УДК 004.4’24 https://doi.org/10.15407/pp2020.02-03.105

АВТОМАТИЗОВАНЕ ПРОЕКТУВАННЯ ТА РОЗПАРАЛЕЛЮВАННЯ ПРОГРАМ ДЛЯ ГЕТЕРОГЕННИХ ПЛАТФОРМ ІЗ ВИКОРИСТАННЯМ АЛГЕБРО-АЛГОРИТМІЧНОГО ІНСТРУМЕНТАРІЮ

А.Ю. Дорошенко, О.Г. Бекетов, М.М. Бондаренко, О.А. Яценко

Запропоновані методи та інструментальні засоби автоматизованого проектування та генерації OpenCL програм на основі алгебри алгоритмів. Розроблено метод напівавтоматичного розпаралелювання циклічних операторів на основі серіалізації даних. Розглянутий підхід полягає у використанні високорівневих алгебро-алгоритмічних специфікацій програм, що подаються у природно-лінгвістичній формі. Розроблені інструментальні засоби забезпечують автоматизоване проектування схем алгоритмів шляхом суперпозиції конструкцій алгебри Глушкова. Інструментарій автоматично виконує генерацію програм цільовою мовою програмування на снові специфікацій. Розроблений програмний засіб паралелізації циклів для оптимізації обчислень за допомогою графічних прискорювачів дозволяє в напівавтоматичному режимі здійснювати перехід від послідовних до паралельних програм і використовує систему переписувальних правил для трансформації програм. Застосування підходу проілюстроване на розробці паралельної OpenCL програми згортки зображень.

Ключові слова: автоматизоване проектування програм, алгебра алгоритмів, гетерогенні системи, графічний прискорювач, згортка зображень, паралельні обчислення, синтез програм, GPU, OpenCL.

Предложены методы и инструментальные средства автоматизированного проектирования и генерации OpenCL программ на основе алгебры алгоритмов. Разработан метод полуавтоматического распараллеливания циклических операторов с использованием сериализации данных. Рассмотренный подход заключается в использовании высокоуровневых алгебро-алгоритмических спецификаций программ, которые представляются в естественно-лингвистической форме. Разработанные инструментальные средства обеспечивают автоматизированное проектирование схем алгоритмов путем суперпозиции конструкций алгебры Глушкова. Инструментарий автоматически выполняет генерацию программ на целевом языке программирования на основе спецификаций. Разработанное программное средство параллелизации циклов для оптимизации вычислений с использованием графических ускорителей позволяет в полуавтоматическом режиме осуществлять переход от последовательных к параллельным программам и использует систему переписывающих правил для трансформации программ. Применение подхода проиллюстрировано на разработке параллельной OpenCL программы свертки изображений.

Ключевые слова: автоматизированное проектирование программ, алгебра алгоритмов, гетерогенные системы, графический ускоритель, свертка изображений, параллельные вычисления, синтез программ, GPU, OpenCL.

Methods and software tools for automated design and generation of OpenCL programs based on the algebra of algorithms are proposed. OpenCL is a framework for developing parallel software that executes across heterogeneous platforms consisting of general-purpose processors and/or hardware accelerators. The proposed approach consists in using high-level algebra-algorithmic specifications of programs represented in natural linguistic form and rewriting rules. The developed software tools provide the automated design of algorithm schemes based on a superposition of Glushkov algebra constructs that are considered as reusable components. The tools automatically generate code in a target programming language on the basis of the specifications. In most computing problems, a large part of hardware resources is utilized by computations inside loops, therefore the use of automatic parallelization of cyclic operators is most efficient for them. However, the existing automatic code parallelizing tools, such as Par4All, don’t account the limited amount of accelerator’s onboard memory space while real-life problems demand huge amounts of data to be processed. Thus, there is a need for the development of a parallelization technique embracing the cases of massive computational tasks involving big data. In the paper, a method and a software tool for semi-automatic parallelization of cyclic operators based on loop tiling and data serialization are developed. The parallelization technique uses rewriting rules system to transform programs. The framework for parallelization of loops for optimization of computations using graphics processing units allows semi-automatic parallelization of sequential programs. The approach is illustrated on an example of developing a parallel OpenCL image convolution program.

Key words: algorithm algebra, automated software design, graphics processing unit, heterogeneous systems, image convolution, OpenCL, parallel computation, software synthesis.

Вступ

Стандарт OpenCL, розроблений Kronos Group [1], на даний час є доволі популярним серед розробників програмного забезпечення для розв’язання задач великої обчислювальної складності. Він ґрунтується на мові C і дозволяє суттєво скоротити час розробки коду для гетерогенних систем з найрізноманітнішим складом обчислювальних пристроїв. Широке розповсюдження стандарт отримав в основному завдяки підтримці з боку модулів графічних прискорювачів. На відміну від свого аналога Nvidia CUDA CUDA SDK [2], що є реалізацією GPGPU (обчислення загального призначення на графічних процесорах) лише для відеокарт Nvidia, OpenCL є специфікацією, яку можуть реалізовувати різні виробники апаратного забезпечення (Nvidia, AMD, Intel, ARM та ін.). Відмітимо, що програмування гетерогенних систем є досить складною задачею порівняно з розробкою застосувань для однорідних архітектур, тому актуальним є питання розробки спеціальних засобів автоматизації розробки програмного забезпечення, що дозволяли б найбільш ефективно генерувати найпродуктивніший код для таких систем.

В більшості обчислювальних задач значну частину апаратних ресурсів витрачають обчислення, що здійснюються всередині циклів, тому використання автоматичної паралелізації на рівні потоків найбільш ефективне саме для них. Так, наприклад, більшість задач математичної фізики розв'язуються чисельними методами з використанням сіткових обчислень. До виникнення циклів призводять різницеві схеми, метод скінченних елементів, загалом, задачі, що вимагають виконання операцій над матрицями даних. Розпаралелювання циклічних операторів є давно відомою проблемою програмування. З широким використанням графічних прискорювачів для обчислювальних задач виникла нова постановка цієї задачі для цього класу мультипроцесорних систем. Існуючі засоби розпаралелювання циклів [3, 4] не враховують обмежений об'єм пам'яті графічних прискорювачів, в той час як реальні задачі вимагають обробки великих об'ємів даних. Таким чином, існує необхідність у розробці методу паралелізації циклів програм, що працюють з великими даними.

