УДК 004.932.2:004.383.5

Алгоритми пошуку фрагментів зображення для відстежування об'єктів у відеопотоці на паралельній архітектурі CUDA

Ладиженський Ю.В., Середа А.О.
Донецький національний технічний університет,
ly@cs.dgtu.donetsk.ua, aas11@bk.ru

Abstract

Ladyzhenskyy Y., Sereda A. The image fragments searching algorithms for object tracking in a video stream on CUDA parallel architecture. *The new algorithms for searching of arbitrary shape image fragments on the CUDA architecture are proposed. The realization of proposed algorithms on a GPU proved significant speed improvement over the realization on CPU.*
Keywords: object tracking in video, image search, template matching, parallel algorithm, CUDA

Вступ

Мета відстежування об'єктів у відеопотоці – отримати множину видимих об'єктів і їх координат у кожному кадрі. Воно використовується в багатьох областях діяльності, зокрема, в системах безпеки та автоматизації аналізу спортивних змагань. Розроблені різні методи та алгоритми відстежування об'єктів.

У [1] об'єкт представлений його зображенням і матрицею, що задає ймовірність приналежності до об'єкту кожного пікселя цього зображення. При відстежуванні відбувається пошук цього зображення в кадрі.

У [2] описано розроблений авторами статті метод відстежування об'єктів у відеопотоці на основі відстежування переміщення фрагментів об'єктів. Кожен фрагмент об'єкту є областю кадру довільного розміру і форми, що містить частину об'єкту або цілий об'єкт. Представлення фрагмента об'єкту схоже з представленням об'єкту в попередньому алгоритмі.

Пошук фрагменту зображення в кадрі може займати багато часу і потребує прискорення. Пошук складається з простих операцій, що повторюються над елементами матриць і може бути ефективно виконаний на паралельній обчислювальній архітектурі. Можливо використовувати паралельну архітектуру загального призначення або архітектури, спеціально розроблені для пошуку фрагментів зображень в кадрі, подібно до описаної у [3].

Поширеними, доступними і достатньо універсальними пристроями, придатними для паралельної обробки великих об'ємів даних, є відеокарти NVIDIA з підтримкою технології CUDA [4]. У даній статті розглянуто нові алгоритми пошуку фрагментів зображень для архітектури CUDA.

Постановка задачі

Нехай фрагмент зображення, який шукають, вписано у прямокутник завширшки W та заввишки H. Нехай F – матриця розміром $H \times W$, що містить пікселі шаблону. Кожен піксель шаблону $f_{y,x}$ ($y = \overline{0,H-1}, x = \overline{0,W-1}$) характеризується наступними властивостями: $f^r_{y,x}$, $f^g_{y,x}$, $f^b_{y,x}$ – відповідно красна, зелена та синя компоненти кольору; $f^m_{y,x} \in [0,1]$ – оцінка приналежності пікселя з координатами (x,y) до фрагменту. Значення $f^m_{y,x}$ може бути чисельно не рівним ймовірності, але зі збільшенням ймовірності монотонно зростає.

Назвемо областю пошуку прямокутну область кадру, що містить пікселі шаблону при всіх його можливих положеннях. Нехай її висота дорівнює H_0, ширина – W_0, а C – матриця розміром $H_0 \times W_0$, що містить її пікселі. Кожен піксель області пошуку $c_{y,x}$ складається з $c^r_{y,x}$, $c^g_{y,x}$, $c^b_{y,x}$ – відповідно красної, зеленої та синьої компонент кольору.

Нехай осі системи координат кадру і шаблону спрямовані вправо і вниз, а піксель із координатами (0,0) розташовано в лівому верхньому куті.

