УДК 004.932.2:004.452.5:004.35 DOI: 10.18372/2073-4751.81.20144

Шевченко А.К.,
orcid.org/0000-0003-3863-0473,
e-mail: andrii.shevchenko@npp.kai.edu.ua

ПІДВИЩЕННЯ ШВИДКОДІЇ ОПЕРАЦІЇ ЗГОРТКИ RGB ЗОБРАЖЕНЬ З ВИКОРИСТАННЯМ АЛГОРИТМУ КЛАСТЕРИЗАЦІЇ ЯДЕР ЗГОРТКИ ДЛЯ АРХІТЕКТУРИ ПРОЦЕСОРІВ ARM64

Державний університет «Київський авіаційний інститут»

Вступ

Стрімкий розвиток технологій обробки цифрових зображень та відео-потоків призвів до постійного зростання вимог щодо швидкодії алгоритмів обробки даних, особливо для пристроїв з обмеженими обчислювальними можливостями (як сучасні *SBC*). Операція згортки є фундаментальною для широкого спектру задач цифрової обробки сигналів, включаючи фільтрацію зображень, виявлення контурів, розмиття, підвищення різкості та багато інших.

Сучасною тенденцією в області обробки цифрових зображень є впровадження високопродуктивних алгоритмів, оптимізованих для конкретних апаратних архітектур, зокрема для процесорів *ARM*64 з підтримкою *NEON*, що є розширеним набором *SIMD*-команд процесорів сімейства *ARMv*8-*A*. *SIMD*, як технологія, дозволяє значно підвищити швидкодію алгоритму за рахунок паралельної обробки даних на рівні однієї ко-манди (для одного) ядра процесора. Тобто паралелізація відбувається на рівні ко-манди процесора і, як наслідок, на рівні потоку.

Аналіз існуючих реалізацій операції згортки, зокрема функції *cv*::*filter*2*D*(…) з бібліотеки *OpenCV*, показує, що незважаючи на використання векторизації, їх продуктивність може бути суттєво покращена, особливо при роботі з розрідженими ядрами згортки - матрицями, що містять значну кількість нульових елементів (понад 25%).

Попередні дослідження авторів [1-3] продемонстрували ефективність використання 16-бітних цілочисельних обчислень та *SIMD*-оптимізацій для прискорення операції згортки на платформі *ARM*64. Однак попередні дослідження не враховували специфіку розріджених матриць, що призводило до виконання надлишкових операцій множення на нульові елементи.

У даній роботі пропонується роз-ширення раніше розробленого методу для обробки *RGB*-зображень з додаванням процедури кластеризації елементів ядер згортки. Запропонований підхід дозволяє групувати ненульові елементи однаково-го знаку, що забезпечує ефективний про-пуск операцій з нульовими елементами та оптимізацію обчислень для груп елементів з однаковими характеристиками.

Мета дослідження

Метою дослідження є розробка та експериментальна перевірка методу кластеризації для підвищення швидкодії векторизованої операції згортки при обробці *RGB*-зображень з використанням розріджених ядер згортки на процесорних платформах *ARM*64.

Математичні основи векторизованої операції згортки

Операція згортки є одночасно найпростішою у виконанні та найскладнішою у контексті ресурсоємності та оптимізації операцією:

$$F\left(p^{i,j}\right)=\sum_{ii=i-r_i}^{i+r_i}\sum_{jj=j-r_j}^{j+r_j}\gamma_{ii-i,jj-j}\,p_{ii,jj} \qquad (1)$$

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

Рівняння (1) є досить сумісною з функцією *cv*::*filter*2*D*(...) бібліотеки *OpenCV* і

в даному дослідженні основна увага буде зосереджена на прямокутних ядрах згортки і надалі така форма є за замовченням.

Формалізація операції згортки для RGB-зображень

Для фактичної реалізації ОЗ необхідно провести модифікацію рівняння (1), а саме переміщення орієнтації індексів рівняння до «0-індексованого вигляду», зміну імен та введення стилістики «само документованого» програмування у математичні вирази. Без зміни змісту (1), лише із зміною назв та використанням «0-індексації», отримуються наступні твердження:

Твердження 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 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*:

$$\begin{aligned} src_{src\_i,src\_j} &= (R_{src\_i,src\_j}, G_{src\_i,src\_j}, B_{src\_i,src\_j}), \\ dst_{dst\_i,dst\_j} &= (R_{dst\_i,dst\_j}, G_{dst\_i,dst\_j}, B_{dst\_i,dst\_j}), \end{aligned} \quad (2)$$

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:

$$\begin{aligned} R_{dst\_i,dst\_j} &\in [0;0\text{xFF}], \\ G_{dst\_i,dst\_j} &\in [0;0\text{xFF}], . \\ B_{dst\_i,dst\_j} &\in [0;0\text{xFF}]. \end{aligned} \quad (3)$$

Твердження 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*.