У попередніх роботах [5–8] авторами розроблялися теорія, методологія та інструментальні засоби автоматизованого конструювання програм, що ґрунтувалися на системах алгоритмічних алгебр (САА) В. М. Глушкова та техніці переписувальних правил. Особливість запропонованої методології полягає у формалізації процесів проектування та синтезу алгоритмів та програм. Алгоритми подаються у вигляді високорівневих схем в САА. Переписувальні правила застосовуються для перетворення програм з метою їх оптимізації за певними критеріями (час виконання, використовувана пам'ять та ін.). Розроблені формальні засоби та програмні інструменти проектування паралельних програм для мультиядерних процесорів [7], графічних прискорювачів Nvidia, що використовують технологію CUDA [6], а також гетерогенних платформ із використанням OpenCL [5]. У даній роботі виконаний подальший розвиток методу та інструментарію розпаралелювання операторів циклу для виконання у гетерогенних середовищах. Застосування методу проілюстроване на розробці паралельної OpenCL програми згортки зображень.

Запропонований у даній статті підхід є близьким до робіт, присвячених синтезу програм на основі специфікацій [9, 10] та автоматизованої генерації OpenCL програм [3, 4, 11–13]. Зокрема, в [3, 4] розглянуті автоматичні розпаралелюючі та оптимізуючі компілятори Par4All та PPCG, що генерують OpenCL код з послідовних програм та ґрунтуються на поліедральній моделі. У роботі [11] запропоновано інструмент OCLoptimizer, який автоматично генерує програмний код для виконання на основному пристрої та оптимізує функції-ядра OpenCL для кожного з обчислювальних пристроїв на основі конфігураційного файлу, наданого користувачем. Конфігураційний файл описує основні характеристики та анотації в ядрах, на основі яких виконуються трансформації. Робота [12] присвячена автоматичній генерації паралельного програмного коду для неоднорідних платформ за допомогою інструментарію REWORK. Інструментарій автоматично визначає ядра в успадкованому C++ коді, та генерує різні версії ядер для покращення C++ застосунків, обираючи найкращу версію на основі вихідного коду та характеристик цільової платформи. В роботі [13] запропоновано інструментарій, що застосовує моделювання мовою UML для специфікації, проектування та генерації застосунків OpenCL. Відмінність методу розпаралелювання, запропонованого у даній статті, від відомих автоматизованих систем паралелізації Par4All [3] та PPCG [4] полягає у можливості обробки обсягів даних, що перевищують обсяг пам'яті графічного прискорювача та можливості залучення декількох прискорювачів одночасно. Особливість підходу також полягає у використанні для автоматизованого проектування паралельних програм специфікацій систем алгоритмічних алгебр, поданих у природно-лінгвістичній формі, що полегшує розуміння алгоритмів і досягнення необхідної якості програм, а також застосуванні методу діалогового конструювання синтаксично правильних програм, що виключає можливість виникнення синтаксичних помилок у процесі проектування схем.

1. Алгебра алгоритмів та проектування паралельних програм для неоднорідних платформ

В основу пропонованого підходу до проектування паралельних програм покладений апарат систем алгоритмічних алгебр та їх модифікацій (САА-М) [6]. САА-М призначені для високорівневого проектування та трансформації послідовних і паралельних алгоритмів, поданих у вигляді схем. На САА-М ґрунтуються розроблені інструментальні засоби автоматизованого проектування та генерації програм [5–7].

Модифіковані САА є двоосновною алгеброю $< Pr, Op; \Omega >$, де Pr – множина логічних умов (предикатів); Op – множина операторів; Ω – сигнатура, що складається з логічних операцій (диз'юнкції, кон'юнкції, заперечення, лівого множення оператора на умову) та операторних операцій (композиції, альтернативи, циклу та ін.), що будуть розглянуті далі. Предикати та оператори можуть бути базисними або складеними. Базисні елементи вважаються атомарними, неподільними абстракціями та пов'язані з предметною областю алгоритму, що проектується. Складені оператори будуються з елементарних за допомогою операцій послідовного й паралельного виконання операторів.

На САА-М ґрунтується алгоритмічна мова САА/1 [6], призначена для багаторівневого структурного проектування й документування послідовних та паралельних алгоритмів і програм. Перевагою її використання є можливість опису алгоритмів у природно-лінгвістичній формі. Подання операторів мовою САА/1 називаються САА-схемами.

Далі наведено перелік назв та специфікації основних операторних операцій сигнатури САА-М, поданих у природно-лінгвістичній формі:

- композиція (послідовне виконання операторів): “*operator* 1”; “*operator* 2”;
- альтернатива (умовний оператор): IF ‘*condition*’ THEN “*operator* 1” ELSE “*operator* 2” END IF;
- цикл типу while: WHILE ‘*condition*’ “*operator*” END OF LOOP;
- цикл типу for: FOR (*counter* FROM *start* TO *fin*) “*operator*” END OF LOOP;
- синхронізатор, що виконує затримку обчислень доти, поки значення умови не стане істинним: WAIT ‘*condition*’.

Розроблений інтегрований інструментарій проектування та синтезу програм (Integrated toolkit for Design and Synthesis of programs, IDS) [6] ґрунтується на використанні розглянутих засобів САА-М та методу діалогового конструювання синтаксично правильних програм (ДСП-методу). На відміну від традиційних синтаксичних аналізаторів, ДСП-метод орієнтований не на пошук і виправлення синтаксичних помилок, а на виключення можливості їх появи в процесі побудови алгоритму. Основна ідея методу полягає у порівневому проектуванні схем зверху вниз за допомогою суперпозиції мовних конструкцій САА-М, які користувач обирає зі списку та додає у дерево алгоритму. На кожному кроці конструювання система надає користувачу на вибір лише ті конструкції, підстановка яких у схему, що проектується, не порушує її синтаксичну правильність. На основі побудованої схеми алгоритму виконується автоматична генерація тексту програми цільовою мовою програмування (Java, C, C++, C for CUDA, OpenCL та ін.). Відображення операцій САА-М у текст мовою програмування подане у вигляді шаблонів і зберігається в базі даних інструментарію.