Міру різниці між пікселем шаблону $f_{y,x}$ і пікселем області кадру $c_{y',x'}$ визначимо як

$$\left\| f_{x,y}, c_{y',x'} \right\| = \quad (1)$$
$$f^m_{x,y}\left(\left| f^r_{x,y} - c^r_{y',x'} \right| + \left| f^g_{dx,dy} - c^g_{y',x'} \right| + \left| f^b_{dx,dy} - c^b_{y',x'} \right| \right).$$

Накладемо шаблон на кадр без повороту так, що піксель шаблону з координатами $(0,0)$ співпаде з пікселем кадру з координатами (x,y). Міру різниці між шаблоном і областю кадру під ним визначимо як

$$D(x,y)=\sum_{dx=0}^{W-1}\sum_{dy=0}^{H-1}\left\|f_{dx,dy},c_{x+dx,y+dy}\right\|. \qquad (2)$$

Нехай $S=\{(x_i,y_i)\}$ – множина всіх можливих координат лівого верхнього пікселя шаблону в системі координат області пошуку, причому $0\le x_i\le W_s$, $0\le y_i\le H_s$, $W_s=W_0-W+1$, $H_s=H_0-H+1$.

Найкращу позицію шаблону визначимо як $D_{best}=D(x_{best},y_{best})$ і $(x_{best},y_{best})=\arg\min_{(x,y)\in S}D(x,y)$. У методі відстежування [2] для визначення ймовірності успішного відстежування фрагмента зображення об'єкту необхідно також визначати альтернативну найкращу позицію фрагменту, тобто кращу з позицій, що залишилися у області пошуку, з якої вилучено околицю кращої позиції: $D_{alt}=\min_{(x,y)\in S,\min(|x-x_{best}|,|y-y_{best}|)\ge d}D(x,y)$, де d – константа, що визначає розмір околиці, яка вилучається.

Для визначення (x_{best},y_{best}) і D_{best} можна використовувати як повний перебір всіх можливих положень шаблону, так і швидкі алгоритми пошуку, засновані на монотонному зменшенні міри різниці між шаблоном і кадром при наближенні до оптимального рішення, такі, як Three-Step Search [3]. Для визначення D_{alt} доцільно використовувати алгоритм повного перебору. Пошук шаблонів повним перебором є найбільш ресурсоємним етапом відстежування об'єктів у методі [2] і потребує прискорення. Далі розглядається його реалізації на архітектурі CUDA.

Архітектура CUDA

У даній статті розглядаються пристрої з compute compatibility 1.1, як найпоширеніші на момент написання. Спрощена архітектура CUDA [5] наведена на рис. 1.

Для мінімізації обміну даними між пам'яттю пристрою та пам'яттю кожного мультипроцесора, дані, що багато разів використовуються, доцільно читати один раз і поміщати в текстурний кеш або спільну пам'ять. Для завантаження кожного пікселю F і C із пам'яті пристрою мінімальне число раз, а також для усунення необхідності синхронізації і обміну даними між різними мультипроцесорами, доцільно проводити пошук одного шаблону на одному мультипроцесорі, тобто використовувати для цього один блок потоків.

На сучасних пристроях цілочисельні арифметичні операції виконуються приблизно в 4 рази повільніше, ніж операції з плаваючою крапкою [5]. Доцільно зберігати в пам'яті колір пікселів як цілі однобайтні числа, а перед обчисленнями перетворювати їх в дійсні числа.