Векторизація операції згортки з використанням NEON64

Отже можна переписати ОЗ (1) у вигляді виразу:

$$dst_{src\_i+\left\lfloor\frac{flt\_rows}{2}\right\rfloor, src\_j+\left\lfloor\frac{flt\_cols}{2}\right\rfloor} = \sum_{flt\_i=0}^{flt\_rows-1}\sum_{flt\_j=0}^{flt\_cols-1} src_{src\_i+flt\_i, src\_j+flt\_j} \cdot flt_{flt\_i,flt\_j}, \quad (4)$$

де, матриця вхідного цифрового зображення *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 (3) використовується взаємо-пов'язаність індексів матриць *SRC* та *DST*. Т.ч. рівняння (3) є аналогом/продовженням основного рівняння (1).

Для спрощення рівняння (4) до рівня цілочисельних операцій, котрі використовуються здебільшого для спрощення обчислювань та підвищують швидкодію операції згортки, можна передставити (4) у наступному вигляді:

$$dst_{src\_i+\left\lfloor\frac{flt\_rows}{2}\right\rfloor, src\_j+\left\lfloor\frac{flt\_cols}{2}\right\rfloor} = flt\_div \cdot \left(\sum_{flt\_i=0}^{flt\_rows-1}\sum_{flt\_j=0}^{flt\_cols-1} src_{src\_i+flt\_i, src\_j+flt\_j} \cdot flt_{flt\_i,flt\_j}\right). \quad (5)$$

де, flt_div має тип *float*/*f*32. Дужки введені для позначення першочерговості виконання операцій.

Для підвищення швидкодії операції згортки (5) є необхідним використання векторізованих/*SIMD* обчислень, оскільки традиційні скалярні обчислення не забезпечують достатньої продуктивності для обробки ЦЗ у режимі реального часу на платформах *ARM*64. Векторизовані/*SIMD* обчислення дозволяють зробити це за одну машинну інструкцію (умовно), що значно підвищить швидкодію операції згортки. Однак повна векторизація виразу (5) не завжди можлива через архітектурні обмеження *SIMD NEON*64, які полягають у фіксованій ширині векторних регістрів (128

біт) та необхідності вирівнювання даних. Розміри цифрового зображення та ядра згортки можуть не бути кратними ширині векторних регістрів, що призводить до необхідності обробки "залишкових" елементів зображення. З цього слідує, що вираз (5) має бути розділено на дві частини: векторизовану частину, що виконує основний обсяг операцію згортки зображення за допомогою *NEON*64 обчислень з максимальною ефективністю, та завершальну частину, що обробляє залишкові елементи скалярними операціями. Така декомпозиція операції згортки забезпечує оптимальне використання обчислювальних ресурсів ЦП та гарантує коректні обчислювальні результати її виконання.

Векторизація операції згортки з використанням NEON64

Акт векторизації коду складається з декількох кроків, але найважливішим є крок визначення кількості векторних регістрів, що використовуються при акті завантаження. У даному дослідженні використовуються шість векторних регістрів для завантаження/збереження базової частини рядка зображення розміром в 32 8-бітних трійок *RGB*. Це призводить, наприкінці, до стабільно зайнятих 24 векторних регістрів. З цього випливає, що кожен канал цифрового зображення 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 векторізованої операції згортки для зображення знаючи: розміри зображення; розміри ядра згортки; базову кількість векторів для збереження одного каналу зображення – 2. Кількість саме векторизованих операцій згортки для одного рядка ЦЗ визначається наступним чином:

$$NC\_vec = \left\lfloor \frac{src\_cols - flt\_cols + 1}{32} \right\rfloor, \quad (6)$$

де, $\lfloor \square \rfloor$ - 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:

$$st32N = \overline{0, NC\_vec - 1}. \quad (7)$$

Знаючи NC_vec (6) можна обраховувати загальну кількість саме векторізованих операцій згортки для всього зображення:

$$N\_vec = NC\_vec \cdot NR\_vec. \quad (8)$$

де, $NR\_vec = src\_rows - flt\_rows + 1$.

Використовуючи (6-8) можна записати (5) у вигляді, що описує обчислення тільки векторизованої частини операції згортки зображення:

$$dst_{src\_i + \left\lfloor \frac{flt\_rows}{2} \right\rfloor, idx_{base}(st32N) + j + \left\lfloor \frac{flt\_cols}{2} \right\rfloor} = flt\_div \cdot \left(\sum_{flt\_i=0}^{flt\_rows-1} \sum_{flt\_j=0}^{flt\_cols-1} src_{src\_i + flt\_i,\ idx_{base}(st32N) + j + flt\_j} \cdot flt_{flt\_i, flt\_j} \right), \quad (9)$$

де, $idx_{base}(st32N) = 32 \cdot st32N$, $j = \overline{0,31}$.