В роботі [5] виконане розширення САА-М операціями, орієнтованими на паралельні обчислення в гетерогенному середовищі, а саме: призначеними для проектування OpenCL програм. Програмна модель OpenCL [4] дозволяє програмісту описувати функції, які будуть паралельно виконані на деякому прискорювачі або наборі прискорювачів, доступних на даному комп’ютері. Для того, щоб вказати конкретний прискорювач, використовується поняття контексту – структури даних, що задає клас потрібного прискорювача (багатоядерний процесор, графічний процесор, тощо), а також черга команд – структура даних, через яку здійснюється виконання операцій на конкретному прискорювачі. В основі програмної моделі OpenCL покладено поняття ядра – функції, яка буде виконана паралельно на прискорювачі певною кількістю потоків.

Далі наведено базисні оператори алгебри алгоритмів, призначені для проектування OpenCL програм.

1. Отримання списку доступних платформ та запис їх у змінну *pl*:

“Get all available platforms (*pl*)”.

2. Отримання списку пристроїв з одержаної платформи *pl*:

“Get all devices (*dvs*) available on a platform (*pl*)”.

3. Створення контексту виконання для пристроїв:

“Create a context (*cnt*) for devices (*dvs*)”.

4. Створення черги виконання для пристрою *dv*:

“Create a command queue (*cmdqueue*) for a context (*cnt*) and a device (*dv*)”.

5. Компіляція файлу, що містить вихідний код функції-ядра OpenCL для виконання на пристрої:

“Create a kernel (*krnl*) from a source (*program*) for a device (*dv*)”,

де *krnl* – назва функції-ядра; *program* – шлях до файлу з вихідним кодом цієї функції.

6. Створення буферу для даних, зазначених у змінній *var*:

“Create a memory buffer (*buff*) of size (*sz*) for data (*var*) on devices in the context (*cnt*)”.

7. Завантаження буферу для змінної *var* в пам’ять пристрою за допомогою черги:

“Add commands to queue (*cmdqueue*) writing a buffer of size (*sz*) of data (*var*) from host to device”.

8. Установка значення *arg_value* для параметру під номером *arg_index* ядра *krnl*:

“Set the argument value (*arg_value*) for the parameter (*arg_index*) of a kernel (*krnl*)”.

9. Завантаження ядра *krnl* в чергу пристрою та його асинхронне виконання:

“Add a command to queue (*cmdqueue*) executing a kernel (*krnl*)(*workdim*)(*globalworksize*)(*localworksize*) on a device”,

де *workdim* – розмірність простору індексів; *globalworksize* – кількість глобальних задач, які будуть виконувати ядро; *localworksize* – розмірність локальної підмножини задач у групі.

10. Інформація про ідентифікатор задачі для виміру *dim_index* простору індексів:

“Get the global work-item identifier for dimension (*dim_index*)”.

11. Синхронізація (очікування завершення) виконання усіх задач, що були занесені в чергу:

WAIT ‘All previously queued commands in (*cmdqueue*) are issued to the device and have completed’.

12. Зчитування даних з буферу, в який записувався результат:

“Add commands to queue (*cmdqueue*) reading from a buffer of data (*var*) from device to host”.

Приклад використання перелічених конструкцій наведено у розділі 3.

2. Метод та інструментарій автоматизованого розпаралелювання циклів

В даному розділі розглядаються метод та фреймворк LoopRipper, призначені для напівавтоматичного розпаралелювання циклів програм для неоднорідних платформ з пристроями, що використовують OpenCL або CUDA. Інструментарій ґрунтується на спільному використанні системи переписувальних правил TermWare та інструментарію IDS.

LoopRipper виконує розпаралелювання вхідного складеного циклу виду

$$
\begin{aligned}
&\text{“SEQUENTIAL LOOP”}\\
&\text{==== FOR } (i_N \text{ FROM 0 TO } I_N - 1)\\
&\qquad \text{FOR } (i_{N-1} \text{ FROM 0 TO } I_{N-1} - 1)\\
&\qquad\quad \ldots\\
&\qquad\quad \text{FOR } (i_0 \text{ FROM 0 TO } I_0 - 1)\\
&\qquad\quad\quad \text{“}F(\vec{i},\ D)\text{”}\\
&\qquad\quad \text{END OF LOOP}\\
&\qquad \text{END OF LOOP}\\
&\quad \text{END OF LOOP,}
\end{aligned}
\tag{1}
$$

де $I_k \in N$, $0 \le k \le N$; $F : I_0 \times \ldots \times I_N \times D \mapsto D$ – відображення над множиною D; i_k – k-й лічильник циклу (1), $\vec{i} = (i_0, i_1, \ldots, i_N) \in I_0 \times \ldots \times I_N$ – вектор лічильників; $F(\vec{i},\ \cdot)$ – виклик функції F для певного значення $\vec{i}$ ітерації.

Виконаємо наступну підстановку для кожного з операторів FOR:

$$
\begin{aligned}
&\text{FOR } (i_N \text{ FROM 0 TO } I_N - 1)\\
&\quad \ldots\\
&\text{END OF LOOP}
\end{aligned}
\quad \mapsto \quad
\begin{aligned}
&\text{FOR } (s_N \text{ FROM 0 TO } S_N - 1)\\
&\quad \text{FOR } (i_N \text{ FROM } s_N \cdot L(I_N,\ S_N) \text{ TO}\\
&\qquad \min((s_N + 1) \cdot L(I_N,\ S_N),\ S_N) - 1)\\
&\quad\quad \ldots\\
&\quad \text{END OF LOOP}\\
&\text{END OF LOOP,}
\end{aligned}
$$

де

$$L(a, b) = \left\lfloor \frac{a}{b} \right\rfloor + 1 - \delta_{0,\, a \bmod b},$$

$\lfloor \cdot / \cdot \rfloor$ – частка від ділення; δ – дельта Кронекера; S_n – кількість частин, на які потрібно розділити кожний цикл, $1 \le S_n \le I_n$.

Після підстановки новий цикл матиме вигляд

“PARALLEL LOOP”
====
FOR (s_N FROM 0 TO $S_N - 1$)
  ...
  FOR (s_0 FROM 0 TO $S_0 - 1$)
    FOR (i_N FROM $s_N \cdot L(I_N,\ S_N)$ TO $\min((s_N + 1) \cdot L(I_N,\ S_N), S_N) - 1$)
      ...
      FOR (i_0 FROM $s_0 \cdot L(I_0,\ S_0)$ TO $\min((s_0 + 1) \cdot L(I_0,\ S_0), S_0) - 1$)
        “$F(\vec{i},\ D)$”
      END OF LOOP
    END OF LOOP
  END OF LOOP
END OF LOOP. (2)

Внутрішній цикл подібний до циклу (1), але має менший розмір. Залишаючи внутрішні $N+1$ цикли, згрупуємо перші $N+1$ циклів:

$$\begin{aligned}&\text{FOR } (e \text{ FROM 0 TO } \prod_{i=0}^{N} S_i - 1)\\ &\quad \vec{i} := g(e);\\ &\text{END OF LOOP,}\end{aligned} \tag{3}$$

де $g(\cdot)$ – відображення, яке відновлює вектор літераторів $\vec{i}$ і будується таким чином:

$$g_0(e) := e \bmod S_0,$$

$$g_k(e) := \left\lfloor \left(e - \sum_{j=k+1}^{N} g_j(e) \prod_{l=0}^{j-1} S_l \right) \Big/ \prod_{j=0}^{k-1} S_j \right\rfloor,\ 0 < k < N,$$

$$g_N(e) := \left\lfloor e \Big/ \prod_{j=0}^{N-1} S_j \right\rfloor,$$

$$0 \le e \le \prod_{k=0}^{N} S_k\ .$$

Цикл (3) залишає послідовність у векторі лічильників рівною послідовності, яка продукується циклом (1).

Позначимо $kernel(e)$ внутрішні $N+1$ циклів схеми (2) разом з $g(e)$. Передбачається, що $kernel(e)$ буде виконуватися на графічному прискорювачі. Оскільки пам’ять графічного прискорювача ізольована від пам’яті головного пристрою (центрального процесора), ми вводимо операцію *serialize*, яка готує вхідні дані, необхідні для виконання обчислень на кроці e і операцію *deserialize* для зберігання вихідних даних, оброблених прискорювачем. Подальша реалізація цих процедур залежить від конкретної задачі. Нарешті, маємо

FOR (e FROM 0 TO $\prod_{i=0}^{N} S_i - 1$)
  “*serialize*(*e*, *inputData*, *dataPull*)”;
  “*transfer2device*(*inputData*)”;
  “*kernel*(*e*, *inputData*, *outputData*)”;
  “*transfer2host*(*outputData*)”;
  “*deserialize*(*e*, *outputData*, *dataPull*)”;
END OF LOOP. (4)

Ітерації циклу (4) можуть бути розподілені між паралельними потоками за допомогою використання кількох додаткових буферів обміну даними. Цей підхід може бути застосований для будь-якої системи з розподіленою пам'яттю, наприклад, кластера з графічними прискорювачами. Для зберігання еквівалентності у сенсі однаковості результатів для тих самих вхідних даних, мають виконуватися умови Бернстайна [14]. Це означає, що ітерації не повинні перезаписувати вхідні дані інших ітерацій та повинні зберігати свої вихідні дані окремо. Множина S_k $(0 \le k \le N)$ містить параметри трансформації, які обираються таким чином, щоб задовольняти умовам Бернстайна й оптимізувати час обробки, тобто знайти компроміс між часом, що витрачається на підготовку даних, їх передачу та виконання ядра. Час згаданих операцій залежить від розміру вхідних та вихідних даних, який обмежується наявним обсягом пам'яті прискорювача та конфігураційних параметрів апаратного забезпечення, таких, як швидкість передачі даних у пам'яті та обчислювальні можливості прискорювача.

Запропонований метод був реалізований в експериментальному інструменті напівавтоматичного розпаралелювання LoopRipper, що використовує систему переписувальних правил TermWare [8] та інструментарій IDS [6].

TermWare є системою символьних обчислень, що надає мову для опису переписувальних правил, які оперують структурами даних, що називаються термами, та інтерпретатор правил для трансформації термів. Терми є деревоподібними структурами виду $f(t_1,t_2,...,t_n)$. Правила задаються у вигляді $f(x_1,x_2,...,x_n) \to g(x_1,x_2,...,x_n)$, де f – вхідний терм, g – вихідний терм, $x_1,x_2,...,x_n$ – змінні. Вихідний код програми, що потрібно трансформувати, подається у вигляді абстрактного синтаксичного дерева, яке перетворюється за допомогою правил. Дерево будується синтаксичним аналізатором TermWare з вихідного коду. Система також містить генератор для зворотного перекладу з моделі програми у вигляді термів у мову програмування.

Користувач системи LoopRipper (рис. 1) надає вихідний код або САА-схему послідовної програми, вказує цикл, який потрібно розпаралелити, а також список вхідних та вихідних параметрів, що використовуються у циклі. За допомогою системи TermWare інструментарій генерує функції *kernel*, *serialize* та *deserialize* і буфери даних. Генерація функцій здійснюється шляхом заміни параметрів на вхідні та вихідні буфери даних та перерозрахунку ітераторів у початковому циклі. Система IDS виконує заміну послідовного циклу програми на відповідний виклик ядра та синтезує паралельну програму зі згаданих функцій та структур даних.

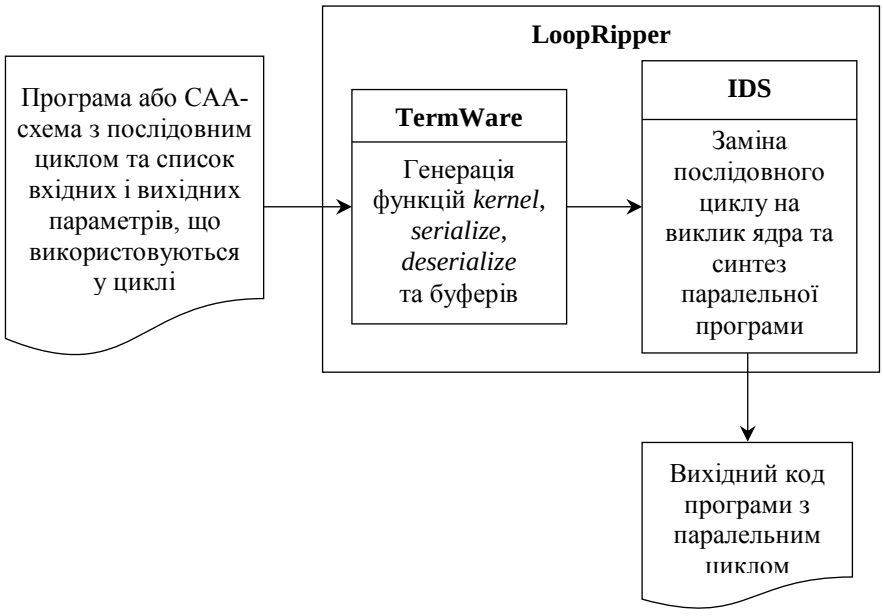

Рис. 1. Розпаралелювання програми в інструменті LoopRipper

Розглянемо далі послідовність виконання у програмі, розпаралеленій за допомогою запропонованого методу. Програма виконується на обчислювальному вузлі, що містить один багатоядерний центральний процесор та один графічний прискорювач. Сучасні графічні прискорювачі підтримують прямий доступ до пам'яті і таким чином дозволяють здійснювати передачу даних і виконання ядра асинхронно. Для оптимізації процесу обміну даними, використовується подвійна буферизація – по два буфери для обміну вхідними й вихідними даними. На головному пристрої обчислювання здійснюються двома потоками, що одночасно виконують функції *kernel*, *serialize* та *deserialize*. Один із них серіалізує вхідні дані та заповнює буфер вхідних даних, а потім передає буфер до графічного прискорювача й запускає ядро, а інший отримує буфер

вихідних даних з прискорювача й десеріалізує їх. Окрім обчислень, прискорювач виконує двосторонній обмін даними за допомогою механізму асинхронного обміну. Обчислення складаються з трьох етапів – початкового, циклічного та завершального. На початковому етапі буфери даних пусті й прискорювач очікує на передачу даних. Немає значення, який потік буде виконувати початковий крок, оскільки всі операції виконуються послідовно й режим асинхронної передачі даних не використовується. Центральний процесор серіалізує буфери вхідних даних перших двох ітерацій та передає буфер, що містить дані першої ітерації, прискорювачу. Після початкового починається циклічний етап. Послідовність дій наведено на рис. 2. Номер ітерації вказано після імені буфера. Один крок циклу розділяється на парні й непарні частини. Тим часом прискорювач виконує обчислення з поточним буфером, потоки головного пристрою беруть дані з буфера з попереднього кроку, десеріалізують буфер передостаннього кроку, надсилають буфер вхідних даних та серіалізують буфер для наступного кроку. За один крок виконується запуск двох ядер. Після того, як кожна частина (парна або непарна) завершена, процеси синхронізуються. На завершальному етапі виконується десеріалізація буферу вихідних даних, переданого на останньому кроці циклу, після чого отримується й десеріалізується буфер остаточних вихідних даних.

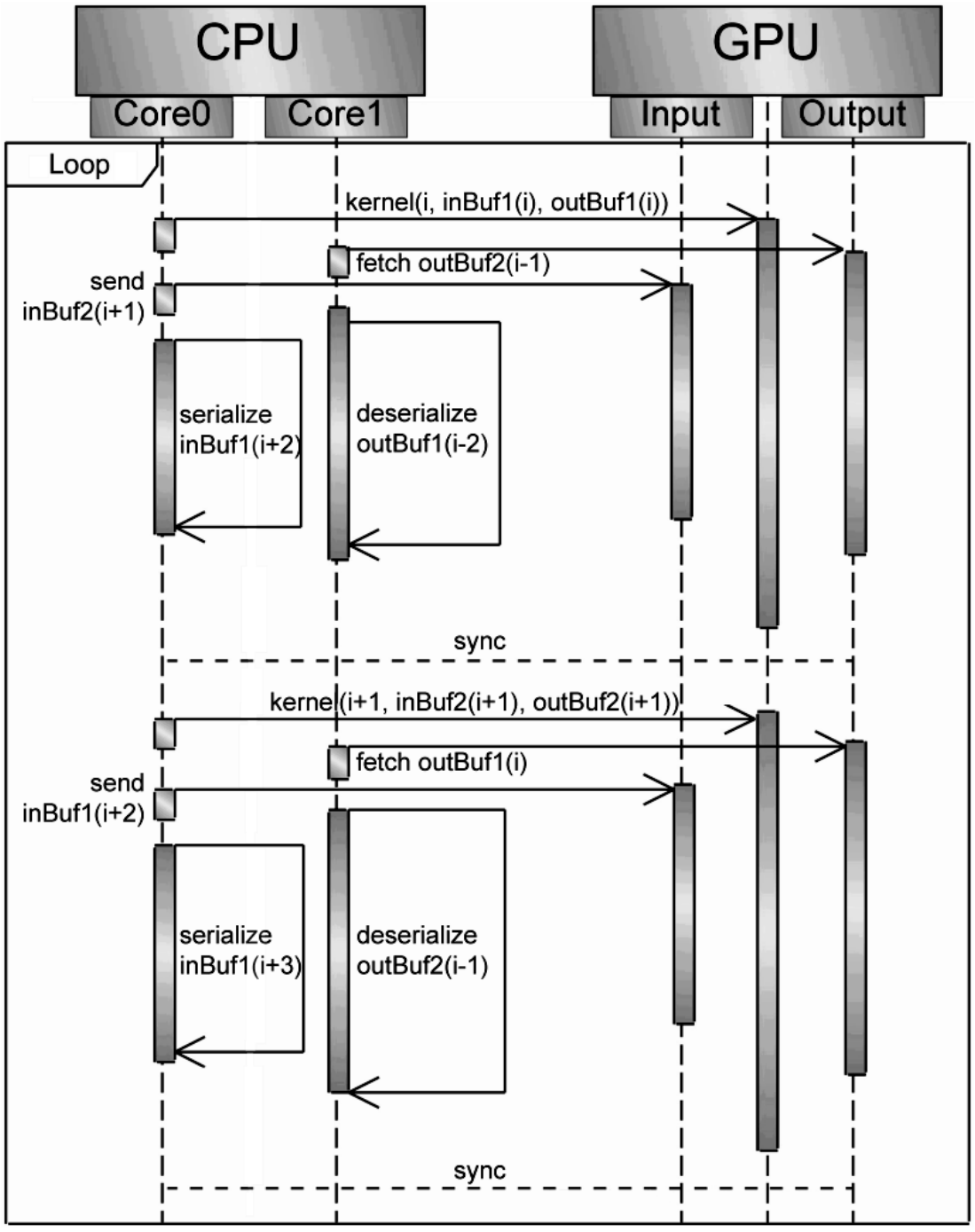

Рис. 2. Діаграма послідовності для циклічного етапу згенерованої паралельної програми для системи з одним прискорювачем, двома потоками та чотирма буферами обміну

3. Застосування алгебро-алгоритмічного інструментарію та методу розпаралелювання для розробки програми згортки зображень

В даному розділі використання систем IDS та LoopRipper проілюстроване на прикладі розробки паралельної OpenCL програми згортки зображень [15]. Згортка виконує фільтрацію значень пікселів в зображенні, яка може використовуватися для розмиття, підвищення різкості, виділення границь зображення та інших удосконалень, що ґрунтуються на ядрі фільтрації.

У загальному вигляді згортка подається формулою

$$g(x,y) = w * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t),$$

де $g(x,y)$ – зображення, отримане у результаті застосування фільтру; $f(x,y)$ – початкове зображення; w – ядро фільтрації.

САА-схема початкового послідовного алгоритму програми згортки зображень, побудована в системі IDS, наведена нижче. Алгоритм виконує згортку вхідного зображення *src* розміром $width \times height$ на основі матриці-ядра *kern* розміром $kernel\_size \times kernel\_size$ і зберігає результат у рисунок *dst*. Функція $\text{clamp}(val, \_min, \_max)$ обрізає значення *val* таким чином, щоб воно було в діапазоні $[\_min, \_max]$. Функція $\text{coord}(i, j, width, channel)$ обчислює індекс в одновимірному масиві/буфері зображення на основі координат пікселя і каналу (R/G/B), тобто повертає індекс елемента $(i, j, channel)$ в тензорі $(width, height, 3)$. Схема містить чотири вкладені цикли. Ітерації перших двох зовнішніх циклів з лічильниками $0 \le j < height$ та $0 \le i < width$ є незалежними і можуть бути виконані паралельно.

SCHEME CONVOLUTION SEQUENTIAL ====

“convolution sequential(*src*, *dst*, *width*, *height*, *kern*, *kernel_size*)”
==== FOR (*j* FROM 0 TO *height*–1)
  FOR (*i* FROM 0 TO *width*–1)
    “Declare a variable (*resultR*) of type (*char*) = (0)”;
    “Declare a variable (*resultG*) of type (*char*) = (0)”;
    “Declare a variable (*resultB*) of type (*char*) = (0)”;
    “Declare a variable (*radius*) of type (*unsigned char*) = (*kernel_size* / 2)”;
    FOR (*kj* FROM 0 TO *kernel_size*)
      FOR (*ki* FROM 0 TO *kernel_size*)
        “Declare the list of variables (*kernel_el*, *coordI*, *coordJ*) of type (*int*)”;
        (*kernel_el* := *ki* * *kj* * *kernel_size*);
        (*coordI* := “clamp(*i* + *ki* – *radius*, 0, *width*)”);
        (*coordI* := “clamp(*j* + *kj* – *radius*, 0, *height*)”);
        “Increase (*resultR*) by (*src*[*coord*(*coordI*, *coordJ*, *width*, 0)] * *kern*[*kernel_el*])”;
        “Increase (*resultG*) by (*src*[*coord*(*coordI*, *coordJ*, *width*, 1)] * *kern*[*kernel_el*])”;
        “Increase (*resultB*) by (*src*[*coord*(*coordI*, *coordJ*, *width*, 2)] * *kern*[*kernel_el*])”
      END OF LOOP
    END OF LOOP;
    (*dst*[*coord*(*i*, *j*, *width*, 0)] := *resultR*);
    (*dst*[*coord*(*i*, *j*, *width*, 1)] := *resultG*);
    (*dst*[*coord*(*i*, *j*, *width*, 2)] := *resultB*)
  END OF LOOP
END OF LOOP;

“clamp(*val*, *_min*, *_max*)”
==== “Return value (min(max(*val*, *_min*), *_max*))”;

“coord(*i*, *j*, *width*, *channel*)”
==== “Return value (3 * (*i* + *j* * *width*) + *channel*)”;

END OF SCHEME CONVOLUTION SEQUENTIAL

САА-схеми головної частини паралельної програми (convolution host) та функції-ядра (convolution kernel) для виконання у гетерогенному середовищі, отримані у результаті розпаралелення послідовної програми за допомогою інструментарію LoopRipper, наведені далі. У порівнянні з послідовною програмою, функція-ядро містить лише два вкладені цикли. Значення змінних *i* та *j* обчислюються за допомогою виклику базисного оператора "Get the global work-item identifier for dimension (*dim_index*)", який повертає ідентифікатор задачі.