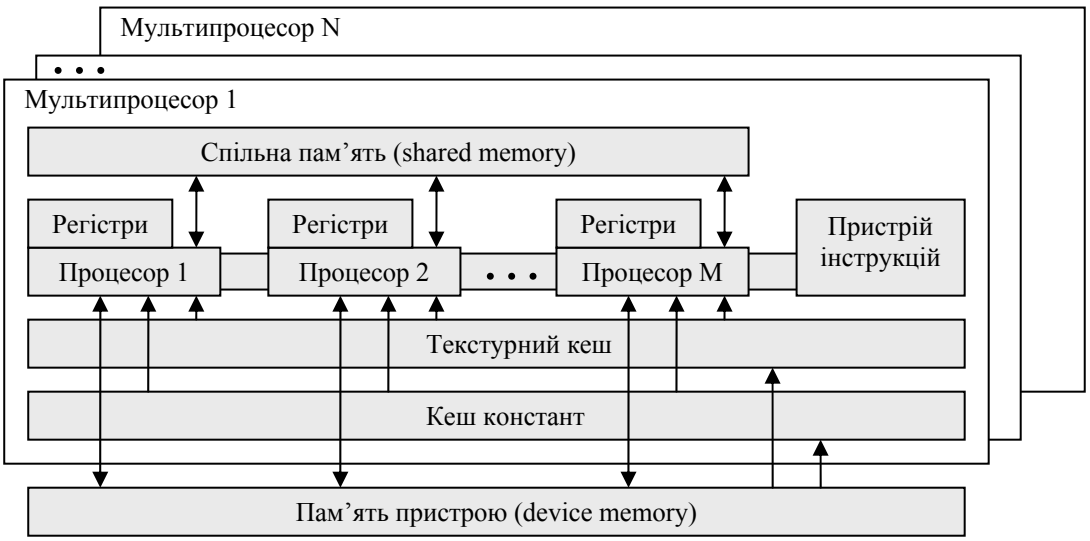

Рисунок 1. – Спрощена архітектура CUDA

При зверненні до пам'яті пристрою без використовування текстурного кешу бажано проводити з’єднані (coalesced) запити до пам'яті пристрою. Для цього 16 потоків, що виконуються одночасно, повинні звертатися до 16-ти чотирьохбайтних слів, розташованих в пам'яті послідовно з адреси, кратної 64 (деякі потоки можуть не брати участь).

Необхідність обчислювати D_{alt} спричиняє необхідність зберігати знайдені значення $D(x,y)$ для $(x,y)\in S$. Їх об'єм може бути завеликий для зберігання в пам'яті мультипроцесора, тому вони зберігаються в пам'яті пристрою. Щоб набувати готові значення (2) в регістрах, кожне значення D повинне обчислюватися одним потоком.

Зручно, щоб група з 16 потоків обчислювала 16 значень *D*, розташованих в пам'яті послідовно. Якщо потоків менше, ніж $H_s W_s$, то деякі потоки можуть обчислювати і зберігати більше одного значення D.

При запропонованій організації обчислень прискорення пошуку множини шаблонів досягається як за рахунок одночасного пошуку різних шаблонів на різних мультипроцесорах, так і за рахунок одночасного обчислення значень (2) різними потоками для різних позицій шаблону.

Алгоритм пошуку фрагмента

Будемо зберігати фрагмент зображення у буфері *Buf* у спільній пам'яті розміром S_{max}, а область пошуку – у текстурному кеші. Доступу до пікселів строки фрагменту у *Buf* здійснюватимемо як до елементів одномірного масиву.

Нехай є *M* потоків і *i*-й потік ($i = \overline{0, M-1}$) обробляє множину позицій шаблону $S^i = \{((x + i + jM) \bmod W_s, (y + i + jM) \operatorname{div} W_s)\}$ для $j = \overline{0, (H_s W_s + M - 1 - i) \operatorname{div} M}$, де div позначає цілочисельне ділення, а mod – залишок від ділення. *i*-й обчислює і зберігає в пам'яті пристрою $D(x, y)$ для всіх $(x, y) \in S^i$. Елементи масиву *D* розташовано в пам'яті послідовно по рядках без дір з адреси, кратної 64. На рис. 2 показані три послідовні етапи пошуку шаблону розміром 3x3 пікселя в області пошуку розміром 7x11 пікселів з використанням 16 потоків. Заштрихованими квадратиками показані пікселі, з якими на кожному кроці співпадає лівий верхній кут шаблону, а сірим – вже оброблені пікселі.