Вираз (9) схематично можна розділити на дві частини: обрахунок векторізованого скалярного добутку по індексу *flt_i* для кожного рядка ядра згортки, та обрахунок векторізованого скалярного добутку для кожного стовпця ядра по індексу *flt_i*, що і є найважливішою частиною даного дослідження. Операція нормалізації (множення на *flt_div*) є операцією, що має незначний вплив на загальну обчислювальну складність алгоритму (швидкодія векторізованого алгоритму операції згортки не залежить сильно від неї), хоча й є необхідною для коректного завершення векторізованої операції згортки.

Щодо завершальних обчислень операції згортки для всього зображення, вони виконуються за допомогою стандартної мови C/C++ і не будуть наділі розглянуті, хоча слід відмітити, що запис коду слід виконати у стилі кеш-френдлі коду.

Розглянемо ЯЗ flt_{flt_i, flt_j}, що описане виразом (3.3), 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

idx_mp 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 *idx_mp* описує кластери ненульових, однакових за знаком елементів ЯЗ.

Введемо змінну *k*, що буде ітератором для елементів у поточному рядку матриці *idx_mp*($k \in [0; flt\_cols - 1]$), яка змінюватиме свої значення за певних умов, що будуть описані далі. Матриця *idx_mp* складається з пар елементів/значень $idx\_mp_{flt\_i,flt\_j} = (ofst_{flt\_i,flt\_j}, cnt_{flt\_i,flt\_j})$ і на початку обробки ініціалізована як $idx\_mp_{flt\_i,flt\_j} = (-1,-1)$. Нехай $sign(x)$ – функція знаку числа, результатом роботи якої є:

$$sign(x) = \begin{cases} -1, & if\ x < 0, \\ 0, & if\ x = 0, \\ 1, & if\ x > 0. \end{cases} \quad (10)$$

Введемо змінну $prev\_sign$, яка зберігає знак попереднього елемента в рядку *flt_i* матриці *flt*. Початкове значення $prev\_sign = sign(flt_{flt\_i,0})$. Тоді умову формування елементів матриці $idx\_mp_{flt\_i,flt\_j}$ можна записати наступним чином:

$$\forall flt\_i \in \overline{0, flt\_rows - 1}, \forall flt\_j \in \overline{1, flt\_cols - 1}$$
$$\text{та } k \in [0; flt\_cols - 1],$$

$$idx\_mp_{flt\_i,k} = \begin{cases} (ofst_{flt\_i,k}, cnt_{flt\_i,k} + 1),\ if: \\ sign(flt_{flt\_i,flt\_j}) = prev\_sign\ and\ sign(flt_{flt\_i,flt\_j}) \neq 0. \\ (ofst_{flt\_i,k}, -1 \cdot cnt_{flt\_i,k}),\ if: \\ prev\_sign < 0\ and\ sign(flt_{flt\_i,flt\_j}) \neq prev\_sign. \\ (ofst_{flt\_i,k}, cnt_{flt\_i,k}),\ if: \\ prev\_sign > 0\ and\ sign(flt_{flt\_i,flt\_j}) \neq prev\_sign. \end{cases} ,(11)$$

$$k = \begin{cases} k+1, & if\ sign(flt_{flt\_i,flt\_j}) \neq prev\_sign\ i\ sign(flt_{flt\_i,flt\_j}) \neq 0 \\ k, & otherwise \end{cases}$$

$$index\_map_{flt\_i,k} = \begin{cases} (flt\_j, 1),\ if\ sign(flt_{flt\_i,flt\_j}) \neq prev\_sign \\ index\_map_{flt\_i,k},\ otherwise \end{cases}$$

$$prev\_sign = sign(flt_{flt\_i,flt\_j})$$

Процес формування матриці *idx_mp* можна описати наступним чином:

1. Якщо знак поточного елемента flt_{flt_i,flt_j} співпадає зі знаком попереднього елемента $\left(sign(flt_{flt\_i,flt\_j}) = prev\_sign\right)$ та не дорівнює нулю $\left(sign(flt_{flt\_i,flt\_j}) \neq 0\right)$, то збільшуємо лічильник $cnt_{flt\_i,k}$ на 1 і зберігаємо оновлений елемент $\left(ofst_{flt\_i,k}, cnt_{flt\_i,k} + 1\right)$ в $idx\_mp_{flt\_i,k}$;

2. Якщо знак поточного елемента flt_{flt_i,flt_j} відрізняється від знаку попереднього елемента $\left(sign(flt_{flt\_i,flt\_j}) \neq prev\_sign\right)$ та: $prev\_sign < 0$, то поточний елемент $cnt_{flt\_i,k}$ в $idx\_mp_{flt\_i,k}$ буде помножено на -1, тобто $\left(ofst_{flt\_i,k}, -1 \cdot cnt_{flt\_i,k}\right)$; якщо $prev\_sign > 0$ то значення $idx\_mp_{flt\_i,k}$ не станеться;

3. Якщо знак поточного елемента flt_{flt_i,flt_j} відрізняється від знаку попереднього елемента $\left(sign(flt_{flt\_i,flt\_j}) \neq prev\_sign\right)$ та не дорівнює нулю $\left(sign(flt_{flt\_i,flt\_j}) \neq 0\right)$, 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 $idx\_mp_{flt\_i,k} = (flt\_j, 1)$;

5. Оновлюємо значення *prev_sign* знаком поточного елемента flt_{flt_i,flt_j} : $prev\_sign = signum(flt_{flt\_i,flt\_j})$.

Сформована матриця $idx\_mp_{flt\_i,k}$ служить орієнтиром по кластерах позитивних та негативних елементів flt_{flt_i,flt_j} в умовах, коли ядро згортки має вигляд розрідженої матриці, та/або коли її елементи групуються за знаком. Т.ч.

запропонований алгоритм [1, 2], можна модифікувати з використанням матриці $idx\_mp_{flt\_i,k}$ для підвищення ефективності/швидкодії обчислень операції згортки зображення.

Також слід зазначити, що кількість однакових за знаком елементів ядра згортки суттєво впливає на продуктивність самої операції згортки в реальних ситуаціях. Мається на увазі кластерізування/групування елементів по 1, 2 та 3 елемента, що за одну ітерацію зовнішнього циклу проходу по $idx\_mp_{flt\_i,k}$[1] операцій призводить до розгортання циклу програми [3, 4]. Таким чином, векторізована операція згортки може бути значно прискорена.

Алгоритм кластеризації елементів ядра згортки

Прикладом ядр згортки, що значно підвищують якість зображення є наступні приклади ядер:

1. Субполосна фільтрація:

$$\gamma H = \frac{1}{36}\begin{pmatrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{pmatrix}. \qquad (12)$$

2. Субполосна фільтрація, що забезпечує більшу, ніж в (**Error! Reference source not found.** разів) ступінь згладжування:

$$\underset{\gamma H}{x} = \frac{1}{121}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -6 & -15 & -6 & 0 \\ -1 & -15 & 88 & -15 & -1 \\ 0 & -6 & -15 & -6 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}. \qquad (13)$$

3. Контрастування цифрового зображення:

$$\gamma K = \frac{1}{16}\begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 3 & -13 & 3 & 0 \\ 2 & -13 & 48 & -13 & 2 \\ 0 & 3 & -13 & 3 & 0 \\ 0 & 0 & 2 & 0 & 0 \end{pmatrix}. \qquad (13)$$

Запропоновані ядра згортки значно підвищують якість цифрового зображення [1-3].

Особливості реалізації з від'ємними елементами ядра

Описані математичні основи та підхід (1, 9, 11) до реалізації операції згортки передбачають, що ядро згортки може містити як додатні, так і від'ємні елементи. Однак, якби всі елементи ядра згортки, описаного виразом (1), були додатними, швидкодія обчислень могла б бути вищою. Це пояснюється невеликим дослідженням у вигляді проведеного експерименту, у рамках якого отримано показники, що свідчать про те, що операція згортки над додатними елементами ядра згортки виконується незначно, але швидше (приблизно на 10–15%). Таке прискорення не є суттєвим, однак воно спостерігається за умови, що всі елементи ядра згортки є додатними і відсутні додаткові попередні обчислення. Натомість, якщо елементи ядра згортки мають від'ємні значення, у реалізації векторізованої операції згортки, що використовує $idx\_mp_{flt\_i,k}$, слід використовувати 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 у конструкціях *switch* та *if* не є стандартним і обумовлене необхідністю якнайшвидшого переходу до наступної ітерації циклу. Як наслідок, оператор *switch* спричиняє зниження швидкодії векторізованої операції згортки для обробки зображення на 5–10%. 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 продуктивності. Дане дослідження базується на припущенні, що ядро згортки містить від'ємні значення, через що в основі ВОЗ застосовуються оператор *switch* та цикли над ним + передобрахунок матриці $idx\_mp_{flt\_i,k}$. Для підвищення швидкодії було використано стратегію розгортання циклу, описану в [2, 3], яка суттєво покращила продуктивність векторізованої операції згортки зображення (навіть за наявності від'ємних значень ядрі згортки), порівняно зі стандартним ітеративним підходом [1]. Останній передбачає виконання лише однієї операції за ітерацію, зокрема обчислення скалярного добутку для одного значення ядра згортки [2, 3].

Техніки оптимізації: розгортання циклів та плитковість

Розгортання циклів є однією з ключових технік оптимізації програмного коду, яка спрямована на підвищення продуктивності шляхом зменшення накладних витрат, пов'язаних з ітераціями оператору 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 з цим тілом є значною. Розгортання циклів дозволяє замінити ітеративну конструкцію на лінійну послідовність команд, усуваючи або значно скорочуючи кількість умовних та перевірок на завершення циклу.

На архітектурі *ARM*64, кожна інструкція виконується за фіксований час, а продуктивність залежить від ефективності конвеєра команд. Умовні переходи, характерні для циклів, можуть призводити до зривів конвеєра (*pipeline stalls*) через помилки передбачення гілок (*branch misprediction*), що і знижує загальну пропускну здатність процесора. Розгортання циклів мінімізує ці ефекти, дозволяючи процесору виконувати більше корисної роботи.

Поряд з розгортанням циклів часто застосовується техніка плитковості [4], яка тісно пов'язана з оптимізацією використання кеш-пам'яті. Ця техніка 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 призвела до значного підвищення продуктивності, особливо для алгоритмів операції згортки, котрий працює з великими обсягами даних

Мінусом таких технік є те що вони збільшують розмір машинного коду. Але сучасні *ARM*64-процесори мають достатньо великі кеші команд (наприклад, 64 КБ *L*1 у *Cortex-A*76), що мінімізує вплив цього фактора і зазвичай швидкість виконання є більш пріоритетною для систем з невеликими обчислювальними ресурсами, у *SBC* на базі архітектури *ARM*64.

Порівняльний аналіз з OpenCV Universal Intrinsics

Огляд альтернативних методів векторізації/оптимізації коду слід продовжити з *OpenCV Universal Intrinsics* [5]. *OpenCV Universal Intrinsics* є тонким абстрактним шаром, що надає єдиний набір *SIMD*-примітивів для різних наборів інструкцій (*SSE*/*AVX*, *NEON*, *VSX*). Детальний перелік векторних типів, функцій завантаження, розширення, *FMA*-обчислень та пакування подано в офіційній документації і у внутрішньому технічному аналізі [6].

Переваги підходу *OpenCV Universal Intrinsics*: 1) Переносимість – один код охоплює *x*86-64, *ARM* і *POWER* без дублювання; 2) Продуктивність – 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 *float*32 усувають переповнення та підтримують дробові ядра.

Спрощений стек викликів функції *cv*::*filter*2*D*(…) до моменту виклику векторізованої функції *FilterVec*_8*u*::*operator*() є наступним:

1. *filter*2*D*(…);
2. *cv*::*hal*::*filter*2*D*(…);
3. *cv*::*ocvFilter*2*D*(…);
4. *cv*::*FilterEngine*::*apply*(…);
5. *cv*::*cpu_baseline*::*FilterEngine*(*__apply*/*__proceed*)(…);
6. *cv*::*cpu_baseline*::*Filter*2*D*<…>::*operator*();
7. *cv*::*cpu_baseline*::*FilterVec*_8*u*::*operator*().

Тож у звичайному виклику *cv*::*filter*2*D*(…) стек виконує близько восьми шарів, доки не дійде до спеціалізованого обробника 8-бітних даних. Алгоритмічний контур *FilterVec*_8*u* є наступним:

1. Завантаження блоків пікселів у *SIMD*-регістри;
2. Розширення 8-бітних значень до розрядів, придатних для обчислень;
3. Перетворення у *float*32 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 наявним *SIMD*-розширеннями. Отжєе, застосування *OpenCV UI* в *cv*::*filter*2*D*(…) демонструє сучасну стратегію оптимізації [5, 6].

Розгортання циклів є однією з ключових технік оптимізації програмного коду, яка спрямована на підвищення продуктивності шляхом зменшення накладних витрат, пов'язаних з ітераціями оператору циклу [4].

Результати експериментального дослідження

Тестування проводилось на апаратній платформі *OrangePi* 5 *Pro*. Для кожної комбінації параметрів (розмір ЯЗ та роздільна здатність фрейму зображення) було зібрано статистично значущу вибірку показників продуктивності у кадрах на секунду (*FPS*), що дозволило об'єктивно оцінити ефективність обох реалізацій та їх порівняльні характеристики.

Візуалізація отриманих представлена у вигляді «матриці» 8x13. Слід зазначити, що по осі Ox розташовано ширину фрейму/зображення (котрий відображає стандартні розміри зображень відеофайлів: 256*x*144, 426*x*240, 640*x*360, 854*x*480, 1280*x*720, 1920*x*1080, 2560*x*1440, 3840*x*2160), а по осі Oy розмір ядра згортки у форматі одного числа – *N* (*N*x*N*). Розміри ядра згортки змінюються з кроком 1, починаючи з 2x2 до 15x15. Після досягнення максимального розміру (15x15) тестовий скрипт переходить до обробки наступного відео з іншою роздільною здатністю. Весь описаний процес повторюється для кожного відеофайлу.

