає методики обходу обмежень (зсув γ_{bias}, декомпозиція). Усі подальші результати дисертації (метод ШОЗ, конкретні маски, експерименти) є інстанціями або наслідками цієї моделі.

Б.7.2 Аналогічні рішення та відомі підходи (Б)

У літературі та практичних реалізаціях існують такі групи аналогів:

- **Класична математична модель ОЗ з обчисленнями з плаваючою комою** [1–3] — описує ОЗ у неперервному просторі або з float-точністю. Не враховує специфіку фіксованих типів даних, не формулює умов коректності для цілочисельних обчислень.
- ***B*-сплайнові підходи Приставки П.О.** [4–10, 65] — розглядають побудову лінійних операторів через лінійні комбінації *B*-сплайнів, переважно у форматі float; теоретично обґрунтовують вибір масок.
- **Загальні принципи fixed-point арифметики** [15–18, 75] — описують перетворення між float і fixed-point, аналіз похибки округлення, але без прив'язки до ОЗ ЦЗ або до 16-біт цілочисельного формату конкретно для 8-біт RGB.
- **Стандарт IEEE 754 для float-обчислень** [76] — альтернативний (уникнений у запропонованій моделі) спосіб представлення проміжних результатів.

- **Реалізація `cv::filter2D(...)` з конвеєром перетворень типів** 8u → 16s → 3
  13] — неявно містить математичну модель, але не формалізує її як окремий об'єкт; спирається на 32-біт float як «безпечний» проміжний тип.
- **Літературні джерела щодо ефективних згорткових алгоритмів** [47–50] — описують алгоритмічні оптимізації (Winograd, im2col, FFT), але працюють зазвичай у float; не пропонують моделей під 16-біт ЦО.
- **Власні попередні роботи автора** [71–74] — ранні стадії формування поточної моделі.

Б.7.3 Відмінність від відомих підходів (В)

Запропонована математична модель відрізняється від наявних аналогів за такими ключовими аспектами:

1. **Експліцитна спеціалізація під 16-бітні цілочисельні обчислення.** На відміну від класичних float-моделей [1–3] та загальних fixed-point принципів [15, 16], запропонована модель явно фіксує тип проміжних обчислень (u16 | s16) та виходить з обмежень цього типу при формулюванні умов коректності.
2. **Параметризація під 8-бітну глибину RGB.** Модель не є загальною — вона спеціалізована під типову задачу обробки 8-бітних кольорових ЦЗ, що дозволяє вивести конкретні чисельні обмеження ($\sum |\gamma| \leq 127$ або $\sum \gamma \leq 255$) замість абстрактних формулювань.
3. **Оригінальна умова коректності** (2.2, 2.3). У згаданих аналогах така умова або не формулюється взагалі (класичні моделі [1]), або сформульована у вигляді абстрактних меж для fixed-point [15], без зв'язку з конкретною задачею ОЦЗ.
4. **Інтегроване моделювання діапазонів типів** (2.15) — зведена таблиця min / max для u/s 8/16/32-бітних типів безпосередньо у тілі моделі, що дозволяє вивести правила застосовності SIMD-інструкцій (UMLAL vs SMLAL, UQXTN vs SQXTN тощо).
5. **Оригінальна методика адаптації знакових ЯЗ до беззнакових ЦО з компенсаційною нормалізацією** (2.17–2.19) — зсув на γ_{bias} + бокс-фільтр для відновлення оригінального результату. Цього аналога немає у жодній зі згаданих груп.
6. **Методика декомпозиції 32-бітних ЯЗ на байтові компоненти** (2.24–2.28) —

дозволяє зберегти переваги 16-біт ЦО навіть для широких ЯЗ; також не зустрічається у наявних моделях.

7. **Зв'язок з SIMD як архітектурним фактором.** Модель формулюється з огляду на майбутню реалізацію через SIMD NEON64 (тобто 16-біт обрано як оптимум для 128-бітних векторних регістрів) — цей архітектурний фактор зазвичай відсутній у класичних математичних моделях ОЗ.
8. **Відкритість до інтеграції з оптимізаціями** (зокрема, з ПМ *index_map*, (А.45), що використовується у Пункті Б.6) — модель допускає алгоритмічні оптимізації не як «розширення», а як органічну частину її структури.