Рисунок 2. – Етапи обробки області пошуку

В процесі обчислення $D(x, y)_{(x,y) \in S^i}$, кожен потік обчислює і зберігає в регістрах значення $(x^i_{best}, y^i_{best}) = \arg \min_{(x,y) \in S^i} D(x, y)$ і $D^i_{best} = D(x^i_{best}, y^i_{best})$.

Після завершення обчислення $D(x, y)_{(x,y) \in S^i}$, а також x^i_{best}, y^i_{best} і D^i_{best}, для $i = \overline{0, M-1}$, визначається такий номер потоку i_{best}, що $\forall_{i \in [0, M-1]} (D^i_{best} > D^{i_{best}}_{best}) \vee ((D^i_{best} = D^{i_{best}}_{best}) \wedge (i \geq i_{best}))$. Потік i_{best} зберігає в змінних в спільній пам'яті значення $x_{best} = x^{i_{best}}_{best}$, $y_{best} = y^{i_{best}}_{best}$, $D_{best} = D^{i_{best}}_{best}$.

Після обчислення D_{best} кожен потік обчислює $D^i_{alt} = \min_{(x,y) \in S^i_{alt}} D(x, y)$, де $S^i_{alt} = S^i \setminus \{(x, y) : \min(|x - x_{best}|, |y - y_{best}|) < d\}$. При цьому використовуються раніше збережені в пам'яті пристрою значення *D*. D_{alt} вибирається з $\{D^i_{alt}\}$ так само, як на попередньому кроці D_{best} з $\{D^i_{best}\}$.

D_{best} і D_{alt} діляться на норму $\sum_{x=0}^{W-1} \sum_{y=0}^{H-1} f^m_{dx,dy}$.

При $(W \times H > S_{\max})$ фрагмент завантажується у *Buf* цілком. Алгоритм обчислення x^i_{best}, y^i_{best}, D^i_{best} і масиву значень *D* приведено на рис. 3, де *Fragment* – масив у пам'яті пристрою, що містить фрагмент.

Якщо $(W \times H > S_{\max})$, пропонується розбити фрагмент по рядках на мінімально можливе число частин, площа кожної з яких не перевищую $S_{\max}$ (рис. 4). Частини завантажуються у *Buf* і обробляються по черзі. Спочатку всі потоки обчислюють часткові суми *D* для першої частини фрагмента, потім – додають до них часткові суми для другої частини, і т.д. Алгоритм обчислення x^i_{best}, y^i_{best}, D^i_{best} і масиву значень *D* приведено на рис. 5.

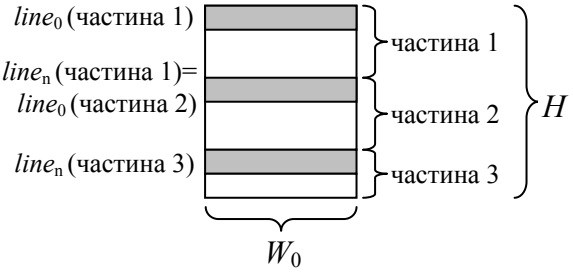

Рисунок 4. – Розбиття фрагменту по рядках

завантажити Fragment у Buf;
синхронізація потоків; // дочекатися завершення завантаження
$D^i_{best} := \infty;$
$p := i;$ // підготовка до обчислень
поки $p < W_s H_s$ // цикл по позиціях фрагменту
$dx := p \bmod W_s;\quad dy := p \operatorname{div} W_s;$ // визначення координат фрагменту
$s := \sum_{y=0}^{H-1}\sum_{x=0}^{W-1} \| c[dy+y, dx+x], Buf[yW+x] \|;$ // обчислення $D(dx, dy)$
$D[p] := s;$ // збереження результату
якщо $s < D^i_{best}$ // перевірка найкращого результату у потоці
то $D^i_{best} := s;\quad x^i_{best} := dx;\quad y^i_{best} := dy;$
$p := p + M;$ // перехід до наступної групи позицій
обчислити $x_{best}, y_{best}, D_{best}, D_{alt}$;