SCHEME CONVOLUTION HOST ====

"convolution host"
==== "Get all available platforms (*platforms*)";
"Get all devices (*dev*) available on a platform (*platform*[0])";
"Create a context (*cnt*) for devices (*dev*)";
"Create a command queue (*commandQueue*) for a context (*cnt*) and a device (*dev*)";
"Create a kernel (*kernel*) from a source (*convolution*) for a device (*dev*)";
"Declare a variable (*totalSize*) of type (*size_t*)";
(*totalSize* := 3 * "Get image height (*image*)" * "Get image width (*image*)");
"Declare a 1D continuous array (*res*) of type (*unsigned char*) and size (*totalSize*)";
"Create memory buffers (*srcBuffer*, *dstBuffer*) for data (*res*) on devices in the context (*cnt*)";
"Declare a constant (*kernelSize*) of type (*size_t*) = (5)";
"Declare a constant (*kernelSizeRaw*) of type (*size_t*) = (*kernelSize* * *kernelSize* * sizeof(*float*))";
"Create a memory buffer (*kernelBuffer*) of size (*kernelSizeRaw*) on devices in the context (*cnt*)";
"Declare and fill array (*kernelArg*) of kernel arguments for image convolution problem";
"Add commands to queue (*commandQueue*) writing a buffer (*srcBuffer*) of size (*totalSize*) of data (*image*) from host to device";
"Add commands to queue (*commandQueue*) writing a buffer (*kernelBuffer*) of size (*kernelSizeRaw*) of data (*kernelArg*) from host to device";
"Set the argument values (*srcBuffer*, *dstBuffer*, *image_width*, *image_height*, *kernelBuffer*, *kernelSize*) for the parameters (0, 1, 2, 3, 4, 5) of a kernel (*kernel*)";
"Add a command to queue (*commandQueue*) executing a kernel (*kernel*)(2)(*image_width*)(*image_height*) on a device";
WAIT 'All previously queued commands in (*commandQueue*) are issued to the device and have completed';
"Add commands to queue (*commandQueue*) reading from a buffer (*dstBuffer*) of data (*res*) from device to host";

END OF SCHEME CONVOLUTION HOST

SCHEME CONVOLUTION KERNEL ====

"convolution(*src*, *dst*, *width*, *height*, *kern*, *kernel_size*)"
==== (*i* := "Get the global work-item identifier for dimension (0)");
(*j* := "Get the global work-item identifier for dimension (1)");
"Declare a variable (*resultR*) of type (*char*) = (0)";
"Declare a variable (*resultG*) of type (*char*) = (0)";
"Declare a variable (*resultB*) of type (*char*) = (0)";
"Declare a variable (*radius*) of type (*unsigned char*) = (*kernel_size* / 2)";
FOR (*kj* FROM 0 TO *kernel_size*)
FOR (*ki* FROM 0 TO *kernel_size*)
"Declare the list of variables (*kernel_el*, *coordI*, *coordJ*) of type (*int*)";
(*kernel_el* := *ki* * *kj* * *kernel_size*);
(*coordI* := "clamp(*i* + *ki* – *radius*, 0, *width*)");
(*coordI* := "clamp(*j* + *kj* – *radius*, 0, *height*)");
"Increase (*resultR*) by (*src*[*coord*(*coordI*, *coordJ*, *width*, 0)] * *kern*[*kernel_el*])";
"Increase (*resultG*) by (*src*[*coord*(*coordI*, *coordJ*, *width*, 1)] * *kern*[*kernel_el*])";
"Increase (*resultB*) by (*src*[*coord*(*coordI*, *coordJ*, *width*, 2)] * *kern*[*kernel_el*])"
END OF LOOP
END OF LOOP;
(*dst*[*coord*(*i*, *j*, *width*, 0)] := *resultR*);

(*dst*[*coord*(*i*, *j*, *width*, 1)] := *resultG*);
(*dst*[*coord*(*i*, *j*, *width*, 2)] := *resultB*);

END OF SCHEME CONVOLUTION KERNEL

На основі побудованих САА-схем в системі ІПС виконана генерація паралельної OpenCL програми згортки зображень. Отриману програму було виконано в обчислювальному середовищі, що складалося з 4-ядерного процесора AMD Ryzen 3 2300X CPU та графічного прискорювача Nvidia GeForce GTX 1050 (640 ядер). Для експериментів обрано розмивання гауссівським ядром. На рис. 3 зліва наведене тестове оригінальне зображення, справа – зображення після застосування згортки гауссівським ядром 5×5:

$$\frac{1}{256}\begin{pmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}.$$

(*а*) (*б*)

Рис. 3. Тестове зображення до (*а*) та після (*б*) застосування гауссівського розмиття програмою згортки зображень

Мультипроцесорне прискорення при обробці зображення розміром 512×512 становило 81 раз, що свідчить про гарний ступінь розпаралелення обчислень.

Висновки

Запропоновані методи та інструментальні засоби автоматизованого проектування та генерації OpenCL програм на основі алгебри алгоритмів та переписувальних правил. Розроблено метод напівавтоматичного розпаралелювання циклічних операторів із використанням серіалізації даних. Розглянутий підхід полягає у використанні високорівневих алгебро-алгоритмічних специфікацій програм, що подаються у природно-лінгвістичній формі. Розроблені інструментальні засоби забезпечують автоматизоване проектування схем алгоритмів шляхом суперпозиції конструкцій алгебри Глушкова, що розглядаються як компоненти повторного використання. Інструментарій автоматично виконує генерацію програм цільовою мовою програмування на снові специфікацій. Розроблений програмний засіб паралелізації циклів для оптимізації обчислень за допомогою графічних прискорювачів дозволяє в напівавтоматичному режимі здійснювати перехід від послідовних до паралельних програм. На відміну від відомих подібних засобів розпаралелювання (наприклад, Par4All), створений інструментарій дозволяє обробляти обсяги даних, що перевищують обсяг пам’яті графічного прискорювача та залучати декілька прискорювачів одночасно.