Б.7.4 Розмежування методу та реалізації (Г)

Що є новим на рівні методу/моделі:

- Сама математична модель ОЗ під 16-біт ЦО як цілісний теоретичний результат.
- Узагальнене рівняння ОЗ у самодокументованій 0-індексованій формі — (А.1–А.4).
- Умова коректності 16-біт ЦО — (2.2, 2.3).
- Інтегроване моделювання діапазонів типів даних — (2.15).
- Методика адаптації знакових ЯЗ до беззнакових ЦО з компенсаційною нормалізацією — (2.16–2.19).
- Принцип декомпозиції 32-бітних ЯЗ на байтові компоненти — (2.24–2.28).
- Параметризація під 8-біт RGB ЦЗ як проектне обмеження.

Що є реалізаційним наслідком (інстанціями моделі):

- Конкретні чисельні маски ЯЗ — це інстанції, що задовольняють умові коректності з (2.3) (див. П. Б.3).
- Конкретні беззнакові маски ЯЗ з компенсаційною нормалізацією (2.20–2.23) — це інстанції методики (2.17–2.19).
- Алгоритм ШОЗ (Пункт Б.6) та його програмна реалізація `libconv_lib.so` — це конкретний обчислювальний апарат для практичної реалізації моделі.
- Спеціалізовані варіанти ядра (`conv_RGB_NEON_U8_matrix`, `conv_RGB_NE` `conv_RGB_NEON_spars_matrix`) реалізують різні гілки моделі залежно від характеру ЯЗ.

Б.7.5 Локалізація доказової бази у тексті дисертації (Д)

Таблиця Б.7 — Локалізація доказової бази для результату №5 (теоретичне підґрунтя п. 3 категорії «Уперше»)

Тип доказу	**Локація**	**Що саме**
Узагальнене рівняння ОЗ (класична форма)	Розд. 2.1	(2.1)
Умова коректності 16-біт ЦО (виведення)	Розд. 2.1	(2.2, 2.3)
Самодокументована 0-індексована форма для ЦО	Дод. А.1	(А.1–А.4); рис. А.1
Моделювання діапазонів типів даних (u/s 8/16/32)	Розд. 2.3	(2.15)
Аналіз впливу знаковості на швидкодію (теоретичне обґрунтування)	Розд. 2.3	—
Умова коректності для беззнакових ЦО	Розд. 2.4	(2.16)
Визначення γ_{bias} та зсуву ЯЗ	Розд. 2.4	(2.17, 2.18)
Компенсаційна нормалізація через бокс-фільтр	Розд. 2.4	(2.19)
Декомпозиція 32-бітних ЯЗ на байтові компоненти	Розд. 2.5	(2.24–2.28)
Висновки розд. 2 (підтвердження п.п. 1, 4, 5)	Розд. 2.6	п.п. 1, 4, 5
Зв’язок моделі з методом ШОЗ (Пункт Б.6)	—	див. табл. Б.5
Інстанції моделі — конкретні маски ЯЗ (Пункт Б.3)	—	див. табл. Б.2
Підтвердження у висновках дисертації	Висновки	п. 3 (математична модель ОЗ)

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Gonzalez R. C.*, *Woods R. E.* Digital image processing (3rd edition). USA : Prentice-Hall, Inc., 2006.
2. *Castleman K. R.* Digital image processing. 2nd ed. Englewood Cliffs, N.J. : Prentice Hall, 1996. 667 p.
3. *Pratt W. K.* Introduction to digital image processing. Boca Raton : CRC Press, 2013. 750 p.
4. *Приставка П.* Побудова контрастних фільтрів за використанням поліноміальних сплайнів. Актуальні проблеми автоматизації та інформаційних технологій. 2007. Т. 11. С. 15–22.

5. *Приставка П.* Лінійні комбінації в-сплайнів, близькі до інтерполяційних у середньому, в задачі моделювання аналогових сигналів. Актуальні проблеми автоматизації та інформаційних технологій. 2011. Т. 15. С. 4–17.

6. *Приставка П.*, *Чолишкіна О.* Піраміда зображень на основі сплайн-моделі у вигляді лінійної комбінації в-сплайнів. Інформаційні технології та суспільство. 2021. 1 (1). С. 34–42. URL: https://doi.org/10.32689/maup.it.2021.1.4 (дата звернення: 04.09.2025).

7. *Приставка П.* Застосування комбінованих фільтрів на основі поліноміальних сплайнів при обробці растрових зображень. Вісник нау. 2008. 4. С. 104–107.

8. *Приставка П.* Поповнення зі згладжуванням послідовностей відліків функцій двох змінних на основі сплайнів. Математичне моделювання. 2008. 1(18). С. 9–12.

9. *Приставка П.*, *Чолишкіна О.* Дослідження комбінованих фільтрів для підвищення різкості зображень. Актуальні проблеми автоматизації та інформаційних технологій. 2009. Т. 13. С. 39–53.

10. *Приставка П. О.* Поліноміальні сплайни в задачах обробки цифрових зображень. Дніпропетровськ : ДНУ, 2012. 264 с.

11. Opencv: image filtering. OpenCV team. URL: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04 (date of access: 05.08.2025).

12. Opencv - open computer vision library. OpenCV team. URL: https://opencv.org/ (date of access: 05.08.2025).

13. Opencv: vectorizing your code using universal intrinsics. OpenCV team. URL: https://docs.opencv.org/4.x/d6/dd1/tutorial_univ_intrin.html (date of access: 05.08.2025).

14. *Barbier A.* ARM-software/computelibrary: the compute library is a set of computer vision and machine learning functions optimised for both arm cpus and gpus using SIMD technologies. Arm-software. URL: https://github.com/ARM-software/ComputeLibrary (date of access: 30.07.2025).

15. *Oberstar E. L.* Fixed-point representation & fractional math. Oberstar Consulting. 2007. URL: http://darcy.rsgc.on.ca/ACES/ICE4M/FixedPo

int/FixedPointRepresentationFractionalMath.pdf (date of access: 13.09.2025).

16. *Kormanyos C.* Real-time C++ : Efficient object-oriented and template microcontroller programming. 3rd ed. Berlin, Heidelberg : Springer, 2018. 426 p. URL: https://doi.org/10.1007/978-3-662-56718-0.
17. *Padgett W. T.*, *Anderson D. V.* Fixed-point signal processing. Vol. 9. Morgan & Claypool Publishers, 2009.
18. Low-precision floating-point formats: from general-purpose to application-specific. A. Sabbagh Molahosseini, L. Sousa, A. A. Emrani Zarandi, H. Vandierendonck. Approximate computing. 2012. P. 77–98.
19. Valgrind. callgrind: a call-graph generating cache and branch prediction profiler. Valgrind Developers. URL: https://valgrind.org/docs/manual/cl-manual.html (date of access: 05.08.2025).
20. Valgrind. massif: a heap profiler. Valgrind Developers. URL: https://valgrind.org/docs/manual/ms-manual.html (date of access: 05.08.2025).
21. Mlperf power: benchmarking the energy efficiency of machine learning systems from μwatts to mwatts for sustainable AI. A. Tschand, A. T. R. Rajan, S. Idgunji, A. Ghosh, J. Holleman, C. Kiraly, P. Ambalkar, R. Borkar, R. Chukka, T. Cockrell, [et al.]. 2025 IEEE international symposium on high performance computer architecture (HPCA). 2025. P. 1201–1216.
22. Benchmarking energy and latency in tinyml: a novel method for resource-constrained ai. P. Bartoli, C. Veronesi, A. Giudici, D. Siorpaes, D. Trojaniello, F. Zappa. 2025 international joint conference on neural networks (IJCNN). 2025. P. 1–8.
23. Benchmarking object detection deep learning models in embedded devices. D. Cantero, I. Esnaola-Gonzalez, J. Miguel-Alonso, E. Jauregi. Sensors. 2022. Vol. 22: 11: 11. P. 4205. URL: https://doi.org/10.3390/s22114205 (date of access: 05.08.2025).
24. The qualcomm® hexagon™ SDK. Qualcomm. URL: https://docs.qualcomm.com/bundle/publicresource/topics/80-77512-1/hexagon-dsp-sdk-collection-landing-page.html?product=1601111740010422 (date of access: 05.08.2025).