Рисунок 3 – Алгоритм пошуку фрагменту

$D^i_{best} := \infty;$
$line_0 := 0;$ // перший рядок фрагменту
поки $line_0 < H$ // цикл по частині фрагменту
$line_n := \min(H, line_0 + S_{\max} \operatorname{div} W);$ // останній рядок фрагменту (не включно)
$p := i - \left((адреса\ Fragment + line_0 W) \operatorname{div} 4\right) \bmod 16;$ // зсув для забезпечення з'єднаних запитів
поки $p < (line_n - line_0)W$ // завантаження частини фрагменту у спільну пам'ять
якщо $p \geq 0$
то $Buf[p] := Fragment[p];$ // копіювання даних
$p := p + M;$
синхронізація потоків; // дочекатися завершення завантаження
$p := i;$
поки $p < W_s H_s$ // цикл по позиціях фрагменту
$dx := p \bmod W_s;\quad dy := p \operatorname{div} W_s;$ // визначення координат фрагменту
якщо $line_0 = 0$ // отримання попереднього значення часткової суми
то $s := 0;$
інакше $s := D[p];$
$s := s + \sum_{y=line_0}^{line_n-1}\sum_{x=0}^{W-1} \| c[dy+y, dx+x], Buf[(y-line_0)W+x] \|$ // обчислення $D(dx, dy)$
$D[p] := s;$ // збереження результату
якщо $(line_n = H) \wedge (s < D^i_{best})$ // перевірка найкращого результату у потоці
то $D^i_{best} := s;\quad x^i_{best} := dx;\quad y^i_{best} := dy;$
$p := p + M;$ // перехід до наступної групи позицій
синхронізація потоків; // перед повторним використанням *Buf*
$line_0 := line_n;$ // перехід до наступної частини фрагменту
обчислити $x_{best}, y_{best}, D_{best}, D_{alt}$;

Рисунок 5. – Алгоритм пошуку великого фрагменту

Алгоритми на рис. 3 і рис. 5 забезпечують повне завантаження всіх потоків при $H_s W_s \equiv 0 (\bmod M)$. Усі звернення до пам'яті пристрою є з'єднаними. Часова складність обох алгоритмів складає

$$O\left(\left\lceil \frac{W_0 H_0}{M} \right\rceil WH\right), \qquad (3)$$

але алгоритм на рис. 5 виконує більше допоміжних операцій. Вибір алгоритму на рис. 3

чи рис. 5 здійснюється для кожного фрагменту в залежності від W та H.

Практичні результати

Описані алгоритми реалізовано і протестовано на відеокарті Palit GeForce GTS 250 E-Green.

На рис. 6 показано отриману експериментально залежність середнього часу пошуку фрагменту від W_s, H_s, W та H із використання 128 потоків на кожен фрагмент. Як видно з рис. 6, при $W_s H_s < M$, час пошуку майже не залежить від W_s і H_s, що узгоджується з (3).

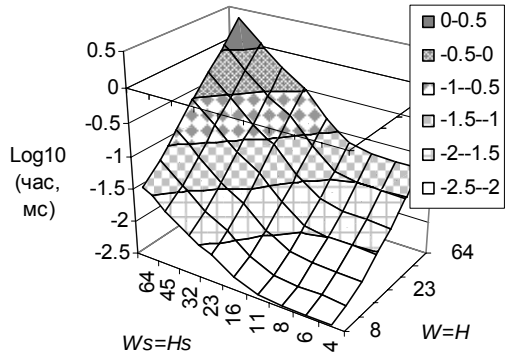

Рисунок 6. – Середній час пошуку фрагмента

Прискорення пошуку на GeForce GTS 250 E-Green порівняно із однопотоковою реалізацією повного пошуку на процесорі Athlon X2 5400 наведено на рис. 7.