Цифра/колір матриці відображає медіанний показник *FPS*, котрий обраховано при збереженні 150 показників, при обробці, наприклад, розміру зображення 1920*x*1080 та ядра згортки 5x5. Тож тестовим скриптом, фіксується визначену кількість показників *FPS*, але не більше обраного значення, для заданого розміру фрейму зображення та ядра згортки, обраховується медіана і зберігається як показник в матриці результату. Результатом такого тестування є матриця розміром 8x13.

Отже матриця (8x13) представлена у декількох варіаціях: 1) матриці медіанних показників *FPS* для кожного алгоритму окремо (для запропонованого алгоритму векторізованої операції згортки (далі *conv_RGB_NEON*(…)) [2, 3] та *cv*::*filter*2*D*(...)); 2) матриця порівняння швидкодії - *conv_RGB_NEON*(…)) *VS* *cv*::*filter*2*D*(...).

Отже слід надати одразу 3 матриці (8x13) накопичених результаті показників, котрі: 1) накопичені при використанні *cv*2::*filter*2*D*(…) (у *PFS*); 2) накопичені при використанні алгоритму [3], що розширено до обробки 8-біт *RGB* зображення, тобто *conv_RGB_NEON*(…) (у *PFS*); 3) порівняння цих двох алгоритмів методом ділення показників *conv_RGB_NEON*(…) на показники *cv*2::*filter*2*D*(…). Т.ч. третя матриця дозволить наочно представити

переваги *conv_RGB_NEON*(…) порівняно з *cv*2::*filter*2*D*(…).

Аналіз результатів показує (рис. 1), що показники *FPS* функції *cv*2::*filter*2*D*(…) різко знижуються при переході від ядер згортки розміром 7×7 до 8×8: падіння з 869.5 *FPS* (при 7×7) до 159.5 *FPS* (при 8×8) для зображення 256×144 - зниження в ~5.5 разів. Причиною цього є виявлений у вихідному коді *OpenCV* механізм автоматичного вибору алгоритму операції згортки. У файлі (*filter.dispatch.cpp*:1275) в функції *dftFilter*2*D*(...) присутня така умова – «*if* (*kernel_width* * *kernel_height* < *dft_filter_size*) { *return false*;}». Ця умова визначає, який алгоритм буде використано: при невиконанні умови (для ядер згортки до 7×7) використовується більш ефективна реалізація (що згадана у попередньому розділі), а при виконанні операції згортки з розмірами ядер від 8×8 і більше, виконується операція згортки на основі швидкого перетворення Фур'є (*FFT*). Хоча теоретично *FFT*-алгоритм має кращу асимптотичну складність для дуже великих ядер, на практиці для ядер згортки середнього розміру він виявляється значно повільнішим через додаткові накладні витрати на перетворення Фур'є та менш ефективне використання кеш-пам'яті.

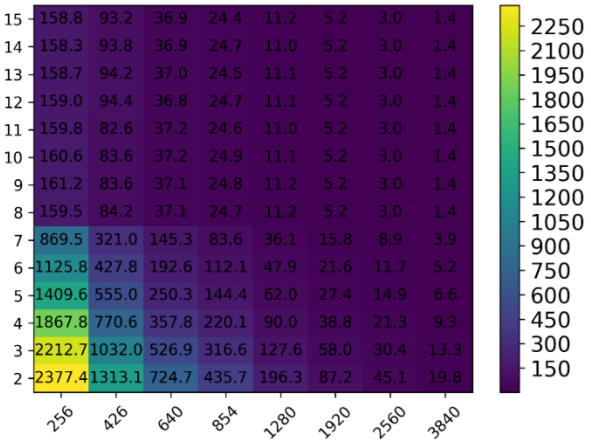

Рис. 1. Продуктивність функції *cv*::*filter*2*D*() для різних розмірів ядер згортки та роздільної здатності зображень (*FPS*).

Показники роботи методу *conv_RGB_NEON*(…) демонструють перевагу над *cv*2::*filter*2*D*(…) для всіх розмірів ядер згортки (рис. 2). Особливо значним є показник для ядер згортки розміром 8×8 і більше (навіть для малих розмірів ядер згортки (2×2 - 7×7), де *cv*2::*filter*2*D*(…) показує хорошу продуктивність) є швидшим завдяки використанню оптимізованого підходу [3], що був розширений до обробки 8-біт *RGB* зображення.