25. Hexagonal convolutional neural network for spatial transcriptomics classification. J. Gao, F. Zhang, K. Hu, X. Cui. 2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). 2022. P. 200–205. URL: https://doi.org/10.1109/BIBM55620.2022.9995701 (date of access: 12.09.2025).

26. *Bryant R. E.*, *O'Hallaron D. R.* Computer systems: a programmer's perspective. 3rd ed. Boston : Pearson Education, 2015. 1120 p.

27. *Fog A.* Optimizing software in C++: an optimization guide for windows, linux and mac platforms. Copenhagen University College of Engineering, 2025. 179 p. URL: https://www.agner.org/optimize/optimizing_cpp.pdf (date of access: 29.07.2025).

28. *Chellappa S.*, *Franchetti F.*, *Püschel M.* How to write fast numerical code: a small introduction. Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. Berlin, Heidelberg : Springer, 2008. P. 196–259. URL: https://doi.org/10.1007/978-3-540-88643-3_5 (date of access: 05.08.2025).

29. *Cardoso J. M. P.*, *Coutinho J. G. d. F.*, *Diniz P. C.* Embedded computing for high performance: efficient mapping of computations using customization, code transformations and compilation. Elsevier Science & Technology Books, 2017. 322 p.

30. *Muresano R.*, *Rexachs D.*, *Luque E.* Learning parallel programming: a challenge for university students. Procedia computer science. 2010. Vol. 1: 1. P. 875–883. (ICCS 2010). URL: https://doi.org/10.1016/j.procs.2010.04.096 (date of access: 13.09.2025).

31. Arm architecture reference manual for a-profile architecture. Arm Developer. URL: https://developer.arm.com/documentation/ddi0487/latest (date of access: 29.07.2025).

32. Arm a-profile a64 instruction set architecture. a64 – SIMD and floating-point instructions (alphabetic order). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions (date of access: 29.07.2025).

33. Arm a-profile a64 instruction set architecture. EOR (bitwise exclusive-OR (vector)). Arm Developer. URL: https://developer.arm.com/documentation/ddi0602/2025-06/SIMD-FP-Instructions/EOR--vector---Bitwise-exclusive-OR--vector--?lang=en (date of access: 13.09.2025).

34. Arm C language extensions. Arm Developer. URL: https://arm-software.github.io/acle/main/acle.html (date of access: 30.07.2025).

35. Arm neon intrinsics reference. Arm Developer. URL: https://arm-software.github.io/acle/neon_intrinsics/advsimd.html (date of access: 30.07.2025).

36. Opencv: vectorizing your code using universal intrinsics. 2-d convolution. OpenCV team. URL: https://docs.opencv.org/4.x/d6/dd1/tutorial_univ_intrin.html#autotoc_md367 (date of access: 13.09.2025).

37. *Pyeatt L. D.*, *Ughetta W.* ARM 64-bit assembly language. Cambridge : Newnes, 2019. 508 p.

38. *Pyeatt L. D.* Modern assembly language programming with the ARM processor. Amsterdam : Elsevier, 2024. 554 p.

39. *Hoey J. V.* Beginning x64 assembly programming: from novice to AVX professional. Berkeley : Apress, 2019. 413 p.

40. *Brooks D. R.* C programming: the essentials for engineers and scientists. Springer Science & Business Media, 2012. 490 p.

41. *Thomas M.* Programmer's guide for ARMv8-a. Arm Community. 2015. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/programmer-s-guide-for-armv8-a (date of access: 13.09.2025).

42. *Flynn M.* Very high-speed computing systems. Proceedings of the IEEE. 1966. Vol. 54: 12. P. 1901–1909. URL: https://doi.org/10.1109/PROC.1966.5273 (date of access: 29.07.2025).

43. *Flynn M. J.* Some computer organizations and their effectiveness. IEEE transactions on computers. 1972. Vol. C–21: 9. P. 948–960. URL: https://doi.org/10.1109/TC.1972.5009071 (date of access: 29.07.2025).

44. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and intel platforms. G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, J. Zhou. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. 2013. P. 1107–1116. URL: https://ieeexplore.ieee.org/abstract/document/6650996/ (date of access: 29.07.2025).

45. *Larsen S.*, *Amarasinghe S.* Exploiting superword level parallelism with multimedia instruction sets. Sigplan not. 2000. Vol. 35: 5. P. 145–156. URL: https://doi.org/10.1145/358438.349320 (date of access: 05.08.2025).

46. NEON is the new black: fast JPEG optimization on ARM server. The Cloudflare Blog. 2018. URL: https://blog.cloudflare.com/neon-is-the-new-black/ (date of access: 05.08.2025).

47. *Liu Z.-G.*, *Mattina M.* Efficient residue number system based winograd convolution. European conference on computer vision. 2020. P. 53–68.

48. Winograd convolution for deep neural networks: efficient point selection. S. A. Alam, A. Anderson, B. Barabasz, D. Gregg. Acm transactions on embedded computing systems. 2022. Vol. 21: 6. P. 1–28.

49. Performance comparison of gcc and llvm on the eisc processor. C. Park, M. Han, H. Lee, S. W. Kim. 2014 international conference on electronics, information and communications (ICEIC). 2014. P. 1–2.

50. Comparison of LLVM and GCC on the ARM platform. J.-J. Kim, S.-Y. Lee, S.-M. Moon, S. Kim. 2010 5th international conference on embedded and multimedia computing. 2010. P. 1–6.

51. *Martin R. C.* Clean code: a handbook of agile software craftsmanship. Upper Saddle River : Pearson Education, 2009.

52. *Rosen I.* Auto-vectorization in GCC. GCC, the GNU Compiler Collection. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html (date of access: 29.07.2025).

53. Auto-vectorization in LLVM. The LLVM Compiler Infrastructure Project. URL: https://llvm.org/docs/Vectorizers.html (date of access: 29.07.2025).

54. *Naishlos D.* Autovectorization in GCC. Proceedings of the 2004 GCC developers' summit. 2004. P. 105–118. URL: https://people.redhat.com/lockhart/.gcc2004/MasterGCC-2side.pdf#page=105 (date of access: 16.09.2025).

55. Comparison of vectorization capabilities of different compilers for x86 and ARM cpus. N. Sakib, T. Prabhu, N. Santhi, J. Shalf, A.-H. A. Badawy. 2024 IEEE High Performance Extreme Computing Conference (HPEC). 2024. P. 1–7. URL: https://doi.org/10.1109/HPEC62836.2024.10938481 (date of access: 13.09.2025).

56. GCC, the GNU compiler collection - GNU project. GNU Project. URL: https://gcc.gnu.org/ (date of access: 13.09.2025).

57. Clang C language family frontend for LLVM. LLVM Project. URL: https://clang.llvm.org/ (date of access: 13.09.2025).

58. CUDA C++ programming guide. NVIDIA. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (date of access: 05.08.2025).

59. The OpenCL™ specification. Khronos Group. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_introduction (date of access: 05.08.2025).

60. *Kirk D. B.*, *Hwu W.-m. W.* Programming massively parallel processors: a hands-on approach. Waltham : Newnes, 2012. 519 p.

61. Hipacc: a domain-specific language and compiler for image processing. R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, W. Eckert. Ieee transactions on parallel and distributed systems. 2016. Vol. 27: 1. P. 210–224. URL: https://doi.org/10.1109/TPDS.2015.2394802 (date of access: 13.09.2025).

62. Convergence and scalarization for data-parallel architectures. Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, K. Asanović. Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2013. P. 1–11. URL: https://doi.org/10.1109/CGO.2013.6494995 (date of access: 05.08.2025).

63. Qualcomm FastCV SDK. Qualcomm. URL: https://www.qualcomm.com/developer/software/qualcomm-fastcv-sdk (date of access: 12.09.2025).

64. P-Im2col: simple yet efficient convolution algorithm with flexibly controlled memory overhead. A. V. Trusov, E. E. Limonova, D. P. Nikolaev, V. V. Arlazarov. Ieee access. 2021. Vol. 9. P. 168162–168184. URL: https://doi.org/10.1109/ACCESS.2021.3135690 (date of access: 13.09.2025).

65. *Приставка П. О.*, *Шевченко А. К.* Дослідження реалізації лінійного оператора згортки цифрового зображення при 16-бітних обчисленнях. Актуальні проблеми автоматизації та інформаційних технологій. 2016. Т. 20: 1. С. 78–90. URL: https://doi.org/10.15421/431608 (дата звернення: 29.07.2025).

66. *Prystavka P.*, *Shevchenko A.*, *Rokitianska I.* Comparative analysis of detector-tracker architecture for object tracking based on SBC for UAV. 2024 IEEE 7th international conference on actual problems of unmanned aerial vehicles development (APUAVD) (22–24 October 2024). 2024. P. 175–178. URL: https://doi.org/10.1109/APUAVD64488.2024.10765890 (date of access: 29.07.2025).

67. *Kelefouras V.*, *Keramidas G.* Design and implementation of deep learning 2d convolutions on modern cpus. IEEE trans. parallel distrib. syst. 2023. Vol. 34: 12. P. 3104–3116. URL: https://doi.org/10.1109/TPDS.2023.3322037 (date of access: 05.08.2025).

68. *Brkayy*. Optimizing matrix operations with NEON intrinsics and OpenMP: a practical guide. Medium. 2025. URL: https://medium.com/@brkayy987/optimizing-matrix-operations-with-neon-intrinsics-and-openmp-a-practical-guide-1c3088df5f21 (date of access: 12.09.2025).

69. *Jiang X.*, *Lin W.*, *Nagda A.* Optimizing matrix multiplication on NERSC's high performance computer cori. Proceedings of the Future Technologies Conference (FTC) 2023, Volume 3. 2023. P. 563–572. URL: https://doi.org/10.1007/978-3-031-47457-6_37.

70. *Kelefouras V.*, *Keramidas G.* Design and implementation of 2d convolution on x86/x64 processors. IEEE transactions on parallel and distributed systems. 2022.

Vol. 33: 12. P. 3800–3815. URL: https://doi.org/10.1109/TPDS.2022.3171471 (date of access: 05.08.2025).

71. *Shevchenko A.*, *Tymchyshyn V.* A SIMD-based approach to the enhancement of convolution operation performance. Proceedings of the international workshop on conflict management in global information networks (CMiGIN). 2019. Vol. 2588. P. 447–458. URL: https://ceur-ws.org/Vol-2588/paper37.pdf (date of access: 29.07.2025).

72. *Shevchenko A.*, *Prystavka P.*, *Tymchyshyn V.* Research on possible convolution operation speed enhancement via AArch64 SIMD. Advances in computer science for engineering and education. Lecture Notes on Data Engineering and Communications Technologies. Cham : Springer International Publishing, 2022. Vol. 134. P. 61–75. URL: https://doi.org/10.1007/978-3-031-04812-8_6.

73. *Shevchenko A.*, *Prystavka P.* Enhancement of convolution operation performance using SIMD of aarch64. Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II. CEUR Workshop Proceedings. 2024. Vol. 3826. P. 385–391. URL: https://ceur-ws.org/Vol-3826/#short32 (date of access: 29.07.2025).

74. *Шевченко А. К.* Підвищення швидкодії операції згортки RGB зображень з використанням алгоритму кластеризації ядер згортки для архітектури процесорів ARM64. Problems of informatization and management. 2025. Т. 1: 81. С. 140–149. URL: https://doi.org/10.18372/2073-4751.81.20144 (дата звернення: 18.08.2025).

75. TMS320C64x DSP library programmer's reference. Texas Instruments. 2003.

76. *IEEE*. IEEE standard for floating-point arithmetic. IEEE std 754-2019 (revision of IEEE 754-2008). 2019. P. 1–84. URL: https://doi.org/10.1109/IEEESTD.2019.8766229.

ДОДАТОК В

АКТ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОГО ДОСЛІДЖЕННЯ У НАВЧАЛЬНИЙ ПРОЦЕС НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

ЗАТВЕРДЖУЮ
Президія Національного університету
«Київський авіаційний інститут»
_____ Ксенія СЕМЕНОВА
(підпис) (Ім'я, ПРІЗВИЩЕ)
"__" _____ 20 __ року
М.П.

АКТ

Комісія у складі:
голови – (*в.о. зав. каф. ПМ ФКНТ, к.ф.-м.н., доцента, Олешко Тетяни Анатоліївни*);
членів комісії: – *доцента каф. ПМ ФКНТ, к.ф.-м.н., доцента Тупко Наталії Петрівни*
– *доцента каф. ПМ ФКНТ, к.ф.-м.н., доцента Хребта Валерія Григоровича*;
– *доцента каф. ПМ ФКНТ, PhD, Зівакіна Валерія Дмитровича*

цим Актом засвідчує, що результати дисертаційного дослідження здобувача кафедри прикладної математики факультету комп'ютерних наук та технологій Національного університету «Київський авіаційний інститут» Шевченка Андрія Костянтиновича (*Прізвище, ім'я, по батькові*) на тему: "Інформаційна технологія обробки графічних даних для SIMD-архітектур процесорів" (*Тема дисертації*) на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.06 «Інформаційні технології» (*назва установи, організації*) впроваджені в діяльності Національного університету «Київський авіаційний інститут».

Зокрема, у процесі підготовки навчальних програм та викладання курсів лекцій з дисциплін «Архітектура обчислювальних систем та операційні системи», «Системне програмування» та «Програмування мовою асемблер» за освітніми програмами «Прикладна математика» і «Прикладне програмне забезпечення» (освітні ступені бакалавр та магістр, спеціальність 113 «Прикладна математика») та за освітньою програмою «Аналітика даних та штучний інтелект» (освітній ступінь бакалавр, спеціальність 122 «Комп'ютерні науки») використано проміжні аналітичні матеріали дисертаційного дослідження, а саме:

— оновлено зміст освітніх компонентів «Архітектура обчислювальних систем та операційні системи» та «Програмування мовою асемблер» через включення у програму навчання модулів із застосування інлайн-асемблеру для оптимізації обчислень на архітектурі ARM64 на основі результатів дисертаційного дослідження;

— запроваджено лабораторні роботи з обробки цифрових зображень засобами інлайн-асемблера, що ілюструють методи підвищення швидкодії ресурсомістких частин програмного забезпечення;

— розроблено електронні засоби навчання (демонстраційні програми, шаблони досліджень та презентаційні матеріали), які імплементовано в робочі програми та силабуси й розміщено у навчальному середовищі;

— до робочих програм та силабусів зазначених навчальних дисциплін включено матеріали щодо технік низькорівневої оптимізації обчислень: кеш-ефективної організації доступу до даних, розгортання циклів та векторизації, а також методики експериментальної оцінки швидкодії програмних реалізацій операції згортки на платформах ARM64.

Голова комісії	к.ф.-м.н., доцент (науковий ступінь, вчене звання)	(підпис)	*Тетяна ОЛЕШКО* (Ім'я, ПРІЗВИЩЕ)
Члени комісії	к.ф.-м.н., доцент (науковий ступінь, вчене звання)	(підпис)	*Наталія ТУПКО* (Ім'я, ПРІЗВИЩЕ)
	к.ф.-м.н., доцент (науковий ступінь, вчене звання)	(підпис)	*Валерій ХРЕБЕТ* (Ім'я, ПРІЗВИЩЕ)
	PhD з інф. тех. (науковий ступінь, вчене звання)	(підпис)	*Валерій ЗІВАКІН* (Ім'я, ПРІЗВИЩЕ)

ДОДАТОК Г

ДОВІДКА ПРО ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОГО ДОСЛІДЖЕННЯ У ДІЯЛЬНОСТІ ТОВ «ЕЛІСАТ»

Товариство з обмеженою відповідальністю "ЕЛІСАТ"

ЕЛІСАТ

УКРАЇНА, 03148, м. Київ-148, просп. Леся Курбаса, 2-Б
Тел.: (050) 352-67-46, E-mail:elisat_ua@ukr.net

Вих. № 2 від «08» 06 2026

ДОВІДКА ПРО ВПРОВАДЖЕННЯ

Результати дисертаційного дослідження старшого викладача кафедри прикладної математики Національного університету «Київський авіаційний інститут» Шевченка Андрія Костянтиновича на тему: «Інформаційна технологія обробки графічних даних для SIMD-архітектур процесорів» на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.06 «Інформаційні технології» впроваджені в діяльності ТОВ «ЕЛІСАТ».