Література

1. OpenCL overview. The open standard for parallel programming of heterogeneous systems. URL: https://www.khronos.org/opencl (дата звернення 21.02.2020 р.)
2. Nvidia CUDA technology. URL: http://www.nvidia.com/cuda (дата звернення 21.02.2020 р.).
3. PIPS: Automatic Parallelizer and Code Transformation Framework. URL: http://pips4u.org (дата звернення 21.02.2020 р.).
4. PPCG: Automatic parallelizing and optimizing compiler. URL: http://freecode.com/projects/ppcg (дата звернення 21.02.2020 р.).
5. Дорошенко А.Ю., Бондаренко М.М., Яценко О.А. Автоматизоване проектування OpenCL програм на основі алгебро-алгоритмічного підходу. *Проблеми програмування*. 2019. № 1. С. 27–36.
6. Algebra-algorithmic models and methods of parallel programming / Andon P.I., Doroshenko A.Yu., Zhereb K.A., Yatsenko O.A. Kyiv : Akademperiodyka, 2018. 192 p.
7. Doroshenko A., Zhereb K., Yatsenko O. Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools. *Communications in computer and information science. Information and communication technologies in education, research, and industrial applications*. 2013. Vol. 412. P. 70–92.
8. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. *Fundamenta informaticae*. 2006. Vol. 72, N 1–3. P. 95–108.
9. Flener P. Achievements and prospects of program synthesis. *Computational logic: logic programming and beyond. Essays in honour of Robert A. Kowalski*. Part I. London, 2002. P. 310–346.
10. Gulwani S. Dimensions in program synthesis. *Proc. 12th Int. ACM SIGPLAN Symposium on principles and practice of declarative programming*, Hagenberg, Austria (26–28 July 2010). New York : ACM, 2010. P. 13–24.
11. Automatic generation of optimized OpenCL codes using OCLoptimizer / Fabeiro J. F., Andrade D., Fraguela B. B., Doallo R. *The computer journal*. 2015. Vol. 58, N 11. P. 3057–3073.
12. Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications / Sotomayor R. et al. *International journal of parallel programming*. 2017. Vol. 45, N 2. P. 262–282.
13. Rodrigues A., Guyomarc'h F., Dekeyser J. L. An MDE approach for automatic code generation from UML/MARTE to OpenCL. *Computing in science and engineering*. 2012. Vol. 15, N 1. P. 46–55.
14. Bernstein A. J. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*. 1966. Vol. EC-15, N 5. P. 757–763.
15. Novak J., Liktor G., Dachsbacher C. GPU computing: image convolution. URL: https://cg.ivd.kit.edu/downloads/GPUComputing_assignment_3.pdf (дата звернення 21.02.2020 р.).

References

1. OpenCL overview. The open standard for parallel programming of heterogeneous systems. [Online] Available from: https://www.khronos.org/opencl [Accessed: 21 February 2020]
2. Nvidia CUDA technology. [Online] Available from: http://www.nvidia.com/cuda [Accessed: 21 February 2020]
3. PIPS: Automatic Parallelizer and Code Transformation Framework. [Online] Available from: http://pips4u.org [Accessed: 21 February 2020]
4. PPCG: Automatic parallelizing and optimizing compiler. [Online] Available from: http://freecode.com/projects/ppcg [Accessed: 21 February 2020]
5. Doroshenko, A.Yu., Bondarenko, M.M. & Yatsenko, O.A. (2019) Automated design of OpenCL programs based on algebra-algorithmic approach. *Problems in programming*. (1). P. 27–36.
6. Andon, P.I., Doroshenko, A.Yu., Zhereb, K.A. & Yatsenko, O.A. (2018) *Algebra-algorithmic models and methods of parallel programming*. Kyiv: Akademperiodyka.
7. Doroshenko, A., Zhereb, K. & Yatsenko, O. (2013) Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools. *Communications in computer and information science. Information and communication technologies in education, research, and industrial applications*. 412. P. 70–92.
8. Doroshenko, A. & Shevchenko, R. (2006) A rewriting framework for rule-based programming dynamic applications. *Fundamenta informaticae*. 72 (1-3). P. 95–108.
9. Flener, P. (2002) Achievements and prospects of program synthesis. *Computational logic: logic programming and beyond. Essays in honour of Robert A. Kowalski*. Part I. London. P. 310–346.
10. Gulwani, S. (2010) Dimensions in program synthesis. *Proc. 12th Int. ACM SIGPLAN symposium on principles and practice of declarative programming*. Hagenberg, Austria, 26-28 July 2010. New York: ACM. P. 13–24.
11. Fabeiro, J.F., Andrade, D., Fraguela, B.B. & Doallo R. (2015) Automatic generation of optimized OpenCL codes using OCLoptimizer. *The computer journal*. 58 (11). P. 3057–3073.
12. Sotomayor, R. et al. (2017) Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. *International journal of parallel programming*. 45 (2). P. 262–282.
13. Rodrigues, A., Guyomarc'h, F. & Dekeyser, J.L. (2012) An MDE approach for automatic code generation from UML/MARTE to OpenCL. *Computing in science and engineering*. 15 (1). P. 46–55.
14. Bernstein, A.J. (1966) Analysis of programs for parallel processing. *IEEE transactions on electronic computers*. EC-15 (5). P. 757–763.
15. Novak, J., Liktor, G. & Dachsbacher, C. GPU computing: image convolution. [Online] Available from: https://cg.ivd.kit.edu/downloads/GPUComputing_assignment_3.pdf [Accessed: 21 February 2020]

Одержано 26.02.2020

***Про авторів*:**

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук, професор, завідувач відділу,
професор кафедри автоматики і управління в технічних системах
НТУУ "КПІ імені Ігоря Сікорського".

Кількість наукових публікацій в українських виданнях – понад 180.
Кількість наукових публікацій в зарубіжних виданнях – понад 60.
Індекс Хірша – 6.
http://orcid.org/0000-0002-8435-1451,

Бекетов Олексій Геннадійович,
молодший науковий співробітник.
Кількість наукових публікацій в українських виданнях – 12.
Кількість наукових публікацій в зарубіжних виданнях – 2.
http://orcid.org/0000-0003-4715-5053,

Бондаренко Микола Миколайович,
аспірант.
http://orcid.org/0000-0002-6362-414X,

Яценко Олена Анатоліївна,
кандидат фізико-математичних наук, старший науковий співробітник.
Кількість наукових публікацій в українських виданнях – 47.
Кількість наукових публікацій в зарубіжних виданнях – 16.
http://orcid.org/0000-0002-4700-6704.

Місце роботи авторів:

Інститут програмних систем НАН України,
03187, м. Київ-187, проспект Академіка Глушкова, 40.
Тел.: (044) 526 3559.
E-mail: doroshenkoanatoliy2@gmail.com,
beketov.oleksii@gmail.com,
bondarenko_mykola@yahoo.com.ua,
oayat@ukr.net