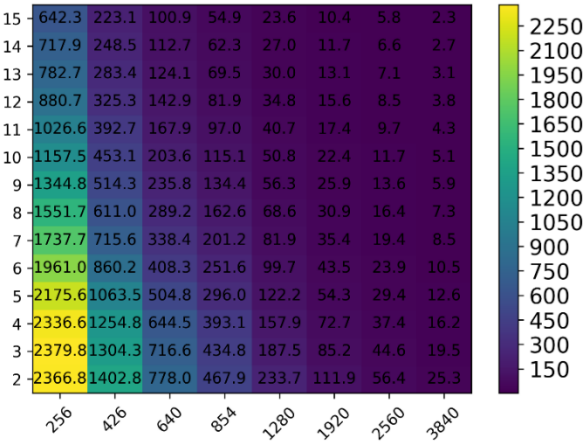

Рис. 2. Продуктивність розробленого алгоритму *conv_RGB_NEON*() для різних розмірів ядер згортки та роздільної здатності зображень (*FPS*).

На практиці запропонований метод дозволяє: мінімізувати кількість звернень до пам'яті; не потребує використання кеш-пам'яті, бо, натомість, максимально ефективно використовує векторні регістри *NEON*64; уникає накладних витрат на повторне завантаження даних зображення.

Слід зауважити, що для обох методів спостерігається закономірне зниження продуктивності зі збільшенням роздільної здатності ЦЗ, але *conv_RGB_NEON*(…) демонструє кращу масштабованість.

Особливо це помітно для розмірів фреймів високої роздільної здатності (1920×1080 і вище), де перевага *conv_RGB_NEON*(…) залишається суттєвою навіть для великих ядер згортки (рис. 3).

Отже тестова платформа *Orange Pi* 5 *Pro* наочно демонструє (рис. 3) наступні патерни прискорення *conv_RGB_NEON*(…) відлносно *cv*2::*filter*2*D*(…):

- максимальне прискорення у 9.7*x* для ядер розміром 7-11;

- помірне прискорення у 0.9-2.2*x* для малих ядер розміром 2-6;
- стабільно високе прискорення у 1.6 - 6.1*x* для великих ядер розміром 12-15.

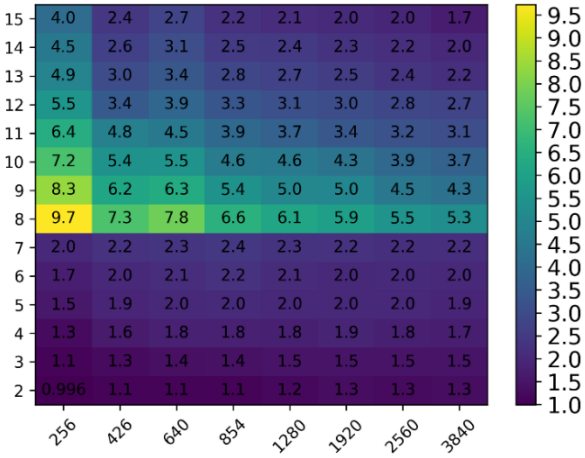

Рис. 3. Коефіцієнт прискорення алгоритму *conv_RGB_NEON*() відносно *cv*::*filter*2*D*() для різних параметрів обробки

Висновки

У даному досліджені проведено експериментальне дослідження швидкодії розробленої операції згортки *conv_RGB_NEON*(…) із застосуванням на основі технології *NEON*64 та алгоритму [3], що використовує 16-бітні цілочисельні обчислення та розширено до робити з 8-біт *RGB* зображеннями. На основі отриманих результатів зроблено такі висновки:

1. Проведено порівняльний аналіз швидкодії (в *FPS*) розробленого алгоритму *conv_RGB_NEON*(...) та функції *cv*::*filter*2*D*(...) бібліотеки *OpenCV* для *C*++ реалізацій. Встановлено, що розроблений алгоритм демонструє значну перевагу в швидкодії на обох тестових платформах:
   - для ядер середнього розміру (7×7 – 11×11) прискорення становить 5,0–9,7 разів;
   - для великих ядер згортки (12×12 – 15×15), де *cv*::*filter*2*D*(...) переходить на використання *FFT*-алгоритмів (що підтверджується різким падінням її швидкодії для ядер розміром 8×8 та більше), запропонований метод прямої згортки з *NEON*64 оптимізацією показує прискорення в 1,7–5,5 разів;
   - для малих ядер (2×2 – 6×6) прискорення є помірним (1,1–2,2 рази), що пояснюється меншим впливом оптимізацій на загальний час виконання для невеликих обчислювальних навантажень.
2. Експериментальні дослідження повністю підтвердили високу ефективність розробленої інформаційної технології та оптимізованого алгоритму *conv_RGB_NEON*(...), що дозволяє суттєво підвищити продуктивність обробки зображень на сучасних одноплатних комп'ютерах з архітектурою ЦП *ARM*64.
3. Використання техніки розгортання циклів, що також було використано в [2, 3], дало підвищення швидкодії для алгоритму *conv_RGB_NEON*(...).

Література

1. Приставка П. О., Шевченко А. К. Дослідження реалізації лінійного оператора згортки цифрового зображення при 16-бітних обчисленнях. *Проблеми програмування*. 2016. № 2-3. С. 207–217. DOI: 10.15421/431608.
2. Shevchenko A., Tymchyshyn V. A SIMD-based approach to the enhancement of convolution operation performance. International Workshop on Conflict Management in Global Information Networks (CMiGIN 2019) : proceedings, Lviv, Ukraine, November 29, 2019 / 2019. P. 447–458. URL: https://ceur-ws.org/Vol-2588/paper37.pdf
3. Shevchenko A., Prystavka P., Tymchyshyn V. Research on Possible Convolution Operation Speed Enhancement via AArch64 SIMD. Lecture Notes on Data Engineering and Communications Technologies. Vol. 134. Advances in Computer Science for Engineering and Education / ed. by Z. Hu et al, 2022. P. 61–75. DOI: doi.org/10.1007/978-3-031-04812-8_6.
4. Fog A. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. Copenhagen : Copenhagen University College of Engineering, 2024. URL : https://www.agner.org/optimize/optimizing_cpp.pdf (access date: 26.05.2025.)
5. Universal intrinsics / OpenCV 4.x Main Documentation. URL:

https://docs.opencv.org/4.x/d6/dd1/tutorial_univ_intrin.html. (access date 26.05.2025.)

6. HAL (Hardware Acceleration Layer) Explanations / OpenCV GSoC 2016 ideas ; GitHub. URL: https://github.com/opencv/opencv/wiki/GSoC_2016_ideas_HAL_Explanations (access date: 26.05.2025.)

Шевченко А.К.

ПІДВИЩЕННЯ ШВИДКОДІЇ ОПЕРАЦІЇ ЗГОРТКИ RGB ЗОБРАЖЕНЬ З ВИКОРИСТАННЯМ АЛГОРИТМУ КЛАСТЕРИЗАЦІЇ ЯДЕР ЗГОРТКИ ДЛЯ АРХІТЕКТУРИ ПРОЦЕСОРІВ ARM64

У статті представлено метод підвищення швидкодії операції згортки RGB-зображень на платформі ARM64 з використанням алгоритму кластеризації елементів ядер згортки. Запропонований підхід базується на векторизації обчислень з використанням SIMD-інструкцій NEON64 та групуванні ненульових елементів ядра згортки однакового знаку для ефективного пропуску операцій з нульовими елементами. Розроблено математичну модель векторизованої операції згортки, яка враховує специфіку розріджених матриць ядер згортки. Експериментальне дослідження на платформі Orange Pi 5 Pro продемонструвало значне прискорення порівняно з функцією cv::filter2D() бібліотеки OpenCV: для ядер середнього розміру (7×7 – 11×11) досягнуто прискорення в 5,0–9,7 разів, для великих ядер (12×12 – 15×15) – в 1,7–5,5 разів. Запропонований метод особливо ефективний для обробки зображень високої роздільної здатності та може бути застосований у системах реального часу на одноплатних комп'ютерах з обмеженими обчислювальними ресурсами.

***Ключові слова**: операція згортки; NEON64; ARM64; SIMD-оптимізація; векторизація; RGB-зображення, кластеризація ядер згортки; цифрова обробка зображень; розріджені матриці; OpenCV.*

Shevchenko A.K.

PERFORMANCE ENHANCEMENT OF RGB IMAGE CONVOLUTION USING CONVOLUTION KERNEL CLUSTERING ALGORITHM FOR ARM64 PROCESSOR ARCHITECTURE

The paper presents a method for improving the performance of RGB image convolution operation on the ARM64 platform using a convolution kernel element clustering algorithm. The proposed approach is based on vectorization of computations using NEON64 SIMD instructions and grouping of non-zero kernel elements with the same sign for efficient skipping of operations with zero elements. A mathematical model of vectorized convolution operation has been developed, which takes into account the specifics of sparse convolution kernel matrices. Experimental study on the Orange Pi 5 Pro platform demonstrated significant acceleration compared to the cv::filter2D() function of the OpenCV library: for medium-sized kernels (7×7 – 11×11), an acceleration of 5.0–9.7 times was achieved, for large kernels (12×12 – 15×15) – 1.7–5.5 times. The proposed method is particularly effective for processing high-resolution images and can be applied in real-time systems on single-board computers with limited computational resources.

***Keywords**: convolution operation; NEON64; ARM64; SIMD optimization; vectorization; RGB images; convolution kernel clustering; digital image processing; sparse matrices; OpenCV.*