Зокрема, у процесі розробки та оптимізації вбудованого програмного забезпечення для пристроїв збору й обробки даних з датчиків використано аналітичні матеріали дисертаційного дослідження щодо методів векторизації операції згортки та технік низькорівневої оптимізації обчислень на процесорних архітектурах сімейства ARM, що підтвердило архітектурну незалежність запропонованих формалізованої нотації та методів оптимізації, а саме:

- застосовано розроблену в дисертації методику векторизованої реалізації операції згортки для прискорення цифрової фільтрації сигналів з датчиків (FIR-фільтрація) та попередньої обробки зображень з малороздільних камер і теплових (інфрачервоних) сенсорів на мікроконтролерах STM32 (ядра Cortex-M4/M7), із застосуванням запропонованих підходів до векторизації у поєднанні з SIMD-інструкціями DSP-розширення архітектури (CMSIS-DSP);
- реалізовано алгоритми виділення контурів (фільтр Собеля) на основі оптимізованої операції згортки для задач детектування об'єктів та руху у системах з обмеженими обчислювальними ресурсами;
- впроваджено техніки кеш-ефективної організації доступу до даних, розгортання циклів та векторизації обчислень, що дозволило підвищити швидкодію ресурсомістких частин програмного забезпечення;
- застосовано методику експериментальної оцінки швидкодії програмних реалізацій, розроблену в межах дисертаційного дослідження, для порівняльного аналізу оптимізованих та базових реалізацій алгоритмів обробки даних.

Директор ТОВ «ЕЛІСАТ» _____ Микола БЕРДАР
(підпис) (Ім'я, ПРІЗВИЩЕ)

ДОДАТОК Д

АКТ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОГО ДОСЛІДЖЕННЯ У ДІЯЛЬНОСТІ ТОВ «ІНСПІРІТО ТІМ»

“ЗАТВЕРДЖУЮ”
Директор
ТОВ “ІНСПІРІТО ТІМ”
Євген ФІЛЯК
06.05. 2026 р.

Акт впровадження

№ 2026.05.06-01
Місце складання: Київ

дисертаційного дослідження старшого викладача кафедри прикладної математики Національного університету «Київський авіаційний інститут» Шевченка Андрія Костянтиновича на тему: «Інформаційна технологія обробки графічних даних для SIMD-архітектур процесорів» на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.06 «Інформаційні технології»

Комісія ТОВ «ІНСПІРІТО ТІМ» у складі провідного інженера-конструктора Горбатюка А.В., інженера-конструктора Чуб’юка В.Ю. та інженера-тестувальника Колесніченка В.М. розглянула матеріали дисертаційного дослідження Шеченко А.К. на тему «Інформаційна технологія обробки графічних даних для SIMD-архітектур процесорів». Комісія визнала, що виконані дослідження є актуальними і мають подвійне призначення як для цивільного, так і для військового використання. Комісія засвідчує, що окремі результати дисертаційної роботи Шевченка А.К. були використані на підприємстві ТОВ «ІНСПІРІТО ТІМ».

Зокрема, у процесі розробки та технічного супроводу програмно-апаратних засобів обробки відеоданих з бортових камер безпілотних літальних апаратів використано аналітичні матеріали дисертаційного дослідження щодо методів векторизації операції згортки та технік високопродуктивної обробки графічних даних на процесорних архітектурах ARM64, а саме:

— застосовано методику векторизованої реалізації операції згортки, розроблену в межах дисертаційного дослідження, що дозволило суттєво підвищити швидкодію обробки відеопотоку в умовах обмежених обчислювальних ресурсів бортового обладнання;

— на основі зазначеної методики реалізовано мовою C/C++ та оптимізовано із застосуванням SIMD-розширення NEON архітектури ARM64 алгоритми швидкого виявлення аномалій на зображеннях для бортових одноплатних обчислювальних модулів;

— застосовано розроблені в дисертаційному дослідженні рекомендації, щодо використання оптимізованого алгоритму операції згортки у вбудованих системах реального часу при налаштуванні та конфігуруванні бортових обчислювальних модулів на базі процесорів ARM64 (Cortex-A), що забезпечило стабільну роботу розробленого програмного забезпечення;

— надано науково-технічні консультації щодо техніки низькорівневої оптимізації обчислень (кеш-ефективна організація доступу до даних, розгортання циклів, векторизація) та методики експериментальної оцінки швидкодії програмних реалізацій.

Члени комісії:

_____ Антон ГОРБАТЮК

_____ Володимир ЧУБ’ЮК

_____ Валерій КОЛІСНІЧЕНКО.