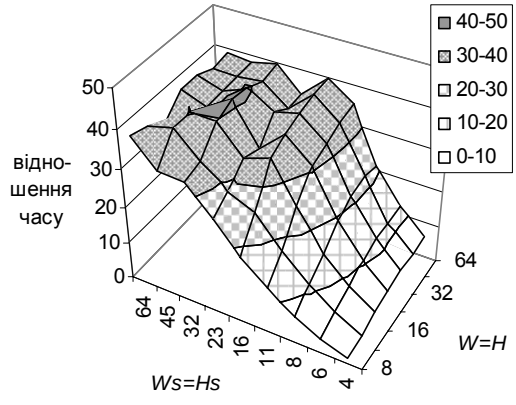

Рисунок 7. – Прискорення пошуку фрагментів на GPU порівняно з CPU

Як видно з рис. 7, використання відеокарти середнього рівня забезпечує прискорення в порівнянні з одним ядром процесора до 40 разів. При цьому знижується завантаження ЦП і він може паралельно проводити інші обчислення. При використанні двох або чотирьох ядер процесора час обробки множин фрагментів скоротиться майже в два або чотири рази; при використанні потужніших відеокарт, час обчислень на них також скоротиться у декілька разів. На практиці прискорення може бути меншим через накладні витрати підготовку даних у пам'яті пристрою.

Висновки

У статті представлено апаратно-орієнтовані алгоритми пошуку фрагментів зображень для паралельної архітектури CUDA. Вони може бути використані для прискорення відстежування об'єктів у відеопотоці.

Алгоритми реалізовані і досліджені експериментально. В порівнянні з одним ядром процесора Athlon X2 5400 на відеокарті Palit GeForce GTS 250 E-Green досягнуте зменшення часу пошуку множини фрагментів однакового розміру до 40 разів.

Використання пристрою з архітектурою CUDA може стати хорошою альтернативою використанню багатопроцесорних ЕОМ або кластера при відстежуванні об'єктів у відеопотоці.

Подальшого збільшення швидкодії розробленої системи відстежування об'єктів [2] можна досягти за рахунок реалізації на архітектурі CUDA алгоритмів моделювання і віднімання фону, а також розпізнавання об'єктів. Результат віднімання фону в пам'яті пристрою може бути вхідними даними алгоритмів пошуку фрагментів і розпізнавання об'єктів, за рахунок чого можна добитися зменшення трафіку меду пам'яттю комп'ютера і пристрою.

Слід зазначити, що задача має цілочисельну природу, і, якщо на майбутніх CUDA GPU буде реалізовано більш швидкі цілочисельні арифметичні операції, прискорення на GPU у порівнянні із CPU може бути значно більшим.

Література

1. R. Cucchiara, C. Grana, G. Tardini. Track-based and object-based occlusion for people tracking refinement in indoor surveillance. //Proceedings of the ACM 2nd international workshop on Video surveillance & sensor networks 2004, New York, NY, USA, October 15, 2004. – 10 p. (http://doi.acm.org/10.1145/ 1026799.1026814).
2. Ладиженський Ю.В., Середа А.О. Відстежування об'єктів у відеопотоці на основі відстежування переміщення фрагментів об'єктів //Наукові праці Донецького національного технічного університету. Серія: "Обчислювальна техніка та автоматизація". Випуск 17 (148) – Донецьк: ДонНТУ, 2009. – 215 с.
3. Adaptive Motion Estimation Processor for Autonomous Video Devices. T. Dias, S. Momcilovic, N. Roma, L. Sousa. EURASIP Journal on Embedded Systems, vol. 2007, Article ID 57234, 2007. – 10 p. (http://www.hindawi.com/GetArticle.spx?doi=10.1155/2007/57234).
4. NVIDIA CUDA Programming Guide Version 2.3.1, 2009. – 145 p. (http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf).
5. NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3, 2009. – 145 p. (http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf).