



جامعة دمشق

كلية الهندسة الميكانيكية والكهربائية

قسم هندسة الإلكترونيات والاتصالات

تحسين أداء الخوارزميات المستخدمة في معالجة الإشارات الرقمية باستخدام الحوسبة غير المتجانسة النقالة

بحث علمي أعد لنيل درجة الماجستير في هندسة الإلكترونيات التطبيقية

إعداد

المهندس لؤي عبد الصاحب ياري

بإشراف

الأستاذ الدكتور المهندس جبر الريحين

العام الدراسي

2016-2017

أعضاء لجنة الحكم والقرار الناظم بذلك

أقدم أسمى الشكر والامتنان والمحبّة لأعضاء لجنة الحكم الموقرة لقبولهم
قراءة خلاصة بحثي والحكم على رسالتي :

عضوا

الدكتور نديم شاهين

عضوا ومشرفا علميا

الدكتور جبر الرمحين

عضوا

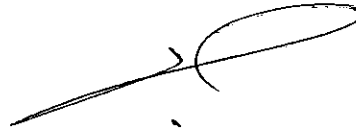
الدكتور عادل خضور علي

بموجب قرار مجلس البحث العلمي والدراسات العليا رقم / ١١٨٣ م.ب.د.
تاريخ ٢٠١٧/٣/٦ والقاضي بتأليف لجنة الحكم

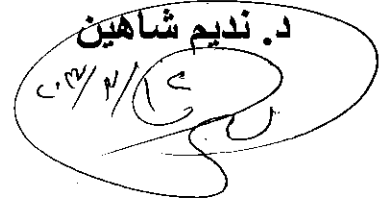
د. عادل خضور علي



د. جبر الرمحين



د. نديم شاهين



كلمة شكر وتقدير

لا يسعني في هذا المقام إلا أن أتقدم بخالص الشكر والعرفان للمشرف على هذا البحث

الأستاذ الدكتور جبر الرمحين

حيث كان لصبره وسعة صدره وعنايته لي الدور الأكبر في إنجاز هذا العمل، وقد بذل من وقته الثمين الكثير من الوقت في توجيهي ونصحي ووضعني على الطريق الصحيح، وأجد من الصعب أن أوافيه حقه، أمد الله بالصحة والعافية.

كما أشكر عمادة كلية الهندسة الميكانيكية والكهربائية ممثلة بـ

الأستاذ الدكتور محمد مازن حاييرجي عميد الكلية

ورئيس قسم هندسة الإلكترونيات والاتصالات

الأستاذ الدكتور عادل خضور

على جهوده ولما قدمه من دعم ومساندة، كما أشكر كل من كان له يد يبذلها في المساعدة على إنجاز هذا العمل المتواضع، سائلي المولى عز وجل التوفيق والسداد.

الباحث

الإهداء

إلى حافزي في الحياة

إلى من بذل من جهده ونفعه من أجلي

إلى من يحمل قمتي

إلى من علمني اللاهنا والعدل

أبي

إلى نفع العطاء الذي لا ينضب

إلى من أودى لها بكل ما وصلت إليه

إلى من علمتني معاني الحياة

إلى أختي ما في الوجوه

أخي

إلى نفع العفو والنباب

إلى فرحي

إلى منى في الحياة

إلى من يناظرني هوى الحياة

أختي

إلى كل من رافقوني رحلة عمري

إلى الذين عزفوا معي على قيثارة الحياة

أصدقائي

إلى أفعلي وأجرائي

إلى جميع أفعلي عملي قنل

ملخص

تعتمد الخوارزميات المستخدمة في عمليات معالجة الإشارة الرقمية Digital Signal Processing، كمعالجة الصور Image Processing وكشف الكائنات Objects Detection، والتي تستخدم في أبحاث وأعمال هامة على الصعيدين العلمي والطبي، على عمليات حسابية معقدة وطويلة، وغالباً ما تنفذ باستخدام أجهزة حاسوبية خارقة Super Computers ضخمة، ومتواجدة في مكان ثابت، وبحجم واستهلاك للطاقة كبيرين، وبسعر مرتفع ولا تكون متاحة لجميع الباحثين وخصوصاً في منطقة الشرق الأوسط.

يهدف البحث إلى توفير أداة حوسبة غير متجانسة نقالة Mobile Heterogeneous Computer للباحثين والعلماء رخيصة الثمن، تعمل كحاسوب خارق، وبالتالي تسريع تجارب البحث العلمي والوصول إلى النتائج واختبارها عملياً بدلاً من الاكتفاء بالدراسة النظرية البحتة دون أي تطبيق يعود بالفائدة على العلم والمجتمع.

وفي هذا البحث تم دراسة آلية استخدام وحدات المعالجة الرسومية Graphical Processing Units (GPUs) كأدوات للبرمجة عامة الغرض General Purpose ضمن الحوسبة غير المتجانسة Heterogeneous Computing، لمعرفة مدى تأثيرها على سرعة وأداء الخوارزميات، والعمليات الحسابية المستخدمة في معالجة الإشارة الرقمية Digital Signal Processing. ثم تم التطرق للحوسبة غير المتجانسة النقالة Mobile Heterogeneous Computing ومقارنة النتائج مع الحواسيب غير المتجانسة التقليدية من حيث سرعة الأداء والتكلفة واستهلاك الطاقة والحجم.

وكتطبيق عملي على البحث تم دراسة التقنيات المتبعة في أنظمة كشف قزحية العين Iris Recognition والتركيز على مرحلة تحديد حدود القزحية Iris Localization وكيفية تسريعها ضمن النظام وذلك بتحويل الرماز التسلسلي Serial Code المستخدم إلى رماز متوازي Parallel Code ثم مقارنة النتائج من حيث سرعة التنفيذ باستخدام الرماز على حاسوب عادي محمول بمواصفات متوسطة وعلى أداة التطوير Jetson TK1 من شركة NVIDIA.

فهرس المحتويات

I	فهرس المحتويات
IV	فهرس الأشكال
VI	فهرس الجداول
VII	جدول المصطلحات
1	الهدف من البحث:
2	مقدمة:
5	Principles of heterogeneous computing
6	مقدمة
6	1. 1. مبادئ الحوسبة المتوازية Basics of Parallel Computing
7	1. 1. 1. التسريع Speedup
8	1. 1. 2. تصنيف الأنظمة المتوازية Classification of Parallel Systems
9	1. 1. 3. منهجيات التواصل Communication Methodologies
11	1. 2. أنماط البرمجة المتوازية Parallel programming patterns
11	1. 2. 1. نظرة عامة Overview
11	1. 2. 2. المراقبة Map
11	1. 2. 3. الجمع Gather
12	1. 2. 4. التبعثر Scatter
13	1. 2. 5. التخفيض Reduce
14	1. 2. 6. المسح Scan
16	الفصل الثاني معمارية وحدة معالجة الرسومات GPU Architecture
17	مقدمة
18	1. 2. معمارية وحدة معالجة الرسومات GPU Architecture
19	1. 1. 2. تنظيم ذاكرة وحدة معالجة الرسومات GPU Memory Organization
21	1. 2. القيود في استخدامات وحدة معالجة الرسومات كتهيئة للأغراض العامة GPGPU usage limitations
21	2. 2. عناصر معمارية تهيئة الحساب الموحد CUDA Elements
21	1. 2. 2. كتل البناء Building Blocks
22	2. 2. 2. قابلية التوسع لتطبيق CUDA Scalability of a CUDA application
23	2. 2. 3. عناصر تطبيق CUDA Elements of a CUDA application
26	الفصل الثالث معمارية وحدة NEON
27	مقدمة
27	1. 3. التحميل والتخزين Load and Stores
29	1. 1. 3. مبدأ عمل إلغاء التشابك de-interleave والتشابك interleave
32	1. 2. العناصر الوحيدة أو المتعددة Single or Multiple Elements

33	3. 1. 3. العنوانة Addressing
34	3. 1. 4. أشكال أخرى من التحميل والتخزين Other Loads and Stores
34	3. 2. التعامل مع أبعاد الصفيفة غير المضاعفة Handling non-multiple array lengths
35	3. 2. 1. البقايا Leftovers
35	3. 2. 2. مثال عن المشكلة Example problem
35	3. 2. 3. استخدام صفيفات أكبر Using Larger Arrays
38	3. 2. 4. التداخل Overlapping
40	3. 2. 5. معالجة العنصر الوحيد Single Element processing
42	3. 2. 6. المحاذاة Alignment
42	3. 2. 7. استخدام تعليمات ARM
42	3. 3. ضرب المصفوفات Matrix Multiplication
43	3. 3. 1. الخوارزمية المستخدمة Used Algorithm
44	3. 3. 2. الرماز المنجز Implemented Code
48	3. 4. عمليات الإزاحة Shift operations
48	3. 4. 1. إزاحة الأشعة Shifting Vectors
49	3. 4. 2. الإزاحة والإدخال Shifting and inserting
50	3. 4. 3. الإزاحة والمراكمة Shifting and accumulating
50	3. 4. 4. معدّلات التعليمات Instruction Modifiers
51	3. 4. 5. جدول بتعليمات الإزاحة المتاحة Table of shifts available
51	3. 4. 6. مثال تطبيقي: تحويل عمق اللون Color Depth Conversion
53	3. 5. تعليمات مبادلة الأشعة Instructions to permute vectors
53	3. 5. 1. مقدمة Introduction
54	3. 5. 2. البدائل Alternatives
58	الفصل الرابع التعرف على قزحية العين Iris Recognition
59	4. 1. مدخل إلى تحديد مكان قزحية العين Introduction to Iris Localization
62	4. 2. دراسة مرجعية
62	4. 3. التعرف على قزحية العين Iris Recognition
64	4. 4. تحديد مكان قزحية العين Iris Localization
65	4. 4. 1. طريقة دوغمان Daugman's Method
65	4. 4. 2. مخطط وايلد وآخرون Wilde et al.'s Scheme
66	4. 4. 3. مخططات أخرى Other Schemes
67	4. 5. منهج موازاة التعرف على قزحية العين Parallel Approach to Iris Recognition
70	4. 6. الخلاصة Summary
71	الفصل الخامس تحديد مكان قزحية العين المتوازي Parallel Iris Localization
72	الهدف

72	 Traditional Hough Transform تحويل هوف التقليدي
74	 Parallel Algorithm خوارزمية التوازي
77	 Time Complexity Analysis تحليل تعقيد الزمن
78	 Implementation & Results الفصل السادس التنفيذ العملي والنتائج
79	 Jetson TK1 1. 6. مدخل إلى أداة التطوير
80	 NVIDIA Tegra K1 2. 6. معمارية
82	 CPU 1. 2. 6. وحدة المعالجة المركزية
83	 GPU 2. 2. 6. وحدة معالجة الرسومات
83	 3. 6. تنفيذ التجربة
83	 Implementation Environment 1. 3. 6. بيئة التنفيذ
85	 Results 2. 3. 6. النتائج
87	 الخلاصة والآفاق المستقبلية:
87	 الخلاصة:
88	 الآفاق المستقبلية:
89	 (References): المراجع
93	 Appendences الملاحق

فهرس الأشكال

الشكل 1-1 نمط المقابلة Map.....	11
الشكل 2-1 نمط الجمع Gather.....	12
الشكل 3-1 نمط التبعثر Scatter.....	12
الشكل 4-1 التداخل بين النياسب thread collision.....	12
الشكل 5-1 نمط التخفيض Reduce قبل التجزئة.....	13
الشكل 6-1 نمط التخفيض Reduce بعد التجزئة.....	13
الشكل 7-1 التخفيض المتوازي Parallel Reduce.....	14
الشكل 8-1 نمط المسح Scan.....	14
الشكل 9-1 المسح المتوازي Parallel scan.....	15
الشكل 1-2 معمارية GPU مكون من 96 نواة، كل SM تحوي 8 نوى.....	20
الشكل 2-2 كتل بناء CUDA.....	22
الشكل 3-2 قابلية التوسع لتطبيق CUDA.....	23
الشكل 4-2 نموذج التنفيذ Execution model.....	25
الشكل 1-3 تحميل معطيات RGB باستخدام التحميل الخطي.....	28
الشكل 2-3 تحميل معطيات RGB باستخدام التحميل البنيوي.....	28
الشكل 3-3 تبديل محتويات المسجلين d0 و d2.....	29
الشكل 4-3 تحميلات وتخزينات NEON البنيوية.....	29
الشكل 5-3 صيغة تعليمية التحميل و التخزين البنيوية.....	30
الشكل 6-3 تحميل وإلغاء تشابك معطيات بعرض 16 بت.....	31
الشكل 7-3 تحميل وإلغاء تشابك معطيات بعرض 32 بت.....	32
الشكل 8-3 تحميل وإلغاء التشابك لجميع مسارب الأشعة.....	33
الشكل 9-3 تحميل وإلغاء تشابك لمسرب شعاع وحيد.....	33
الشكل 10-3 حشو صفيقة (اللون الرمادي) لملء عدد صحيح من الأشعة.....	36
الشكل 11-3 الأشعة المتداخلة، حيث يتم معالجة العناصر البرتقالية مرتين.....	38
الشكل 12-3 معالجة العناصر الوحيدة.....	40

الشكل 3-13	ضرب مصفوفتين يظهر عمود وحيد من النتائج	43
الشكل 3-14	ضرب شعاع مع موسع ضمن بنية NEON	43
الشكل 3-15	إزاحة شعاع باستخدام شعاع الإزاحة	49
الشكل 3-16	إزاحة شعاع وإدخال النتائج إلى شعاع آخر	50
الشكل 3-17	صيغ اللون RGB888 و RGB565	52
الشكل 3-18	مبدأ عمل مسجل الفهرسة	57
الشكل 4-1	صورة عينة لقزحية العين	59
الشكل 4-2	المراحل المستخدمة ضمن التعرف على قزحية العين	61
الشكل 4-3	نتائج تحديد موقع القزحية بدون إزالة الجزء المغطى من قبل الجفون	62
الشكل 4-4	تحديد مكان قزحية العين بعد إزالة الجفون	64
الشكل 4-5	حوسبة متوازية لمسافة هامينغ على وحدة معالجة الرسومات GPU	69
الشكل 6-1	أداة التطوير Jetson TK1	79
الشكل 6-2	المخطط الصندوقي لأداة التطوير Jetson TK1	80
الشكل 6-3	شريحة NVIDIA Tegra K1 (إصدار 32-bit)	81
الشكل 6-4	زمن التنفيذ مقابل حجم الصورة بالاعتماد على نتائج جدول 4-6	86
الشكل 6-5	اختلاف مقدار التسريع وفقاً لحجم الصورة بالاعتماد على نتائج جدول 4-6	86

فهرس الجداول

جدول 1-3 عمليات الإزاحة ضمن NEON	51
جدول 1-6 مقارنة القياسات بين Tesla K40C و Tegra K1 من NVIDIA. حيث تقيس " power consumption" متوسط الطاقة المستهلك من قبل GPU أثناء تشغيل مجموعة من ألعاب النقال المشهورة.	83
جدول 2-6 تفاصيل بيئة التنفيذ	84
جدول 3-6 أزمان التنفيذ التسلسلية و المتوازية لخوارزمية تحديد موقع القزحية على عينة من الصور العشوائية بقياس 320x280 مأخوذة من قاعدة معطيات CASIA	85
جدول 4-6 أزمان تنفيذ خوارزمية تحديد مكان قزحية العين على صور قزحيات عين بأحجام مختلفة	86

جدول المصطلحات

API	Application Programming Interface
ARM	Advanced Risc Machines, British Chip Design Developer. The ARM Chips are produced by many companies.
ARM Cortex	Is the name of a CPU architecture starting with ARMv7 which has some things in common like floating point support (Cortex-A). Cortex-A, Application profile Cortex-R, Real-time profile Cortex-M, Microcontroller profile
Cortex A15	This CPU design offers about 3.5 to 4 DMIPS/Mhz which makes it about 30% percent faster than older Cortex A9 designs at the same clock rate. The Tegra K1 32 Bit CPU is based on a quad-core Cortex A15 design with an additional power saving core.
CPU	Central Processing Unit
CSI	Camera serial interface, connection to cameras
CUDA	Compute Unified Device Architecture, from NVidia developed programming model, which allows code to be executed on the GPU. The programming model allows to write parallel computing programs which can be much faster than code running on the CPU.
Embedded System	Computer which does a specialized task (e.g. control task) in contrast to a desktop computer.
eMMC	embedded Multimedia Card, the board has on-board 16 GB eMMC flash memory
GPIO	General purpose input/output pin which can be used to control external hardware (switch on/off something)
GPU	Graphical Processing Unit
ISP	Image signal processor, used for processing input data from cameras
OpenCL	Open Computing Language
OpenGL	3D API for PC's and also embedded systems
OpenVG	2D API for vector- and bitmap graphics
USB 2.0	Universal Serial Bus - here mouse, keyboard and other devices are connected.
LED	Light Emitting Diode
HDMI	High Definition Multimedia Interface (Jetson TK1 supports HDMI Version 1.4) The maximum screen resolution is 4096x2160@30 on the HDMI interface.
PCIE	Peripheral Component Interconnect Express, is a high-speed serial computer expansion bus
RAM	Random Access Memory, the main memory
RS-232	serial interface (the Jetson TK1 has one RS-232 interface) which can be used for diagnosis.
SD Card	Secure Digital Memory Card. If you buy an SD card it's good to choose a higher class SD card like class 10.
SoC	System on Chip, means that different functions like processing and video interfaces and Input/Output interfaces are integrated on one chip
SPI	Serial Peripheral Interface, interface to displays
Tegra	Is a SoC series developed by Nvidia. The 32 Bit Tegra K1 is a Cortex A15 multicore CPU design combined with a 192 core Kepler GPU (graphics engine). The 32 Bit K1 modelname is T124. Kepler is just a name for a GPU microarchitecture developed by Nvidia. The K1 is produced with a 28 nm process. Nvidia is also developing a 64 Bit Tegra K1 chip with the same GPU.

الهدف من البحث:

تعتمد الخوارزميات المستخدمة في عمليات معالجة الإشارة الرقمية Digital Signal Processing، كمعالجة الصور Image Processing وكشف الكائنات Objects Detection، والتي تستخدم في أبحاث وأعمال هامة على الصعيدين العلمي والطبي، على عمليات حسابية معقدة وطويلة، وغالباً ما تنفذ باستخدام أجهزة حاسوبية خارقة Super Computers ضخمة، ومتواجدة في مكان ثابت، وبحجم واستهلاك للطاقة كبيرين، وبسعر مرتفع ولا تكون متاحة لجميع الباحثين وخصوصاً في منطقة الشرق الأوسط.

يؤدي توفير أداة حوسبة غير متجانسة نقالة Mobile Heterogeneous Computer للباحثين والعلماء رخيصة الثمن، تعمل كحاسوب خارق، إلى تسريع تجارب البحث العلمي والوصول إلى النتائج واختبارها عملياً بدلاً من الاكتفاء بالدراسة النظرية البحتة دون أي تطبيق يعود بالفائدة على العلم والمجتمع.

تم في هذا البحث دراسة آلية استخدام وحدات المعالجة الرسومية Graphical Processing Units (GPUs) كأدوات للبرمجة عامة الغرض General Purpose ضمن الحوسبة غير المتجانسة Heterogeneous Computing، لمعرفة مدى تأثيرها على سرعة وأداء الخوارزميات والعمليات الحسابية المستخدمة في معالجة الإشارة الرقمية Digital Signal Processing. ثم تم التطرق للحوسبة غير المتجانسة النقالة Mobile Heterogeneous Computing ومقارنة النتائج مع الحواسيب غير المتجانسة التقليدية من حيث سرعة الأداء والتكلفة واستهلاك الطاقة والحجم.

وكتطبيق عملي على البحث تم دراسة التقنيات المتبعة في أنظمة كشف قزحية العين Iris Recognition والتركيز على مرحلة تحديد حدود القزحية Iris Localization وكيفية تسريعها ضمن النظام وذلك بتحويل الرمز التسلسلي Serial Code المستخدم إلى رمز متوازي Parallel Code ثم مقارنة النتائج من حيث سرعة التنفيذ عندما يتم استخدام الرمز على حاسوب عادي محمول بمواصفات متوسطة، وعلى أداة التطوير Jetson TK1 من شركة NVIDIA.

مقدمة:

تطورت الأجهزة النقلة بشكل سريع على مدى العقد الماضي وأصبحت متوفرة في كل مكان، مما أدى إلى إيجاد تطبيقات جديدة تعتمد بشكل أساسي على الحوسبة النقلة Mobile Computing، الاتصالات اللاسلكية Wireless communications وتقنيات التصوير الرقمي Digital imaging technologies.

وأصبحت معالجات الأجهزة النقلة Mobile Processors أكثر وفرة وتنوعاً خلال الأعوام الماضية وذات أداء أعلى، إذ يشهد العالم حالياً نمواً سريعاً في الطلب على تطبيقات الرؤية الحاسوبية Computer Vision العاملة على الأجهزة النقلة كتحريك الصور Image editing، الواقع المعزز Augmented Reality والتعرف على الكائنات وغيرها. [1] [2] [3]

ومن جهة ثانية فإن التحسينات والتطورات في مجالات الرؤية الحاسوبية Computer Vision والواقع المعزز Augmented Reality زاد من تعقيد وكثافة العمليات الحسابية للخوارزميات الناشئة، وبالتالي أدى إلى زيادة الزمن اللازم للمعالجة بسبب التعقيد الحسابي العالي الذي منع بدوره من تطبيق خوارزميات الرؤية الحاسوبية Computer Vision عملياً واستخدامها في التطبيقات النقلة.

ولحل هذه المشكلة عمد الباحثون إلى استكشاف الحوسبة عامة الغرض باستخدام وحدات معالجة الرسومات (GPGUs) general-purpose computing using graphics processing units كمسرعات لتسريع خوارزميات معالجة الصور Image Processing والرؤية الحاسوبية Computer Vision بفضل المعمارية غير المتجانسة للمعالجات النقلة الحديثة heterogeneous architecture of the modern mobile processors. [4] [5] [6] [7] [8] [9]

على صعيد الحواسيب المكتبية Desktop Computers أو الأجهزة الحاسوبية الخارقة Super Computers فقد تم دراسة العديد من نماذج البرمجة programming modules على نطاق واسع والاستفادة منها لتسهيل البرمجة المتوازية Parallel Programming باستخدام general-purpose computing using graphics processing units (GPGUs) مثل معمارية الجهاز للحساب الموحد Compute Unified Device Architecture (CUDA) ولغة الحوسبة المفتوحة Open Computing Language (OpenCL). [10] [11]

ونظراً لافتقار نطاق الأجهزة النقلة Mobile Domain لنماذج البرمجة المتوازية Parallel Programming Models، تم استخدام نموذج برمجة مكتبة الرسومات المفتوحة للأنظمة المضمنة

Open Graphics Library for Embedded Systems أو ما يعرف اختصاراً بـ OpenGL ES بشكل شائع لتسخير قدرة الحساب لوحدة معالجة الرسوميات النقلة (GPU) Mobile Graphical Processing Unit (GPU) [12]. ومع ذلك أدت محدودية نموذج OpenGL ES إلى صعوبة التطوير والمتابعة فضلاً عن محدودية الأداء المتوازي Parallel Performance، ويرجع ذلك إلى حقيقة أن الغرض الأساسي من تصميم نموذج OpenGL ES هو تصيير الرسوميات ثلاثية الأبعاد 3D Graphics Rendering.

وفي الآونة الأخيرة تم دعم المعالجات الحديثة النقلة من قبل بعض نماذج البرمجة الناشئة مثل OpenCL Embedded [10] و Render-Script [13] مما جعل الحوسبة عامة الغرض باستخدام وحدات معالجة الرسوميات النقلة general-purpose computing using graphics processing units (GPGPUs) أمراً ممكناً للهواتف النقلة في العالم الحقيقي لأول مرة.

ولقد وفرت شركة Nvidia شرائح SoC System on Chip تحتوي على معالجات رسومية متعددة النوى بعمارية Kepler في شرائح TK1 و Maxwell في شرائح TX1، والمتوافقة مع نمودجي البرمجة CUDA و OpenCL، وبأداء يوافق أداء وحدات الرسوميات على الأجهزة المكتبية وباستهلاك طاقة منخفض جداً [15] [14]، وبالتالي سمحت بإمكانية نقل الخوارزميات التي تم عملها على الحواسيب المكتبية أو الخارقة إلى الأجهزة النقلة وغير المتجانسة Mobile Heterogeneous Devices بأقل جهد وبأعلى توافقية.

وكتطبيق عملي على هذا البحث تم اختيار تطبيق كشف قزحية العين Iris Recognition من أجل استكشاف قدرة وحدة معالجة الرسوميات النقلة كأداة عامة الغرض general-purpose computing using graphics processing unit (GPGPU) لتسريع خوارزميات الرؤية الحاسوبية Computer Vision المعتمدة عليها.

تنظيم البحث:

الهدف من البحث ومقدمة وملخص باللغتين العربية والإنكليزية وست فصول:

الفصل الأول: دراسة مرجعية عن مبادئ الحوسبة المتوازية Parallel Computing وأنماط برمجتها مع مقدمة عن تحديد مكان قزحية العين والدراسات والتطبيقات المتوازية التي تم تطبيقها عليها. وتم التطرق بشكل موجز عن كيفية تحديد مكان قزحية العين Iris Localization وعرض بعض خوارزمياتها التي تم تسريعها باستخدام الحوسبة المتوازية.

الفصل الثاني: دراسة مرجعية عن مبادئ الحوسبة المتوازية Parallel Computing باستخدام GPGPU وشرح عن نموذج البرمجة المستخدم فيها CUDA وهو محور أساسي لهذا البحث.

الفصل الثالث: يستعرض مبادئ البرمجة باستخدام معمارية NEON الموجود في الأجيال الجديدة من معالجات ARM والتعامل معها بكفاءة عالية من أجل تحسين سرعة أداء البرامج والخوارزميات في أثناء كتابتها وترجمتها على معمارية الأجهزة غير المتجانسة النقالة.

الفصل الرابع: دراسة مرجعية عن نظام التعرف على قزحية العين Iris Recognition والخوارزميات المستخدمة ضمنها والأعمال المنجزة إلى الآن.

الفصل الخامس: يستعرض إمكانية تسريع الخوارزمية المستخدمة في مرحلة تحديد موقع قزحية العين Iris Localization ضمن أحد مراحل كشف قزحية العين Iris Recognition وذلك بنقل الأجزاء البرمجية من رماز الخوارزمية من النمط البرمجي التسلسلي Serial Code إلى النمط البرمجي المتوازي Parallel Code وتنفيذه على وحدة معالجة الرسومات GPU.

الفصل السادس: تم القيام بتطبيق الخوارزمية على النظام المضمن Jetson TK1 وذلك قبل عملية نقل أجزاء الرماز من تسلسلي إلى متوازي، وبعدها تمت مقارنة النتائج من حيث السرعة والأداء واستهلاك الطاقة والتكلفة بينها وبين استخدام حاسوب شخصي بوحدة معالجة رسومات GPU للاستخدام العادي ثم عرض الآفاق المستقبلية.

الفصل الأول

أساسيات الحوسبة غير المتجانسة

Principles of heterogeneous computing

مقدمة

خلال السنوات القليلة الماضية، تصدرت الحوسبة المتوازية Parallel Computing أهم مجالات البحث النشطة، مع محاولة العديد من رواد التقنية بتحسين كفاءة تطبيقاتهم من خلال خوارزمياتها وتقانتها. كما تم تطبيقها في العديد من المجالات بدءاً من التنبؤ بحالة الطقس Weather Forecasting مروراً باستكشاف المعادن mineral discovery، بالإضافة إلى معالجة الصور حيث أن جميع هذه التطبيقات تحتاج إلى كم هائل من الحوسبة.

وقد ازداد الطلب على المعالجة السريعة لمواكبة النمو الأخير في القدرة التخزينية والحسابية للآلات، ولقد أظهرت التوجهات الأخيرة تحسن ملحوظ في نمو الحوسبة المتوازية وخصوصاً في مجالات معالجة الصورة image processing ومعالجة الفيديو video processing وتصيير الرسومات graphics rendering والمعاينة visualization.

تعدُّ البطاقات البيانية المتقدمة اليوم أكثر قدرة على إنجاز تطبيقات تصيير المعطيات بانسيابية على عكس الحال في السنين السابقة لاحتوائها على العشرات وحتى المئات من نوى المعالجة القادرة على إجراء عمليات الفاصلة العائمة عالية الدقة High precision floating point arithmetic operations.

1.1. مبادئ الحوسبة المتوازية Basics of Parallel Computing:

تهدف الحوسبة المتوازية إلى حل المشاكل المعقدة عبر تقسيم المسألة p إلى مسائل جزئية وحل كل منها بواسطة نيسب thread منفصل والتي يمكن تنفيذها على التوازي [16]. تميل الخوارزميات المتوازية إلى الاستفادة من موارد النظام الأكثر فعالية والاستفادة من النظم متعددة النوى والتي تستخدم اليوم على نطاق واسع. وتهدف هذه الخوارزميات إلى استكشاف التفرع concurrency الممكن انجازه أثناء تنفيذ تطبيق ما.

تميل معظم تطبيقات الزمن الحقيقي إلى كونها كثيفة الحوسبة intensive computing، وبالتالي أصبح من الضروري استكشاف التوازي في التطبيقات إن وجدت، والتي تقدم تسريع كبير significant speedup للتطبيق.

1.1.1. التسريع Speedup:

يتم التعبير عن الربح gain في أي خوارزمية متوازية بمصطلح التسريع speedup والذي يعرف بنسبة الزمن اللازم لتنفيذ execute التطبيق application على آلة تسلسلية serial machine إلى الزمن المطلوب لتنفيذ التطبيق على آلة متوازية parallel machine [16] والتي تعطى بالمعادلة:

$$Speedup = \frac{T_s}{T_p} \quad (1-1)$$

حيث T_s : الزمن المستغرق لتنفيذ التطبيق على آلة تسلسلية، و T_p : الزمن المستغرق لتنفيذ التطبيق على آلة متوازية. ويتأثر هذا التسريع بالعديد من العوامل factors والأعباء overheads. والقيود التي تؤثر على تحقيق التسريع في الآلة المتوازية، وهي:

أ- تلبّث الاتصال Communication Latency

يتم تحليل المسألة التتابعية sequential problem ومقابلتها mapping على عدد n من المعالجات processors، حيث يقوم كل معالج بتنفيذ مجموعة جزئية محدّدة particular subset من مجموعة المعطيات dataset. خلال الحوسبة المتوازية يصبح التواصل communication المعني بين عناصر المعالجة processing elements عامل التقييد limiting factor على الربح في التسريع، والتي تستوجب

بذل اهتمام كبير لمعالجة مقدار amount التواصل المشمول بين المعالجات من جهة ومقدار التواصل المشمول بين المعالج والذاكرة من جهة أخرى.

ب- مقدار (حجم) الرماز التتابعي Amount of Sequential Code

تُقَيَّد الخوارزميات المتوازية أيضاً بمقدار حجم الرماز التتابعي الموجود ضمن الخوارزمية، فإذا استغرق الرماز التتابعي زمن تنفيذ أكبر من الرماز المتوازي داخل الخوارزمية، عندها سيفقد الرماز المتوازي مزاياه.

فعلى سبيل المثال، إذا كان 25% من التطبيق تسلسلياً serial عندها لن نحصل على تسريع أكثر من أربع مرات مهما كان عدد النوى cores. وبالتالي يجب أخذ الحذر عند عملية موازنة التطبيق وإلا فسيتم هدر الموارد المتاحة في النظام غير المتجانس ببساطة.

ج- تبعية المهام Task Dependency

تعدُّ التبعية في المهام قيداً آخر على تطبيقات الحوسبة المتوازية، فإذا كانت المهام الحالية تعتمد على المهام السابقة عندها يصبح من الصعب توظيف employ الموازنة parallelism على هذه المهام لأن نتائج المهام السابقة يجب أن تكون متاحة لأجل إنجاز المهام الجديدة، لذلك لا يمكن تنفيذ الموازنة.

تكون عملية الموازنة أكثر فائدة عندما تكون المهام المقسّمة divided tasks مستقلة عن بعضها افتراضياً virtually ويمكن تنفيذها في آن واحد simultaneously بدون أن تؤثر إحدى النتائج على أخرى، وفي النهاية يمكن دمج النتائج ببساطة للحصول على النتيجة النهائية. وهذا الأمر يتطلب مجهوداً كبيراً في البرمجة لإتمام التنفيذ المتوازي المستقل للمهام.

1.1.2. تصنيف الأنظمة المتوازية Classification of Parallel Systems

تصنف أطر عمل الحوسبة المتوازية parallel computing frameworks حسب تصنيف فلين Flynn classification [17] الذي قسّم معمارية التوازي parallel architecture إلى أربع مجموعات:

أ- تعليمية وحيدة معطيات وحيدة (SISD) (Single Instruction Single Data):

تحتوي هذه المجموعة من أنظمة الحوسبة معالج تتابعي وحيد sequential processor يقوم بتنفيذ المهام واحدة تلو الأخرى وبالتالي لا يوظف أي نوع من الموازنة parallelism.

ب- تعليمة وحيدة معطيات متعددة (SIMD (Single Instruction Multiple Data):

تعدّ هذه المجموعة من أهم الأصناف في المعماريات المتوازية parallel architecture والتي توظف عناصر معالجة متعددة multiple processing elements، تعمل على مجموعة مختلفة من المعطيات، ومن الملاحظ أن جميع عناصر المعالجة تقوم بتشغيل نفس الخوارزمية لتوظف نفس العملية على حزمة المعطيات، وبالتالي فإنّ هذه المعمارية تصبح أكثر فائدة عند تطبيق عمليات متماثلة على معطيات مختلفة.

ج- تعليمات متعددة معطيات وحيدة (MISD (Multiple Instruction Single Data):

يطبق هذا الإصدار من معمارية الحاسوب عدة تعليمات على مجموعة وحيدة من المعطيات على معالجات مختلفة. وعموماً هذا النوع من الخصائص ذو استخدام قليل عندما يتعلق الأمر بالحوسبة المتوازية.

د- تعليمات متعددة معطيات متعددة (MIMD (Multiple Instruction Multiple Data):

تعدّ هذه المعمارية عملية أيضاً عندما يلزم تشغيل عدة خوارزميات على معطيات مختلفة، وهذا النوع من المعماريات مستخدم غالباً في النظم الموزعة Distributed environments. وبالتالي تقوم آلات SIMD بتوظيف الموازة على مستوى المعطيات. وتعدّ وحدات معالجة الرسوميات Graphical Processing Units (GPUs) تطبيقات واسعة لـ SIMD قادرة على تفريع branches، تحميل loads وتخزين stores من 128-bit إلى 256-bit في آن معاً. تستخدم تعليمات SIMD بشكل واسع لمعالجة الرسوميات الثلاثية الأبعاد، كما أخذت بطاقات الإظهار الحديثة، بتعليماتها SIMD المضمنة، هذه المهمة بشكل كامل عوضاً عن وحدة المعالجة المركزية CPU.

1.1.3 منهجيات التواصل Communication Methodologies

يعدّ نقل المعطيات بين النوى المتوازية parallel cores العمود الفقري للمعالجة المتوازية، لأنه يتطلب في معظم الحالات نقل المعطيات من نواة إلى أخرى، وهذا الأمر يكون أكثر تكراراً more frequent إذا كانت عناصر المعطيات elements of data مترابطة interdependent. وفي الواقع، فإن كفاءة

الخوارزمية المتوازية parallel algorithm مقيّدة وتعتمد على الزمن المستغرق لتواصل communicate الأجزاء المختلفة من المعطيات بين عناصر المعالجة processing elements.

وأحياناً يكون الزمن المستغرق لتواصل المعطيات بين عناصر المعالجة أكبر من الزمن المستغرق للحساب المتوازي parallel calculation للنتائج. وفي مثل هذه الحالة تسقط مزايا الخوارزمية المتوازية كلياً. لذا أصبح من الضروري تصميم تقنيات مناسبة suitable techniques لتواصل المعطيات وتحسين مقدار المعطيات المتدفقة data flow بين نوى المعالجة processing cores بالشكل الأمثل optimized.

وهناك تقنيتان لتواصل المعطيات بين عناصر المعالجة المختلفة [18] [16]:

أ- آلية فضاء العنوان المشترك Shared Address Space Mechanism:

يوجد في هذه الآلية ذاكرة مشتركة يمكن النفاذ إليها accessible من قبل جميع نوى المعالجة processing cores. ويمكن النفاذ لفضاء العنوان المشترك بشكل منتظم Uniformly أو غير منتظم non-uniformly. ففي نفاذ الذاكرة المنتظم (UMA) Uniform Memory Access يكون زمن النفاذ إلى أي موقع في الذاكرة ثابت، في حين يختلف زمن النفاذ إلى مواقع الذاكرة باختلاف المواقع الذي يستغرقه المعالج للنفاذ إلى موقع في الذاكرة في نفاذ الذاكرة غير المنتظم Non Uniform Memory Access (NUMA). وكلا الذاكرتين متساويتين بصلاحيات النفاذ لأي موقع ذاكرة.

ب- منصة تمرير الرسائل Message Passing Platform:

يوجد لكل معالج في هذه الآلية ذاكرته الخاصة، وفضاء عناوينه الخاص، ويتحقق التفاعل interaction بين نوى المعالجة من خلال تبادل الرسائل بينها، وتستخدم واجهة تمرير الرسائل لدعم مثل هذه التفاعلات. يستخدم تبادل الرسائل لنقل المعطيات، والعمل ومزامنة الحركة Synchronize action بين المعالجات. وبشكل عام تدعم نماذج تمرير الرسالة message passing paradigms تنفيذ برنامج مختلف في كل عقدة P من عقده. والعمليات الأساسية في هذا النموذج البرمجي هي ارسال واستقبال الرسائل. كما ينبغي وجود طريقة method لإسناد تعريف هوية فريد unique identification لكل من المعالجات المشغلة للتطبيق المتوازي parallel application.

1. 2. أنماط البرمجة المتوازية Parallel programming patterns

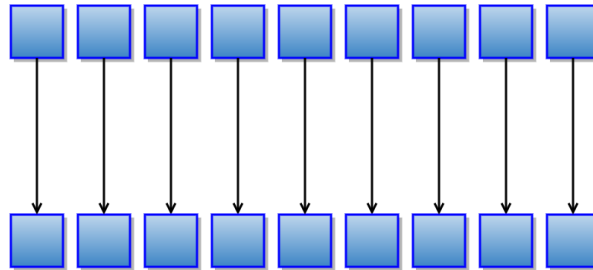
1. 2. 1. نظرة عامة Overview

سيتم الحديث في هذه الفقرة عن أنماط البرمجة المتوازية. لأن وحدة معالجة الرسوميات هي معالج ذو تعليمات عالية التوازي، والحوسبة المتوازية سواء كانت باستخدام وحدة معالجة الرسوميات أو نوى متعددة من وحدة المعالجة المركزية CPU أو مجموعة من العناقيد Clusters فهي تخضع لمجموعة من الأنماط التي تستخدم للاستفادة بشكل فعال من المعماريات المتوازية.

1. 2. 2. المقابلة Map

تعدُّ المقابلة أبسط أنماط الحوسبة المتوازية. إذ تقوم بتطبيق تابع على صفيفة الدخل مع تكرار هذا التابع على كل عنصر من عناصر الصفيفة كما في الشكل 1-0 ويمكن تمثيلها بالعلاقة الرياضية

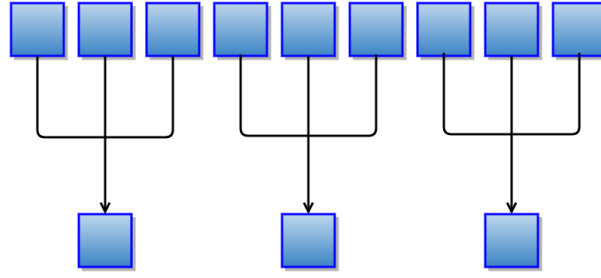
$$x_i = f(x_i) \quad (2-1)$$



الشكل 1-0 نمط المقابلة Map

1. 2. 3. الجمع Gather

وهو تطبيق تابع على قيم مختارة اعتباطياً من الدخل للحصول على قيمة خرج كما هو مبين في الشكل 2-0



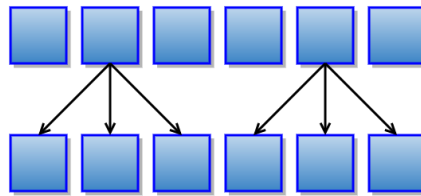
الشكل 2-0 نمط الجمع Gather

ويمكن تمثيلها رياضياً كتابع بعدة متحولات

$$y_i = f(a_m, b_n, c_p, \dots) \quad (3-1)$$

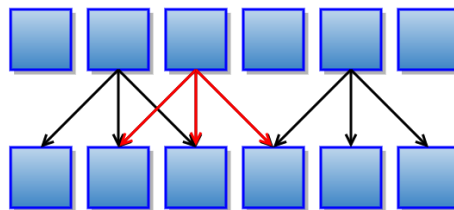
1. 2. 4. التبعثر Scatter

وهو عكس نمط الجمع Gather، فإذا كان الجمع يقوم على حساب قيمة وحيدة من عدة قيم أخرى، فيقوم التبعثر بالعملية العكسية وهي حساب عدة قيم من قيمة وحيدة كما هو مبين في الشكل 3-0



الشكل 3-0 نمط التبعثر Scatter

وهنا يجب الانتباه في حالة وجود تصادم بين النياسب thread collisions أي عندما تقوم عدة نياسب بالكتابة على نفس الأماكن كما هو موضح في الشكل 4-0

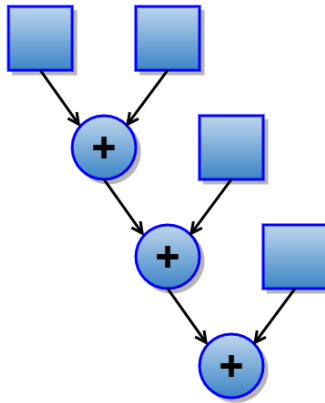


الشكل 4-0 التداخل بين النياسب thread collision

1.2.5. التخليض Reduce

هي عملية تتم بين أزواج من العناصر المعطاة لتقليل عدد العناصر ضمن المجموعة الواحدة، والعملية النموذجية هي الجمع المتوازي parallel sum. على فرض أنه يراد حساب مجموع جميع العناصر ضمن صيغة حيث توصف بالمعادلة $\sum_{i=1}^n x_i$ ، يمكن التعبير عنها كآتي:

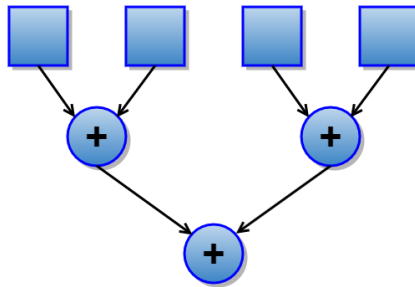
$$((x_1 + x_2) + x_3) + \dots \quad (4-1)$$



الشكل 5-0 نمط التخليض Reduce قبل التجزئة

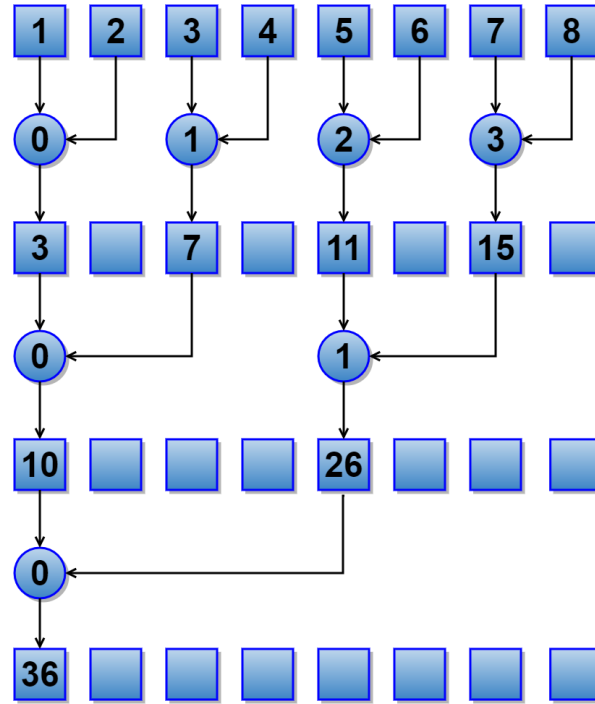
وبما أن الجمع هو عملية ترابطية associative عندها يمكن تجزئة فضاء المسألة كما يلي

$$\sum_{i=1}^n x_i = \sum_{i=1}^{n/2} x_i + \sum_{i=n/2}^n x_i = \dots \quad (5-1)$$



الشكل 6-0 نمط التخليض Reduce بعد التجزئة

يوضح الشكل 7-0 تطبيقاً عملياً على عملية التخليض بشكل متوازي



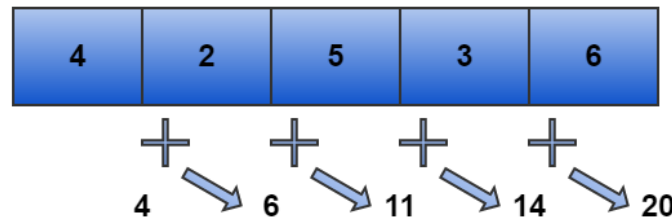
الشكل 7-0 التخفيض المتوازي Parallel Reduce

1. 2. 6. المسح Scan

حيث يتم حساب كل قيمة خرج كتاب مرتبط بجميع المدخلات السابقة من 1 إلى n أي

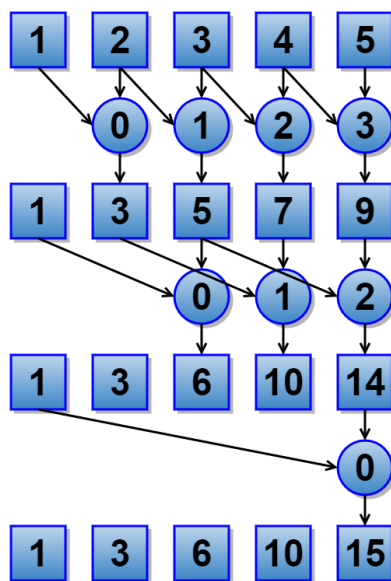
$$y_n = f(x_1, x_2, \dots, x_n) \quad (6-1)$$

يلاحظ من الشكل 8-0 استخدام المسح للقيام بعملية الجمع حيث تبدو العملية تتابعية (مثل عملية التخفيض Reduce).



الشكل 8-0 نمط المسح Scan

ويظهر الشكل 9-0 المسح المتوازي حيث يحتاج إلى $N-1$ نيسب لتنفيذه كما يلاحظ تضاعف حجم الخطوة عند كل مرحلة.



الشكل 9-0 المسح المتوازي Parallel scan

الفصل الثاني

معمارية وحدة معالجة الرسومات

GPU Architecture

مقدمة

أدى النمو الهائل في تقدم رسومات الحاسوب إلى ثورة في بطاقات الرسومات graphics cards عالية الأداء والقادرة على إجراء العمليات الحسابية بالفاصلة العائمة بدقة عالية high precision floating point arithmetic وعمليات التصيير المرئي visual rendering. تستخدم بطاقات الرسومات من أجل إنشاء سريع للأطر rapid creation of frames لعرضها على الشاشة المرئية، وتستخدم عموماً في أجهزة الحواسيب الشخصية personal computers، الهواتف النقالة الذكية Smart Phones ومنصات الألعاب Gaming Consoles. يحتاج إنشاء هذه الأطر إلى حوسبة ذو كثافة عالية highly computation intensive وفقاً لمتطلبات الإظهار في العصر الحديث خصوصاً في مجال ألعاب الفيديو video games حيث يعتمد الإطار التالي المراد عرضه على تفاعل معقد بالزمن الحقيقي complex real time interaction من قبل المستخدم مع اللعبة وأيضاً ضمن عرض الفيديو video display أثناء تنفيذ عمليات الترميز وفك الترميز المنطقية المعقدة في أثناء التشغيل. تم صياغة مصطلح وحدة معالجة الرسومات Graphics Processing Unit (GPU) من قبل شركة NVIDIA والتي استخدمته بدورها لجعل بطاقتها الرسومية أكثر شعبية.

كما لفتت، القوة الحسابية العالية لبطاقات الرسومات الحديثة، انتباه الباحثين على مختلف مجالاتهم الذين يحاولون تسريع تطبيقاتهم بغية الحصول على نتائج سريعة لاختبارات أبحاثهم. ويشار إلى مثل هذه البطاقات بوحدات معالجة الرسومات عامة الغرض General Purpose Graphics Processing Units (GPGPUs). كما قادت بعض الباحثين لمحاولة تسريع تطبيقاتهم من أجل زيادة كفاءة خوارزمياتهم، وذلك بإنشاء إصدار GPU لخوارزمياتهم، ويعدّ مجال معالجة الصور image processing واحداً من أهم المجالات المستفيدة من هذه ثورة التقنية بالإضافة إلى مجالات القيادة الذاتية للسيارات Self-driving cars وتقنيات كشف الكائنات objects detection والتعلم العميق Deep Learning.

2. 1. معمارية وحدة معالجة الرسومات GPU Architecture:

تعدُّ وحدات معالجة الرسومات (GPUs) Graphical Processing Units أجهزة ذات توازٍ عالٍ highly parallel متعددة النياسب multithreaded قابلة للبرمجة programmable، لها القدرة على تصيير تطبيقات الرسومات بالزمن الحقيقي. وتملك بين العشرات إلى المئات وأحياناً الآلاف من النوى بقدرة كبيرة على إجراء العمليات الحسابية بالفاصلة العائمة بدقة عالية، والتي تعدُّ أحد متطلبات معالجة الفيديو بالزمن الحقيقي real-time video processing. وتمتاز بطاقات الرسومات بخاصية هامة أخرى وهي عرض حزمة عالية جداً للذاكرة very high memory bandwidth والتي لا تقل عن 64-bit في المعالجات الحديثة بالإضافة إلى احتوائها على عدد كبير من المسجلات ضمن الشريحة on chip registers. يسمح عرض الحزمة العالي بنقل مقادير كبيرة من المعطيات ضمن الدفق الوحيد single flow بينما يوفر عدد السجلات الكبيرة من مسك hold عدة قيم للمتغير several variable values أثناء العملية الحسابية [18].

إن عدد العمليات بالفاصلة العائمة بالثانية floating point operations per second (FLOPS) لوحدة معالجة الرسومات GPU أكبر بكثير (أسياً) مقارنةً بعدد العمليات الموجودة في وحدة المعالجة المركزية الراقية high end CPU. فبينما تمتلك وحدة المعالجة المركزية الراقية سرعة حوسبة تقدر ببضع عشرات من الـ gigaflops، تمتلك وحدة معالجة الرسومات العادية سرعة حوسبة تقدر ببضع مئات من الـ gigaflops. وتقدر سرعة حوسبة وحدة معالجة الرسومات الراقية ببضع الـ teraflops. ويعود السبب لهذا الفارق الكبير إلى أنَّ وحدات معالجة الرسومات مخصصة بشكل عالٍ لأداء المهام كثيفة الحوسبة وعالية التوازي compute intensive highly parallel tasks مثل تصيير الرسومات graphics rendering. وهكذا فهي مصممة بحيث يتم تسخير العدد الهائل من الترانزستورات لمعالجة المعطيات data processing بدلاً من التحكم بالتدفق flow control وتخزين المعطيات data caching.

وبالتالي فإن وحدات معالجة الرسومات مناسبة للمسائل problems التي يمكن اعتبارها كتطبيقات معطيات متوازية data parallel applications، على سبيل المثال، عند استخدام نفس البرنامج على مجموعات مختلفة من المعطيات وتكون العمليات الحسابية arithmetic operations أكثر بكثير من عمليات الذاكرة memory operations. عندها يحتاج تدفق المعطيات من معالج إلى آخر إلى متطلبات صغيرة جداً طالما يتم تنفيذ البرنامج نفسه على مجموعة معطيات مختلفة وعناصر معالجة مختلفة [18].

يمكن لأي عملية تعتمد على حزمة معطيات كبيرة أن تستخدم نموذج البرمجة المتوازية parallel programming model لتسريع التطبيق. ففي تطبيقات تصيير الصور ثلاثية الأبعاد 3D image rendering مثل برامج ألعاب الحاسوب يتم مقابلة mapping عدد هائل من البيكسلات إلى نوى المعالجة من أجل المعالجة المتوازية المستقلة independent parallel processing. وبشكل مماثل تقابل تطبيقات ترميز وفك ترميز الفيديو مجموعة من المعطيات مثل كتلة صورة معينة إلى معالج مستقل للتنفيذ [18].

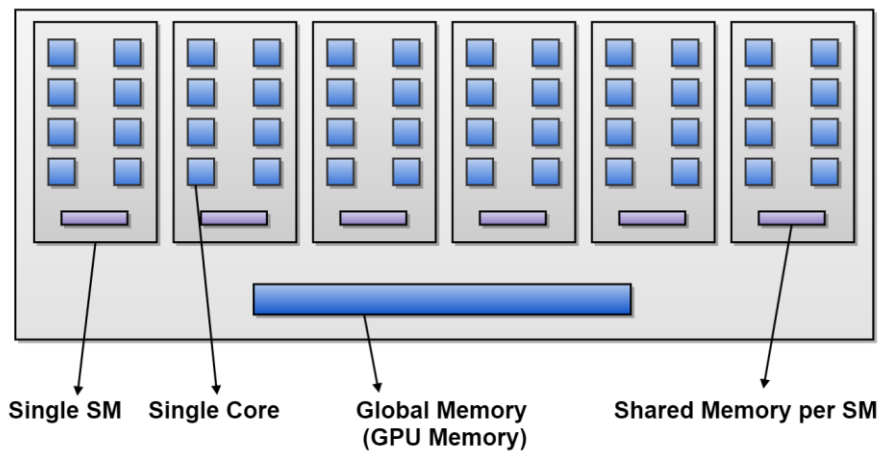
2. 1. 1. تنظيم ذاكرة وحدة معالجة الرسومات GPU Memory

:Organization

تتألف بطاقة الرسومات من دائرة مطبوعة printed circuit board مثلها مثل أي لوحة أم motherboard عادية تحتوي على معالج وذاكرة نفاذ عشوائي RAM. وتمتلك نظام إدخال إخراج قاعدي BIOS مخصص عادةً لإعدادات التهيئة initial settings لوحدة معالجة الرسومات Graphical Processing Unit (GPU). كما تلعب ذاكرة النفاذ العشوائي RAM دور دارئ الإطار frame buffer والذي يعني مسك الإطار المحسوب بالذاكرة قبل عرضه حيث تحتاج عملية الحوسبة لكل إطار لتطبيق في الزمن الحقيقي باستهلاك حوسبة كثيف.

تجزأ وحدات معالجة الرسومات Graphical Processing Units (GPUs) من شركة NVidia إلى مجموعة من معالجات تدفق متعددة Streaming Multiprocessors (SMs) كما هو مبين في الشكل 0-1. ويتكون كل معالج تدفق متعدد عادةً من 8 نوى cores، فإذا كان عدد النوى في وحدة معالجة الرسومات Graphical Processing Unit (GPU) 96 نواة يكون عدد معالجات التدفق المنظم بها هو 12 معالج تدفق متعدد كل منها مؤلف من 8 نوى. وهذا يعتمد على ضبط configuration وحدة معالجة الرسومات Graphical Processing Unit (GPU) ويمكن أن تحتوي على عدد أكبر من النوى cores بكل معالج تدفق متعدد Streaming Multiprocessor إذا احتوت وحدة معالجة الرسومات Graphical Processing Unit (GPU) على عدد كبير من النوى cores. وليس للبرامج التي يتم تشغيلها على وحدة معالجة الرسومات أي علاقة بالاختلافات المعمارية architectural differences، وهذا بدوره يجعل برمجة وحدة معالجة الرسومات Graphical Processing Unit (GPU) قابلة للتوسع scalable.

يقوم معالج التدفق المتعدد (SM) Streaming Multiprocessor بجدولة schedules التعليمات حول النوى cores. حيث يمكن جدولة 32 نيسب معاً كحد أعظمي ضمن معالج التدفق المتعدد وتدعى بجدولة الحزمة warp scheduling. تحتوي كل نواة ضمن معالج التدفق المتعدد على ذاكرتها الخاصة من مستوى المسجل register level memory والتي تكون متاحة للنيسب thread الوحيد أثناء تنفيذه على النواة، بالإضافة إلى وجود ذاكرة مشتركة Shared Memory لمعالج التدفق المتعدد والتي تكون متاحة لجميع النياسب threads التي يتم تنفيذه عليها. إن جميع النياسب threads المشغلة على معالج تدفق متعدد، والتي يجب أن تتعاون cooperate فيما بينها تستخدم هذا الجزء من الذاكرة للتواصل.



الشكل 0-1 معمارية GPU مكون من 96 نواة، كل SM تحوي 8 نوى

ينفذ كل معالج تدفق متعدد مجموعة من النياسب المتعاونة cooperating threads. ويوجد أيضاً ذاكرة شمولية Global Memory تُعرّف عموماً بذاكرة وحدة معالجة الرسومات GPU Memory التي تكون متاحة لكل وحدة معالجة الرسومات (GPU) Graphical Processing Unit، وتستخدم بشكل عام لمسك hold المعطيات المنسوخة من وحدة المعالجة المركزية والمسماة بالمضيف (CPU) host إلى وحدة معالجة الرسومات والمسماة بالتجهيزة (GPU) device بالإضافة إلى النتائج البينية قبل نقلها إلى وحدة المعالجة المركزية CPU. في معالجة الفيديو، تقوم هذه الذاكرة بمسك الإطار التالي المراد إظهاره بعد عملية الحوسبة. وتشبه الذاكرة الشمولية Global Memory ذاكرة النفاذ العشوائي RAM المتاحة لعمل وحدة المعالجة المركزية CPU. وتتم جميع عمليات التواصل بين وحدة المعالجة المركزية CPU ووحدة معالجة الرسومات (GPU) Graphical Processing Unit عن طريق هذه الذاكرة.

2.1. القيود في استخدامات وحدة معالجة الرسومات كتهيزة للأغراض

العامية GPGPU usage limitations

عند استخدام وحدة معالجة الرسومات GPU كأداة حوسبة عامة الغرض تقيد بإمكانية معالجة التجزيئات المستقلة independent fragments من المعطيات فقط، وتكمن الفائدة من ذلك أنه يمكن معالجة العديد من التجزيئات المتماثلة بشكل متوازي. ولذلك تعرف وحدات معالجة الرسومات Graphical Processing Units (GPUs) كمعالجات تدفق stream processors طالما يمكنها تشغيل نفس العملية operation على التدفقات المتعددة من المعطيات وبالتالي يمكن استخدامها فقط لعمليات SIMD.

2.2. عناصر معمارية تهيزة الحساب الموحد CUDA Elements

معمارية تهيزة الحساب الموحد Compute Unified Device Architecture أو اختصاراً CUDA هي تقنية خاصة تم تقديمها لأول مرة عام 2006 من قبل شركة NVIDIA. CUDA هي منصة حوسبة متوازية parallel computing platform ونموذج برمجة programming model يستخدم محرك حوسبة متوازي Parallel Compute Engine لإدارة النياسب threads المشغلة على وحدة معالجة الرسومات Graphical Processing Unit (GPU). حيث أن مهمة محرك الحوسبة المتوازي هي جدولة schedule النياسب لكي تشغل run على وحدة معالجة الرسومات Graphical Processing Unit (GPU) ومطابقتها على النوى المتاحة، وبهذا توافق conforms معمارية ومواصفات CUDA [18].

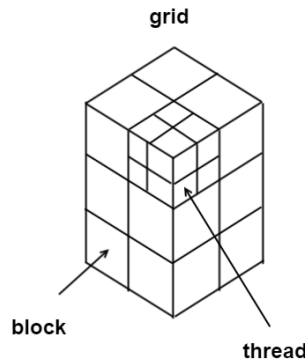
تم تصميم معمارية منصة CUDA بطريقة تسمح بدعم عدة لغات برمجة وواجهات البرمجية للتطبيقات APIs. كما تقدم CUDA الدعم للغات البرمجة عالية المستوى مثل C، C++، FORTRAN، DirectCompute و OpenACC، كما تتوفر مغلفات للغات برمجة أخرى مثل Python و Java من قبل طرف ثالث [18].

2.2.1. كتل البناء Building Blocks

يعدّ النيسب thread أصغر وحدة في برنامج الـ CUDA ويخدم كعنصر بناء ضمنه. ويطلق على مجموعة من النياسب threads مصطلح كتلة block، وبشكل مماثل يطلق على مجموعة من الكتل مصطلح الشبكة grid الشكل 2-0. وتقع على عاتق المبرمج مهمة تحديد عدد الكتل اللازم لإنشاءها لتشغيل التطبيق المتوازي. ويقوم المبرمج بتحديد العدد الأعظمي من النياسب threads الممكن إنشاءها ضمن كل كتلة block.

$$(1-2) \quad \text{عدد الكتل} = \frac{\text{العدد الكلي من النياسب}}{\text{عدد النياسب ضمن الكتلة الواحدة}}$$

يوجد قيد على عدد النياسب الممكن إضافتها ضمن الكتلة وذلك بسبب الحجم المحدود من الذاكرة المشتركة Shared memory المتاحة لكل معالج تدفق متعدد SM. بالنسبة للجيل الحالي من وحدات معالجة الرسومات GPUs فإن العدد الأعظمي من النياسب التي يمكن إنشاؤها ضمن الكتلة الواحدة ليتم تنفيذها على معالج تدفق متعدد SM وحيد هو 1024 نيسب. وطالما أن النياسب المتعاونة cooperating threads تستخدم الذاكرة المشتركة يصبح هذا التقييد واضحاً.



الشكل 2-0 كتل بناء CUDA

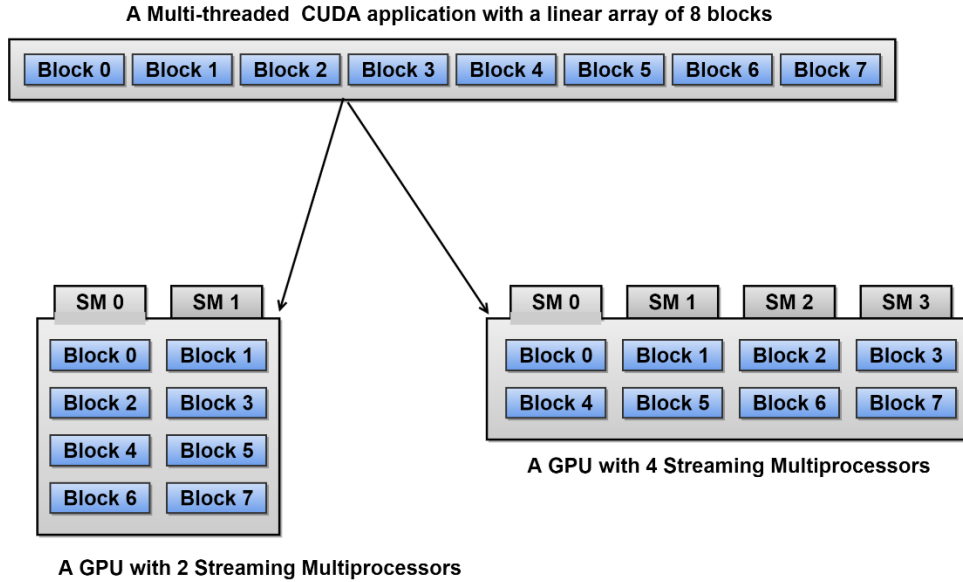
يقوم محرك الحوسبة CUDA بمطابقة maps الكتل إلى معالجات التدفق المتعدد المتاحة، مع وجود تقييد آخر على هذا الأمر، حيث أنه ينبغي جدولة جميع النياسب ضمن الكتلة الواحدة لتشغل على نفس معالج التدفق المتعدد. فإذا كان عدد النياسب ضمن الكتلة أعلى من العدد الكلي من معالجات التدفق SP الموجودة ضمن المعالج متعدد التدفق SM، عندها يتم في البداية تمرير عدد من النياسب يكافئ عدد المعالجات الموجودة ضمن معالج التدفق المتعدد SM لجدولتها، وتنتظر النياسب المتبقية انتهاء تنفيذ الدفعة الحالية من النياسب. وبهذه الطريقة تنهي جميع النياسب تنفيذها [18].

2.2. قابلية التوسع لتطبيق CUDA

Scalability of a CUDA application

تمتاز برامج CUDA بقابلية عالية للتوسع highly scalable لدرجة أنها تغني المبرمج عن التفكير في عملية تمثيل optimizing البرنامج حسب وحدة معالجة الرسومات GPU الذي يقوم باستخدامه، ويمكن لنفس البرنامج أن يعمل بنجاح على أي GPU مهما كان عدد من النوى cores. حيث يقع على عاتق

محرك حوسبة وحدة معالجة الرسومات GPU Compute Engine جدول النياسب threads على التجهيزة device بشكل مناسب لمعمارية معطاة. يبين الشكل 0-3 عملية مطابقة mapping لثمان كتل من النياسب على وحدتي معالجة رسومات مختلفتين، الأولى بمعالجين متعددي التدفق SMs والآخر بأربعة معالجات متعددة التدفق SMs.



[19]

الشكل 0-3 قابلية التوسع لتطبيق CUDA

تسمح قابلية التوسع لتطبيقات CUDA بتشغيل نفس التطبيق على منتجات مختلفة من شركة NVidia بدءاً من وحدات معالجة الرسومات للاستخدام العادي نوع GeForce مروراً بالمنتجات المضمنة النقلة Tegra إلى المنتجات الاحترافية مثل Quadro و Tesla. ويتم هذا الأمر ببساطة عن طريق توسعة scaling عدد معالجات التدفق المتعدد وأجزاء الذاكرة memory partitions. وهكذا فإن وحدة معالجات رسومات GPU بعدد كبير من معالجات تدفق متعدد ستقوم تلقائياً بتشغيل برنامج بوقت أقل مقارنة بوحدة معالجة رسومات GPU بعدد أقل من معالجات التدفق المتعدد.

2.2. عناصر تطبيق CUDA Elements of a CUDA application

كما تم ذكره سابقاً، يسمح نموذج CUDA باستخدام عدة لغات برمجة عالية المستوى لتطوير تطبيق متوازي. عندما يتم استخدام لغة ال C، عندها تعتبر برامج CUDA بشكل أساسي برامج لغة C مع مجموعة من الواجهات البرمجية للتطبيقات APIs ومكتبة توابع library functions، وتبقى القواعد البرمجية programming syntax ذاتها كما لو كان برنامج بلغة ال C.

تبنى برامج CUDA حول توابع خاصة تسمى النوى Kernels. عند استدعاء توابع النواة kernel functions، تُنفذ عدد N من المرات على التوازي. ويتم حمل كل N تنفيذ للنواة باستخدام نيسب منفصل، والذي يكون بدوره تابع لأحد الكتل المنشأة من قبل البرنامج والمجدولة ليتم تشغيلها على أحد من معالجات التدفق المتعدد [18].

يتم تعريف نواة CUDA كما يلي:

```
__global__ void kernel_function_name (parameter list)
```

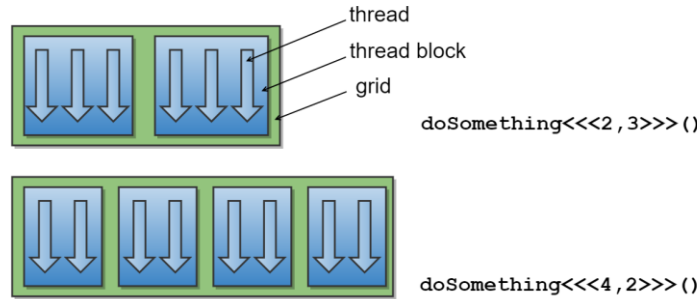
تشير الكلمة المفتاحية `__global__` إلى أن الرمز code يعمل على التجهيزة device وقد تم استدعاءه من رماز المضيف host code. يتم إعطاء كل نيسب thread يقوم بتنفيذ نواة kernel محدد تعريف فريد ID يسمى `threadIdx`، تكون قيمته متاحة من خلال النواة kernel. تتكوّن قيمة متغير `threadIdx` من ثلاث عناصر بحيث يمكن ترتيب النياسب على طول صفيقة أحادية البعد single dimensional array أو لصفيقة ذات بعدين أو ثلاثة. فمن أجل ترتيب arrange صفيقة أحادية البعد، يتم تحديد النيسب من خلال المتغير `threadIdx.x`، ومن أجل ترتيب الصفيقة ثنائية البعد يتم تحديده من خلال `threadIdx.x` و `threadIdx.y`، أما الصفيقة ثلاثية البعد فيتم حساب عنوان النيسب thread index من خلال `threadIdx.x`، `threadIdx.y` و `threadIdx.z`. وهذا يساعد في تبسيط منطق البرنامج أثناء التعامل مع الأشعة vectors والمصفوفات matrix والحجوم volumes [18].

تدير CUDA النياسب threads والكتل blocks بشكل صفيقة أحادية أو ثنائية أو ثلاثية البعد، فكما تم تنظيم النياسب threads، يتم اتباع نفس السلوك في تنظيم الكتل blocks ضمن شبكة grid أحادية البعد أو ثنائية البعد أو ثلاثية البعد من كتل النياسب thread blocks. ويتم بشكل مشابه تحديد الكتل blocks ضمن الشبكة grid من خلال شعاع ثلاثي العناصر three component vector مخصص بقيمة المتغير `blockIdx` والذي له ثلاثة عناصر `blockIdx.x` و `blockIdx.y` و `blockIdx.z`.

تُتاح أبعاد كتلة النيسب thread block، أو بمعنى آخر عدد النياسب threads ضمن الكتلة الواحدة، ضمن المتغير المعرّف مسبقاً `blockDim` كقيمة ثلاثية العناصر. وتعطى التركيبة لاستدعاء نواة CUDA، والمسمّاة بإطلاق النواة kernel launch، كما يلي:

```
kernel_function_name<<<NUM_BLOCKS, THREADS_PERBLOCKS>>>(parameter list);
```

يحدد المتغير NUM_BLOCKS عدد الكتل الكلي اللازم إنشاءه من أجل التطبيق بينما يحدد المتغير THREADS_PER_BLOCK عدد النياسب threads الكلي اللازم جدولتها ضمن الكتلة block كما هو موضح في الشكل 4-0.



الشكل 4-0 نموذج التنفيذ Execution model

يتم في برمجة CUDA تسمية وحدة المعالجة المركزية CPU وذاكرة النفاذ العشوائي RAM باسم المضيف host، ويتم تسمية وحدة معالجة الرسومات GPU بالتجهيزة device. ويتألف أي برنامج CUDA من الخطوات التالية:

- تخصيص ذاكرة المضيف من أجل المتغيرات المطلوبة Host Memory allocation for the required variables
- تخصيص ذاكرة التجهيزة من أجل المتغيرات Device Memory allocation for the variables
- نسخ القيم من المضيف إلى التجهيزة Copying the values from host to device
- حوسبة النتائج على التجهيزة وذلك باستخدام تنفيذ النياسب المتوازي مع مساعدة نوى CUDA Computation of the results on the device by the use of parallel threads executing with the help of CUDA kernels.
- نسخ القيم المحسوبة مرة ثانية إلى المضيف Copying the computed values back to the host

الفصل الثالث

معمارية وحدة NEON

مقدمة

تقنية NEON هي عبارة عن معمارية SIMD 128-bit كتوسعة لسلسلة معالجات ARM Cortex-A من شركة ARM، صممت من أجل تسريع تطبيقات الوسائط المتعددة الاستهلاكية consumer multimedia applications بشكل عالٍ ومرن مقدمةً بذلك تجربة محسنة الأداء بشكل ملحوظ للمستخدم. تمتلك معمارية NEON 32 مسجل بعرض 64-bit (أو 16 مسجل بعرض 128-bit بنمط الرؤية المزدوجة dual view).

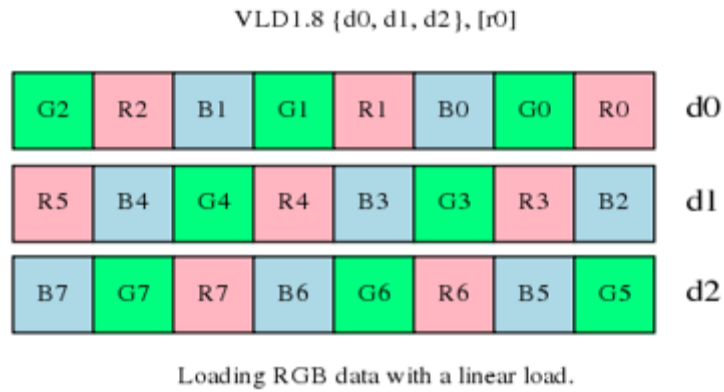
تستخدم تقنية NEON بشكل واسع في الوسائط المتعددة multimedia وخوارزميات معالجة الإشارة الرقمية مثل ترميز وفك ترميز الفيديو، الرسوميات ثنائية وثلاثية الأبعاد، الألعاب، الصوتيات ومعالجة الكلام، معالجة الصور، الاتصالات الهاتفية وتركيب الصوت sound synthesis [19].

يعرض هذا الفصل شرح مبادئ البرمجة باستخدام معمارية NEON والتعامل معها بكفاءة عالية، تبيان الملاحظات الهامة للبرمجة بشكل احترافي، وفي البداية سيتم التطرق إلى تعليمات الذاكرة وكيفية الاستخدام المرن لعمليات التحميل load والتخزين store عن طريق التعليمات البديلة.

3.1. التحميل والتخزين Load and Stores

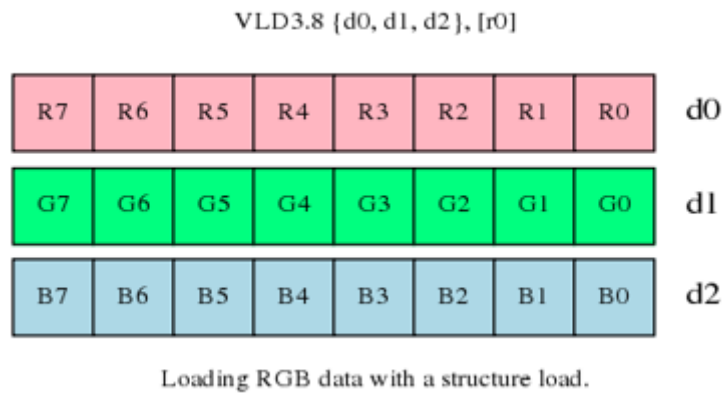
لفهم المشكلة المطروحة سيتم البدء بمثال عملي ملموس، على فرض أنه يوجد لدينا صورة ملونة بنمط 24 بت RGB، حيث أن كل البيكسلات مرتبة في الذاكرة بشكل R, G, B, R, G, B, ... وكان المطلوب تنفيذ عملية معالجة بسيطة مثل التبديل بين قناتي اللونين الأحمر والأزرق. عندئذ ما هي الطريقة الممكنة لتنفيذ المطلوب بكفاءة باستخدام NEON؟

إن استخدام التحميل load العادي الذي يقوم بجلب معطيات RGB بشكل خطي من الذاكرة إلى المسجلات يجعل عملية تبديل اللونين الأحمر والأزرق غير ملائمة. الشكل 0-1، يؤدي إلى جعل الرماز اللازم لنقل القنوات المعتمد على هذه الدخل غير أنيق من ناحية استخدام الأقنعة masks، والإزاحة shifting والضم combining. وهذا الأمر لا يعدّ ذو كفاءة على الإطلاق من الناحية البرمجية.



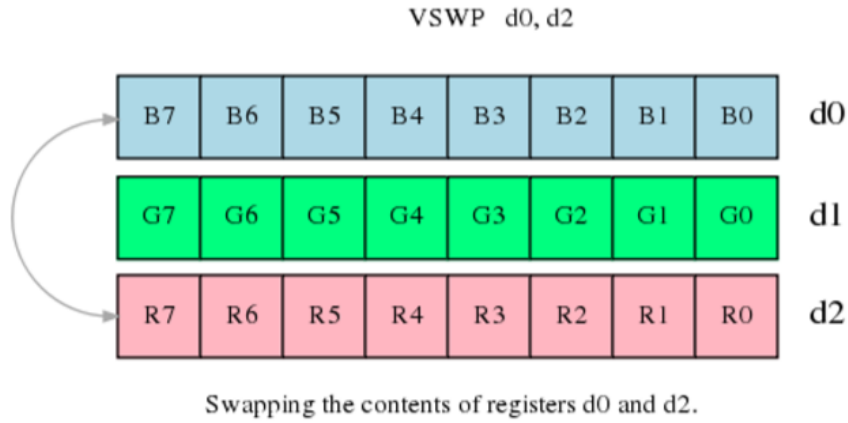
الشكل 1-0 تحميل معطيات RGB باستخدام التحميل الخطي

تؤمن معمارية NEON بنية تعليمات تحميل وتخزين تساعد في مثل الحالات السابقة. حيث تقوم بجلب المعطيات من الذاكرة وتوصل في الوقت نفسه القيم إلى مسجلات مختلفة. على سبيل المثال يمكن استخدام التعليمة VLD3 لفصل الأحمر، والأخضر والأزرق أثناء تحميلهم. الشكل 2-0



الشكل 2-0 تحميل معطيات RGB باستخدام التحميل البنيوي

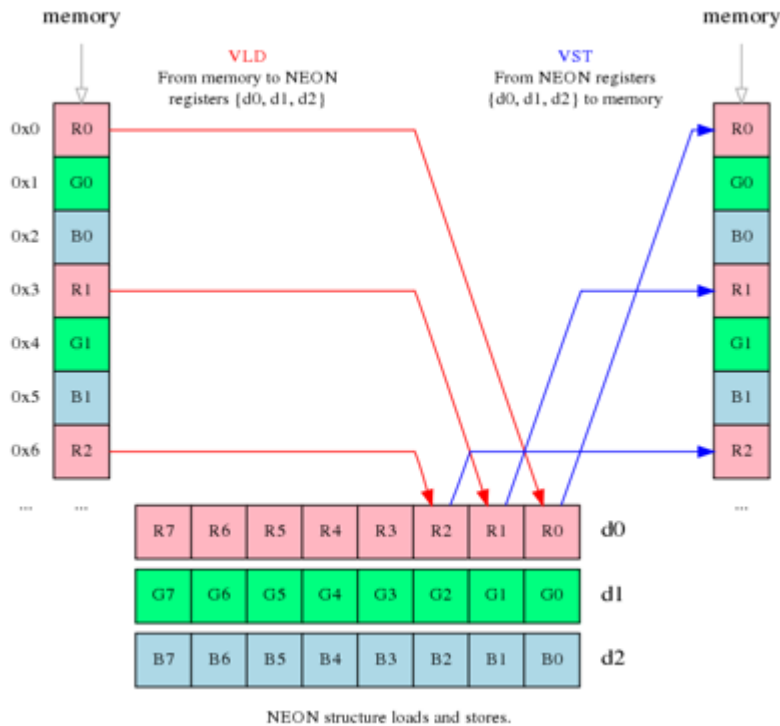
بعد ذلك تستخدم التعليمة VSWP d0, d2 كما هو موضح بالشكل 3-0 للتبديل بين مسجلات اللونين الأحمر والأزرق ثم تتم كتابة المعطيات مرة ثانية إلى الذاكرة، مع إعادة التشابك re-interleaving، باستخدام تعليمة مشابهة اسمها VST3.



الشكل 3-0 تبديل محتويات المسجلين d0 و d2

3.1.1. مبدأ عمل إلغاء التشابك de-interleave والتشابك interleave

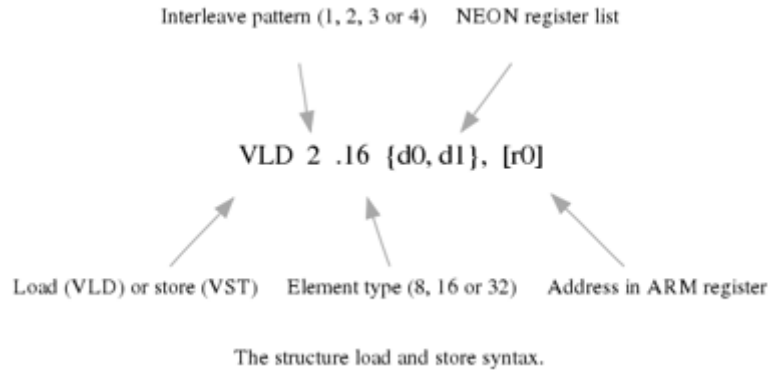
تقوم تعليمات التحميل ببنية NEON بقراءة المعطيات من الذاكرة إلى مسجلات NEON بعرض 64-bit، مع إلغاء التشابك de-interleaving إذا لزم الأمر. كما تعمل تعليمات التخزين بشكل مشابه مع إعادة تشابك re-interleaving المعطيات من المسجلات قبل كتابتها في الذاكرة كما هو مبين بالشكل 4-0



الشكل 4-0 تحميلات وتخزينات NEON البنيوية

أ- تركيب التعليمات Syntax

تتكون تعليمات التحميل والتخزين NEON البنيوية من خمسة أجزاء. الشكل 5-0



الشكل 5-0 صيغة تعليمية التحميل و التخزين البنيوية

- اسم التعليمات المستعار **instruction mnemonic** وهو إما VLD للتحميل أو VST للتخزين stores.
- الشكل النمطي للتشابك **interleave pattern** (عددي) وهو عدد من المسجلات من أجل التشابك. وهو أيضاً الفاصل بين العناصر المتعلقة ضمن كل بنية.
- نوع العنصر **Element type** يحدد عدد البتات ضمن العناصر المولج إليها.
- قائمة من مسجلات 64-bit NEON للتحميل من الذاكرة أو للكتابة إليها. ويمكن لهذه القائمة أن تحوي لغاية أربعة مسجلات وهذا يعتمد على الشكل النمطي للتشابك.
- عنوان مسجل **ARM address register** يحتوي على الموقع اللازم النفاذ إليه في الذاكرة من أجل التخزين إليه أو التحميل منه. ومن الممكن تحديث العنوان بعد النفاذ.

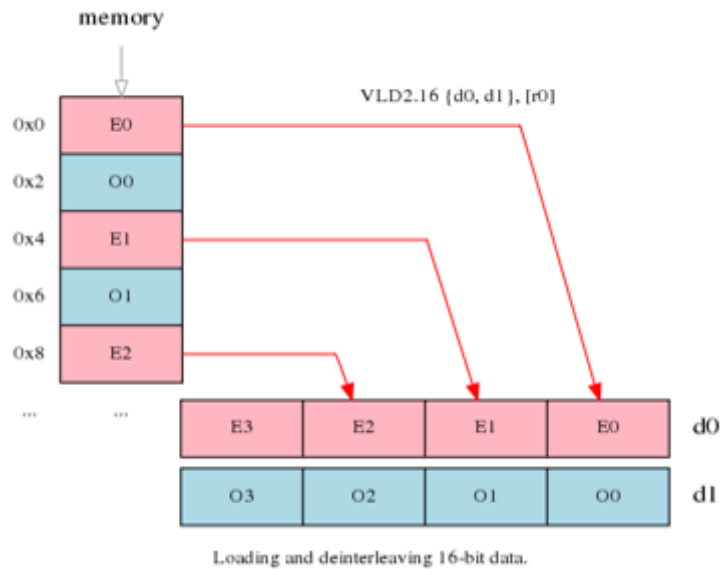
ب- الشكل النمطي للتشابك Interleave Pattern

تتاح تعليمات NEON من أجل بنى التحميل load، التخزين store، التشابك interleave، وإلغاء التشابك de-interleave والتي تحتوي على عدد من العناصر elements المتساوية يتراوح بين عنصر واحد إلى أربعة عناصر. ويمكن أن يكون حجم العنصر 8-bit، 16-bit، أو 32-bit.

- **VLD1** و هي أبسط صيغة، حيث تقوم بتحميل من مسجل إلى أربع مسجلات بالمعطيات من الذاكرة، بدون أي إلغاء للتشابك de-interleaving. تستخدم عندما يراد معالجة مصفوفة من المعطيات غير المتشابكة.
- **VLD2** تقوم بتحميل مسجلين أو أربع مسجلات بالمعطيات. وتقوم بإلغاء تشابك العناصر الزوجية والفردية ضمن هذه المسجلات. تستخدم لفصل معطيات الصوت المجسم Stereo audio إلى قناتين يمينى ويسرى.
- **VLD3** تقوم بعمليات التحميل وإلغاء التشابك ضمن ثلاث مسجلات. وهي مفيدة لفصل بيكسلات الألوان RGB إلى ثلاث قنوات.
- **VLD4** تقوم بعمليات التحميل وإلغاء التشابك ضمن أربع مسجلات. وهي مفيدة من أجل معالجة معطيات الصورة من نوع ARGB.

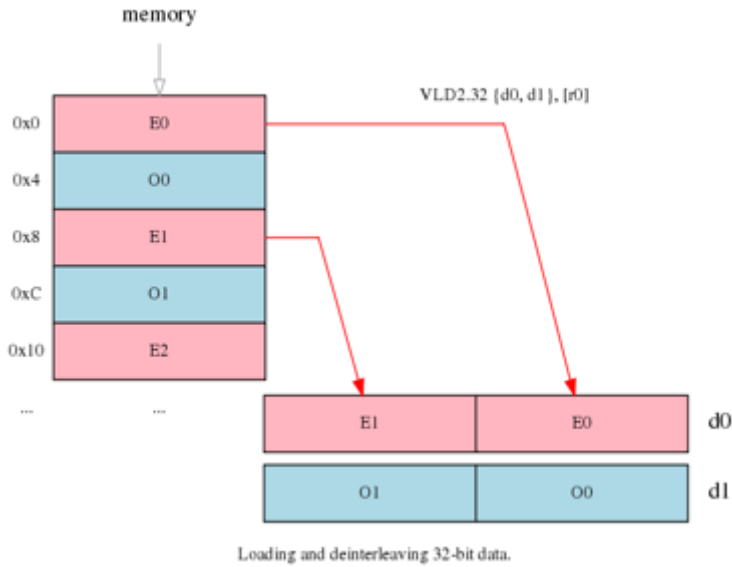
ج- أنواع العناصر Element types

تعتمد عناصر تشابك عمليات التحميل والتخزين على الحجم الموصوف ضمن التعليمية. على سبيل المثال تحميل مسجلي NEON 64-bit باستخدام VLD2.16 ينتج لدينا أربع عناصر بحجم 16-bit ضمن المسجل الأول، وأربع عناصر بحجم 16-bit ضمن المسجل الثاني كما هو موضح في الشكل 6-0. يقوم إلغاء التشابك بفصل الأزواج من العناصر المتجاورة ضمن الذاكرة إلى مسجلات منفصلة.



الشكل 6-0 تحميل وإلغاء تشابك معطيات بعرض 16 بت

ويؤدي تغيير حجم العنصر إلى 32-bit إلى تحميل نفس المقدار من المعطيات. لكن في هذه الحالة سيتكون الشعاع المشكل فقط من عنصرين بسبب تغيير حجم كل عنصر إلى 32-bit بدلاً من 16-bit. وهذا الأمر يقوم بفصل العناصر الزوجية والفردية أيضاً من الذاكرة إلى مسجلات منفصلة كما هو موضح في الشكل 7-0.



الشكل 7-0 تحميل وإلغاء تشابك معطيات بعرض 32 بت

يؤثر حجم العناصر على التعامل الطرفي endian handling. بشكل عام إذا تم توصيف الحجم الصحيح للعنصر ضمن تعليمات التحميل والتخزين عندها سيتم قراءة البايتات من الذاكرة وفق ترتيب مناسب، وبنفس الوقت سيعمل نفس الرماز على كل من الأنظمة ذو الطرفي الصغير Little-endian والطرفي الكبير Big-endian¹.

في النهاية فإن حجم العنصر له تأثير على محاذاة المؤشر pointer alignment. كما تعطي المحاذاة لحجم العنصر أداء أفضل عموماً، وربما تكون أحد متطلبات نظام التشغيل المطلوب. على سبيل المثال، أثناء تحميل عناصر بحجم 32-bit، يجب محاذاة عنوان أول عنصر إلى 32-bit [19].

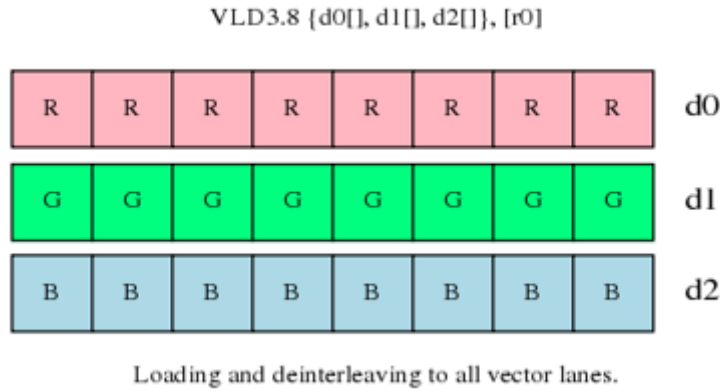
3.1.2. العناصر الوحيدة أو المتعددة Single or Multiple Elements

تقوم بنى التحميل بإلغاء التشابك من الذاكرة، كما يمكنها، ضمن كل مسجل NEON، أن تقوم بـ:

- تحميل مسارب متعددة Multiple lanes بعناصر مختلفة، كما هو مبين في الشكل 3-0

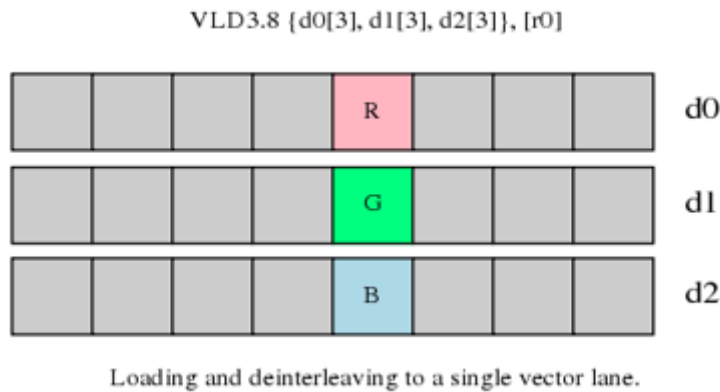
¹ المقصود بذلك هو كيفية ترتيب البايتات في الكلمة من اليمين إلى اليسار أو العكس

- تحميل مسارب متعددة بنفس العناصر، كما هو مبين في الشكل 8-0
- تحميل مسرب وحيد بعنصر وحيد مع ترك المسارب الأخرى دون أي تأثير. الشكل 9-0



الشكل 8-0 تحميل و إلغاء التشابك لجميع مسارب الأشعة

تعدّ الصيغة الأخيرة مفيدة عندما يراد تشكيل شعاع من معطيات مبعثرة في الذاكرة.



الشكل 9-0 تحميل وإلغاء تشابك لمسرب شعاع وحيد

تزود تعليمات التخزين دعم مماثل من ناحية كتابة عنصر وحيد أو عناصر متعددة مع التشابك.

3.1.3. العنونة Addressing

تدعم تعليمات بنى التحميل والتخزين عدد من الصيغ لتوصيف عناوين الذاكرة.

- مسجل Register: [Rn {,align}]
- وهذه أبسط صيغة. حيث تقوم التعليمات بتحميل وتخزين المعطيات ضمن عنوان مخصص.

- **مسجل مع زيادة تالية** Register with increment after $[Rn \{, :align\}]!$:
تقوم التعليمات بتحديث المؤشر بعد التحميل أو التخزين. وبالتالي يصبح المؤشر جاهزاً من أجل التعليمات التالية لتحميل أو تخزين الحزمة التالية من العناصر. وتساوي الزيادة عدد البايتات المقروءة أو المكتوبة من قبل التعليمات.
 - **مسجل مع دليل لاحق** Register with post-index $[Rn \{, :align\}]$, Rm :
بعد النفاذ إلى الذاكرة، تقوم التعليمات بزيادة المؤشر بمقدار القيمة الموجودة ضمن المسجل Rm . ويعدّ هذا الأمر مفيداً من خلال قراءة أو كتابة مجموعة من العناصر التي تكون مفصولة بأبعاد متساوية، وكمثال على هذا الأمر عندما يراد قراءة معطيات خط شاقولي من صورة.
- يمكن، ضمن كل الصيغ، توصيف المحاذاة للمؤشر الذي تم تمريره ضمن Rn ، باستخدام المتوسط الاختياري $align$. حيث أن توصيف المحاذاة غالباً ما يسرع عمليات النفاذ للذاكرة [19].

3.1. 4. أشكال أخرى من التحميل والتخزين Other Loads and Stores

يتعامل مثال تبادل قنوات الألوان السابق مع تعليمات NEON من أجل بنى التحميل والتخزين. حيث تدعم هذه التعليمات أنواع معطيات من حجم 8-bit و 16-bit و 32-bit فقط. تؤمن حزمة تعليمات NEON أيضاً:

- **VLDR و VSTR** لتحميل و تخزين مسجل وحيد كقيمة 64-bit.
- **VLDM و VSTM** لتحميل مسجلات متعددة كقيمة 64-bit. وهي مفيدة من أجل تخزين وطلب مسجلات من المكس.

3.2. التعامل مع أبعاد الصفيفة غير المضاعفة

Handling non-multiple array lengths

يصف هذا الجزء مشكلة شائعة عندما يكون عدد العناصر المدخلة للمعالجة ليس من مضاعفات طول الشعاع. على سبيل المثال لمعالجة عناصر 16-bit ضمن مسجل 64-bit، فإن طول الشعاع هو أربعة لأنه يمكن أن يتسع لأربع عناصر 16-bit. ومع ذلك يمكن أن تحوي صفيفة معطيات الدخل ستة وعشرين عنصراً بطول 16-bit. عندها يجب معالجة بقايا العناصر في بداية أو نهاية الصفيفة [19].

3.2.1. البقايا Leftovers

تقوم عادة تعليمات NEON بعملياتها على الأشعة التي يتراوح طولها من 4 إلى 16 عنصراً. وعادة لا يكون طول صفيقة الدخل من مضاعفات طول الشعاع. ويكون الحل بمعالجة أكبر قدر ممكن من العناصر باستخدام طول الشعاع الكامل. بعدها تتم معالجة بقايا العناصر بشكل منفصل.

3.2.2. مثال عن المشكلة Example problem

يعرض هذا الجزء مسألة وجود صفيقة دخل تحوي 21 عنصراً. وضمن كل عملية تكرار يتم تحميل ومعالجة ثم تخزين ثمانية عناصر. يتم معالجة ثمانية عناصر ضمن كل من عمليتي التكرار الأولى، في حين تتم فقط معالجة خمس عناصر ضمن عملية التكرار الثانية.

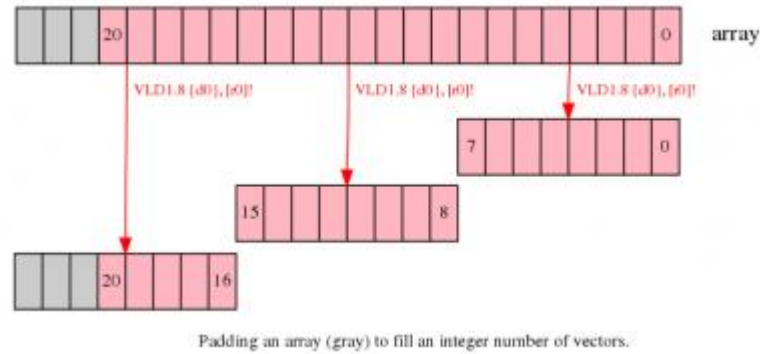
يوجد ثلاثة طرائق لمعالجة هذه البقايا، وهذه الطرائق مختلفة عن بعضها بعضاً من حيث المتطلبات، والأداء وحجم الرماز. من ناحية سرعة التعامل مع البقايا، مع ترتيب الطريقة الأسرع أولاً، تعطى هذه الطرائق كما يلي:

- استخدام صفيقات أكبر Larger arrays
- التداخل Overlapping
- معالجة العناصر الوحيدة Single element processing

3.2.3. استخدام صفيقات أكبر Using Larger Arrays

يمكن تغيير حجم صفيقات الدخل، وذلك بزيادة طول الصفيقة للمضاعفات التالية لحجم الشعاع باستخدام عناصر حشو padding elements. وهذا يسمح لتعليمات NEON بالقراءة والكتابة إلى ما بعد نهاية صفيقة الدخل بدون تخريب معطيات الذاكرة المتجاورة.

في المثال الموصوف بـ 21 عنصر دخل، تسمح زيادة حجم صفيفة الدخل إلى 24 عنصر للتكرارية الثالثة بالانتهاء من دون أي تخريب محتمل للمعطيات. الشكل 10-0



الشكل 10-0 حشو صفيفة (اللون الرمادي) لملء عدد صحيح من الأشعة

ملاحظة: قد تحتاج عناصر الحشو المنشأة في نهاية الصفيفة إلى تهيئتها بقيمة بحيث لا تؤثر على نتيجة الحساب. على سبيل المثال، إذا كانت العملية هي لجمع صفيفة، عندئذ يجب تهيئة العناصر الجديدة بقيمة صفر كي لا تتأثر النتيجة. وإذا كانت العملية لإيجاد القيمة الدنيا من الصفيفة، يمكن فقط إعطاء العناصر الجديدة القيم الأعظمية التي يمكن أن تؤخذ.

عيوب طريقة تطويل الصفيفة هي:

- يستهلك تخصيص صفيفات أكبر ذاكرة أكثر. ويمكن ظهور الزيادة بشكل واضح إذا كان هناك العديد من الصفيفات القصيرة.
- لا يمكن في بعض الحالات أن تتم تهيئة عناصر الحشو بقيمة لا تؤثر على نتيجة الحساب. على سبيل المثال عند إيجاد مجال range لمجموعة من الأرقام.

جزء الرماز Code fragment

```

@ r0 = input array pointer
@ r1 = output array pointer
@ r2 = length of data in array

@ We can assume that the array length is greater than zero, is an
integer
@ number of vectors, and is greater than or equal to the length of data
@ in the array.

    add    r2, r2, #7           @ add (vector length-1) to the data length
    lsr    r2, r2, #3           @ divide the length of the array by the length
                                @ of a vector, 8, to find the number of
                                @ vectors of data to be processed

loop:
    subs    r2, r2, #1          @ decrement the loop counter, and set flags
    vld1.8  {d0}, [r0]!         @ load eight elements from the array pointed to
                                @ by r0 into d0, and update r0 to point to the
                                @ next vector
    ...
    ...                         @ process the input in d0
    ...

    vst1.8  {d0}, [r1]!         @ write eight elements to the output array,
and
    bne     loop                @ update r1 to point to next vector
                                @ if r2 is not equal to 0, loop

```

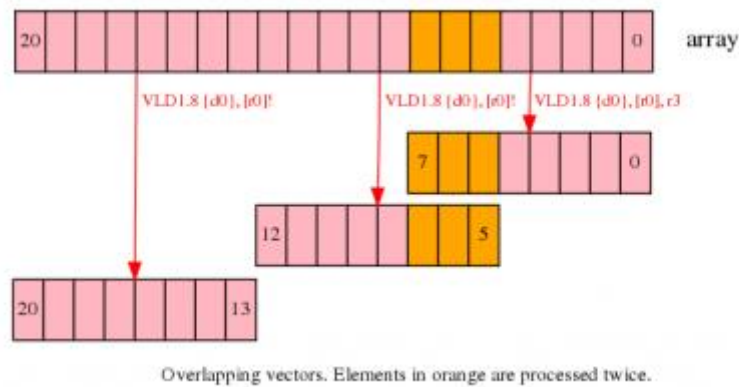
3. 2. 4. التداخل Overlapping

تضمن طريقة التداخل معالجة بعض العناصر ضمن الصفيفة مرتين. ويمكن أن تناسب هذه الطريقة بعض الحالات.

لمعالجة 21 عنصر باستخدام شعاع بطول 8:

- يعالج التكرار الأول العناصر من 0 إلى 7.
- يعالج التكرار الثاني العناصر من 5 إلى 12.
- يعالج التكرار الثالث العناصر من 13 إلى 20.

ملاحظة: تعالج هذه الطريقة العناصر من 5 إلى 7 مرتين لتواجد هذه العناصر مرتين ضمن كل من تكراري الأشعة الأول والثاني كما هو موضح في الشكل 0-11.



الشكل 0-11 الأشعة المتداخلة، حيث يتم معالجة العناصر البرتقالية مرتين

يمكن استخدام التداخل فقط عندما تكون العملية هي idempotent. وهذا يعني أنه يجب ألا تتغير القيمة اعتماداً على عدد مرات تطبيق العملية على معطيات الدخل. على سبيل المثال، يمكن استخدام هذه الطريقة لإيجاد العنصر الأكبر أو الأصغر ضمن صفيفة. بينما لا يمكن استخدامها لجمع صفيفة لأن العناصر المتداخلة ستجمع مرتين.

ملاحظة: يمكن استخدام طريقة التداخل فقط إذا كان عدد عناصر صفيفة الدخل أكثر من طول الشعاع. ولا يمكن تطبيق هذه الطريقة ضمن أي تكرار للأشعة vector iteration.

جزء الرماز Code fragment

```

@ r0 = input array pointer
@ r1 = output array pointer
@ r2 = length of data in array

@ We can assume that the operation is idempotent, and the array is
greater
@ than or equal to one vector long.

    ands    r3, r2, #7      @ calculate number of elements left over
after                          @
                                @ processing complete vectors using
                                @ data length & (vector length - 1)
    beq     loopsetup      @ if the result of the ands is zero, the
length                          @
                                @ of the data is an integer number of
vectors,                          @
                                @ so there is no overlap, and processing
can begin                          @
                                @ at the loop

                                @ handle the first vector separately
    vld1.8  {d0}, [r0], r3  @ load the first eight elements from the
array,                          @
                                @ and update the pointer by the number of
elements                          @
                                @ left over
    ...
    ...                      @ process the input in d0

    vst1.8  {d0}, [r1], r3  @ write eight elements to the output array,
and                              @
                                @ update the pointer
                                @ now, set up the vector processing loop
loopsetup:
    lsr     r2, r2, #3      @ divide the length of the array by the
length                          @
                                @ of a vector, 8, to find the number of
                                @ vectors of data to be processed

                                @the loop can now be executed as normal.the
                                @first few elements of the first vector will
                                @overlap with some of those processed above
loop:
    subs     r2, r2, #1      @ decrement the loop counter, and set flags
    vld1.8  {d0}, [r0]!     @ load eight elements from the array, and
update                          @
                                @ the pointer
    ...
    ...                      @ process the input in d0

    vst1.8  {d0}, [r1]!     @ write eight elements to the output array,
and                              @
                                @ update the pointer
    bne     loop            @ if r2 is not equal to 0, loop

```

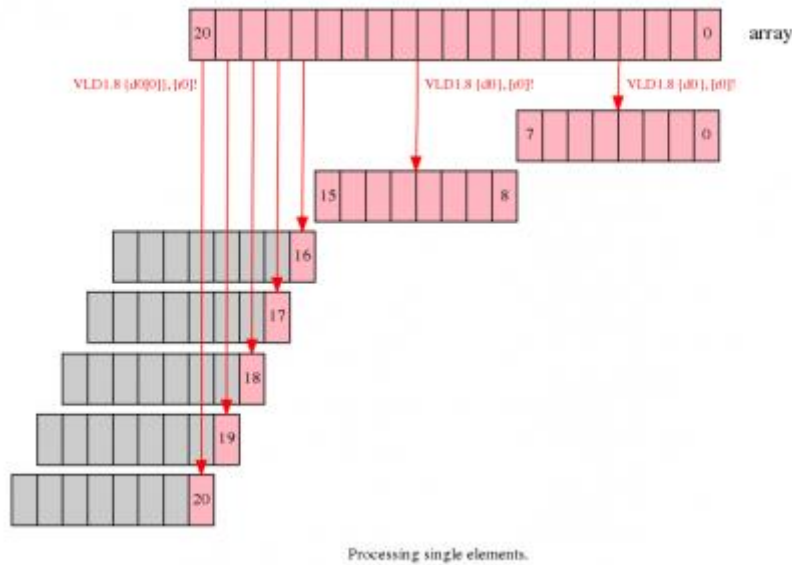
3.2.5. معالجة العنصر الوحيد Single Element processing

تؤمن حزمة تعليمات NEON تعليمات تحميل وتخزين قادرة على العمل على عناصر وحيدة ضمن الشعاع. وباستخدام هذه التعليمات يمكن عندها تحميل شعاع جزئي يحتوي على عنصر وحيد، وإجراء العمليات عليه، ثم كتابة العنصر لاحقاً في الذاكرة.

لمعالجة 21 عنصر باستخدام شعاع بطول 8:

- يعالج التكرار الأول العناصر من 0 إلى 7.
- يعالج التكرار الثاني العناصر من 8 إلى 12.
- حلقة منفصلة separate loop تقوم بمعالجة العناصر من 16 إلى 20 بشكل منفصل.

بعد الانتهاء من أول تكرارين، يتبقى فقط 5 عناصر للمعالجة، عندها تقوم حلقة منفصلة بعمليات التحميل والمعالجة وتخزين هذه العناصر باستخدام تعليمات NEON التي لها القدرة على التعامل مع هذه العناصر بشكل منفصل كما هو موضح في الشكل 0-12.



الشكل 0-12 معالجة العناصر الوحيدة

ملاحظات:

- معالجة عنصر وحيد هي أبطأ من الطرق السابقة، حيث يجب تحميل ومعالجة ثم تخزين كل عنصر بشكل مستقل.

- التعامل مع البقايا مثل هذه الحالة يتطلب حلقتي تكرار:
 - من أجل تكرار الأشعة vector iterations
 - من أجل عمليات العنصر الوحيد
 وهذا يؤدي إلى مضاعفة مقدار الرماز ضمن التابع.
- تقوم تعليمة تحميل العنصر الوحيد بتغيير قيمة عنصر الوجهة destination element فقط وتترك بقية الشعاع بدون أي تغيير. إذا احتاجت العملية تعليمات التي تعمل على كل الشعاع، مثل VPADD، عندها يجب تهيئة الشعاع بقيمة مناسبة قبل تحميل أول عنصر وحيد ضمنه.

يمكن تطبيق طريقة العناصر الوحيدة إما عند بداية أو نهاية معالجة صفيفة الدخل.

جزء الرماز Code fragment

```
@ r0 = input array pointer
@ r1 = output array pointer
@ r2 = length of data in array

    lsrs    r3, r2, #3    @ calculate the number of complete vectors
to be
    beq     singlesetup  @ processed and set flags
to
    @ if there are zero complete vectors, branch
    @ the single element handling code
    @ process vector loop
vectors:
    subs    r3, r3, #1    @ decrement the loop counter, and set flags
    vld1.8  {d0}, [r0]!   @ load eight elements from the array and
update
    @ the pointer
    ...
    @ process the input in d0
    ...
    vst1.8  {d0}, [r1]!   @ write eight elements to the output array,
and
    @ update the pointer
    bne     vectors      @ if r3 is not equal to zero, loop

singlesetup:
    ands    r3, r2, #7    @ calculate the number of single elements to
process
    beq     exit          @ if the number of single elements is zero,
branch
    @ to exit
    @ process single element loop
singles:
    subs    r3, r3, #1    @ decrement the loop counter, and set flags
    vld1.8  {d0[0]}, [r0]! @ load single element into d0, and update
the
```

```

@ pointer
...
@ process the input in d0[0]
...

vst1.8 {d0[0]}, [r1]! @ write the single element to the output
array,
@ and update the pointer
bne singles @ if r3 is not equal to zero, loop
exit:

```

3.2.6. المحاذاة Alignment

ينبغي أن يتم محاذاة عناوين التحميل والتخزين لخطوط الخابية cache lines للسماح بالنفاذ إلى الذاكرة بشكل أكثر كفاءة. وهذا يتطلب محاذاة 16-word على الأقل على معالجات Cortex-A8. وإذا كان من غير الممكن محاذاة البداية لصفيفات الدخل والخرج، عندها يفضل معالجة العناصر غير المحاذاة كعناصر وحيدة. وهذا يعني أنه يمكن معالجة بعض العناصر عند بداية ونهاية الصفيفة كعناصر وحيدة، ويسمح بدوره للعناصر في منتصف الصفيفة، حيث يوجد محاذاة، أن تعالج كأشعة.

عند محاذاة النفاذ إلى الذاكرة يجب عدم نسيان استخدام مؤهلات العنوان address qualifiers: 64 أو 128: أو 256: مع تعليمات التحميل والتخزين من أجل أداء أمثل. يستعرض كتيب المرجع التقني لـ Cortex-A8 عدد الدورات cycles اللازمة لتعليمات التحميل والتخزين للمحاذاة المختلفة [19].

3.2.7. استخدام تعليمات ARM

يمكن استخدام تعليمات ARM ضمن طريقة معالجة العنصر الوحيد للعمل على كل عنصر. إلا أن التخزين على نفس مساحة الذاكرة من قبل ARM و NEON معاً يؤدي إلى تقليل الأداء. بسبب تأخير عمليات الكتابة من قنوات ARM pipelines إلى أن يتم الانتهاء من كتابات قنوات NEON pipelines. وكنتيجة يجب تجنب عملية الكتابة على نفس مساحة الذاكرة، وخصوصاً نفس خط الخابية cache line، من قبل كل من رمزي ARM و NEON [19].

3.3. ضرب المصفوفات Matrix Multiplication

تعرض هذه الفقرة مثلاً عن كيفية ضرب مصفوفات بقياس 4x4 بشكل فعال. حيث تستخدم هذه العملية بشكل متكرر ضمن معالجة الرسوميات ثلاثية الأبعاد. حيث يفترض ضمن هذا المثال أن المصفوفات مخزنة في لذاكرة حسب الترتيب الأعلى للأعمدة. وهي الصيغة المستخدمة من قبل OpenGL-ES.

3.3.1 الخوارزمية المستخدمة Used Algorithm

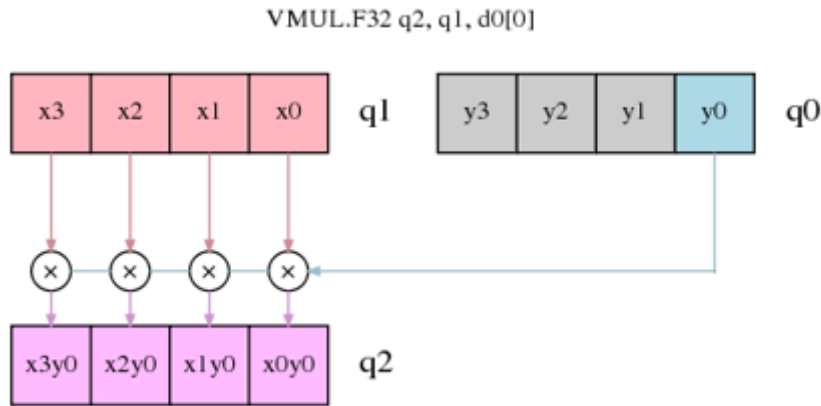
من خلال عرض تفاصيل عملية ضرب المصفوفات، يمكن تحديد عمليات فرعية والتي يمكن إنجازها باستخدام تعليمات NEON.

$$\begin{array}{cccc}
 x0 & x4 & x8 & xC \\
 x1 & x5 & x9 & xD \\
 x2 & x6 & xA & xE \\
 x3 & x7 & xB & xF
 \end{array}
 \cdot
 \begin{array}{cccc}
 y0 & y4 & y8 & yC \\
 y1 & y5 & y9 & yD \\
 y2 & y6 & yA & yE \\
 y3 & y7 & yB & yF
 \end{array}
 =
 \begin{array}{cccc}
 x0y0 + x4y1 + x8y2 + xCy3 & \dots & \dots & \dots \\
 x1y0 + x5y1 + x9y2 + xDy3 & \dots & \dots & \dots \\
 x2y0 + x6y1 + xAy2 + xEy3 & \dots & \dots & \dots \\
 x3y0 + x7y1 + xBy2 + xFy3 & \dots & \dots & \dots
 \end{array}$$

Matrix multiplication showing one column of results.

الشكل 13-0 ضرب مصفوفتين يظهر عمود وحيد من النتائج

يلاحظ من الشكل 13-0 أن كل عمود من أعمدة المصفوفة الأولى (الموضح باللون الأحمر) يُضرب بقيمة وحيدة مقابلة في المصفوفة الثانية (الظاهرة باللون الأزرق). بعدها يتم إضافة نتائج الضرب المنفصلة كما هو موضح في الشكل 13-0، وهذا يعطي عموداً من القيم لمصفوفة النتيجة. تكرر هذه العملية من أجل باقي الأعمدة في مصفوفة النتيجة.



NEON vector-by-scalar multiplication.

الشكل 14-0 ضرب شعاع مع موسع ضمن بنية NEON

إذا كان كل عمود هو شعاع ضمن مسجل NEON، عندها فإن تعليمة ضرب الشعاع باستخدام الموسع vector-by-scalar multiply ستقوم بحساب كل عمود ضمن مصفوفة النتيجة بشكل أكثر كفاءة. يمكن إنجاز العملية الفرعية المسلط عليها الضوء في الشكل 14-0 باستخدام هذه العملية. وتستخدم النسخة التراكمية لتعليمة الضرب لمراكمة نتائج الضرب المستقلة من كل عمود للمصفوفة الأولى.

يقوم هذا المثال بالعمل على أعمدة المصفوفة الأولى مولداً عمود من النتائج. وبالتالي فإن قراءة وكتابة العناصر من وإلى الذاكرة هو عملية خطية، ولا تحتاج إلى تعليمات التحميل والتخزين المتشابكة.

3.3. الرماز المنجز Implemented Code

تملك وحدة NEON إثنين وثلاثين مسجل 64-bit. يمكن لكل مسجل نوع D أن يمسك عنصرين من نوع الفاصلة العائمة بعرض 32-bit. وبالتالي يمكن تحميل كل العناصر من كلا مصفوفتي الدخل إلى مسجلات NEON، مع بقاء مسجلات أخرى لاستخدامها كمراكمات. تقوم مسجلات D الثمانية من d16 إلى d23 بمسك 16 عنصراً من المصفوفة الأولى. بينما مصفوفات D الثمانية من d0 إلى d7 بمسك 16 عنصراً من المصفوفة الثانية.

تستخدم عمليات الضرب بالفاصلة العائمة مسجلات Q لاحتواء كل عمود مصفوفة على أربعة أرقام نقطة عائمة نوع 32-bit والتي تتسع ضمن مسجل Q وحيد ذو طول 128-bit.

أ- الفاصلة العائمة Floating-point

يعرض هنا طريقة انجاز عمليات ضرب مصفوفات بالفاصلة العائمة بدقة وحيدة single precision floating-point matrices، حيث تحوي كل مصفوفة على 16 قيمة نوع 32-bit floating-point.

في البداية تُحمّل المصفوفات من الذاكرة إلى مسجلات NEON. حيث تكون مخزنة بترتيب العمود الأعلى وزنا column-major order، وبالتالي فإن أعمدة المصفوفة مخزنة بشكل خطي ضمن الذاكرة. ويمكن تحميل عمود ضمن مسجلات NEON باستخدام التعليمات VLD1، وإعادة كتابته إلى الذاكرة باستخدام VST1.

```
vld1.32 {d16-d19}, [r1]! @ load first eight elements of matrix 0
vld1.32 {d20-d23}, [r1]! @ load second eight elements of matrix 0
vld1.32 {d0-d3}, [r2]! @ load first eight elements of matrix 1
vld1.32 {d4-d7}, [r2]! @ load second eight elements of matrix 1
```

يمكن حساب عمود النتائج باستخدام أربعة تعليمات ضرب NEON فقط

```
vmul.f32    q12, q8, d0[0]    @ multiply col element 0 by matrix col 0
vmla.f32    q12, q9, d0[1]    @ multiply-acc col element 1 by matrix col 1
vmla.f32    q12, q10, d1[0]   @ multiply-acc col element 2 by matrix col 2
vmla.f32    q12, q11, d1[1]   @ multiply-acc col element 3 by matrix col 3
```

تقوم التعليمة الأولى vmul.f32 بإنجاز العملية الموصوفة ضمن الشكل (3-14). ويتم ضرب كل من x0، x1، x2 و x3 (ضمن المسجل q8) بـ y0 (العنصر 0 ضمن المسجل d0) ثم تخزينها ضمن المسجل q12.

تعمل التعليمات اللاحقة على الأعمدة الأخرى من المصفوفة الأولى. حيث تراكم النتائج ضمن المسجل q12 لإعطاء العمود الأول من القيم لمصفوفة النتيجة.

ملاحظة: يشير الموسّع scalar المستخدم في تعليمات الضرب إلى مسجلات D، وعلى الرغم من أن q0[3] هو نفس القيمة كما لو كان d1[1]، قد لا يقبل مجمع GNU assembler استخدام مسجل Q للإشارة إلى الموسّع. لذا يجب توصيف الموسّع باستخدام المسجل D.

يؤدي الرماز السابق عملية ضرب شعاع بمصفوفة بشكل فعال، والتي تعد عملية شائعة ضمن الرسومات ثلاثية الأبعاد. ومع ذلك يقوم الرماز السابق بحساب العمود الأول من المصفوفة الناتجة فقط من أجل ضرب مصفوفتين. ويحتاج حساب كامل نتيجة ضرب مصفوفة بمصفوفة ثلاثة تكرارات إضافية:

- يستخدم التكرار الثاني القيم y4-y7 في المسجل q1.
- يستخدم التكرار الثالث القيم y8-yb في المسجل q2.
- يستخدم التكرار الرابع القيم yc-yf في المسجل q3.

ويمكن تبسيط الرماز المطلوب بشكل ملحوظ إذا أمكن إنشاء ماكرو للتعليمات السابقة.

```
.macro mul_col_f32 res_q, col0_d, col1_d
vmul.f32    \res_q, q8, \col0_d[0]    @ multiply col element 0 by matrix
col 0
vmla.f32    \res_q, q9, \col0_d[1]    @ multiply-acc col element 1 by
matrix col 1
vmla.f32    \res_q, q10, \col1_d[0]   @ multiply-acc col element 2 by
matrix col 2
vmla.f32    \res_q, q11, \col1_d[1]   @ multiply-acc col element 3 by
matrix col 3
.endm
```

وعندها يصبح إنجاز ضرب مصفوفة 4 بـ 4 ذو الفاصلة العائمة مشابهاً للرمز الآتي:

```
vld1.32 {d16-d19}, [r1]!    @ load first eight elements of matrix 0
vld1.32 {d20-d23}, [r1]!    @ load second eight elements of matrix 0
vld1.32 {d0-d3}, [r2]!      @ load first eight elements of matrix 1
vld1.32 {d4-d7}, [r2]!      @ load second eight elements of matrix 1

mul_col_f32 q12, d0, d1      @ matrix 0 * matrix 1 col 0
mul_col_f32 q13, d2, d3      @ matrix 0 * matrix 1 col 1
mul_col_f32 q14, d4, d5      @ matrix 0 * matrix 1 col 2
mul_col_f32 q15, d6, d7      @ matrix 0 * matrix 1 col 3

vst1.32 {d24-d27}, [r0]!    @ store first eight elements of result
vst1.32 {d28-d31}, [r0]!    @ store second eight elements of result
```

ب- الفاصلة الثابتة Fixed Point

يعدّ إجراء العمليات الحسابية بالفاصلة الثابتة fixed point arithmetic غالباً أسرع من العمليات الحسابية بالفاصلة العائمة floating-point. كونها تتطلب عرض حزمة ذاكرة أقل لقراءة وكتابة القيم نظراً لاستخدامها بتات أقل. عموماً إن ضرب القيم الصحيحة أسرع من نفس العمليات المطبقة على أعداد الفاصلة العائمة.

ومع ذلك من الضروري، عند إجراء الحسابات بالفاصلة الثابتة، الانتباه إلى:

- اختيار التمثيل representation بشكل حذر لتجنب الطفحان (الفيض) overflow أو الاشباع saturation.
- الحفاظ على درجة الدقة في النتائج التي تلزم التطبيق.

يشبه إنجاز ضرب المصفوفات باستخدام الفاصلة الثابتة إلى حد كبير إنجازها بالفاصلة العائمة. يستخدم هذا المثال صيغة الفاصلة الثابتة Q1.14، وتتشابه التعليمات المطلوبة للصيغ الأخرى ومن الممكن أن تحتاج فقط إلى تغيير الإزاحة النهائية المطبقة إلى المراكز. ويعطى الماكرو كما يلي:

```
.macro mul_col_s16 res_d, col_d
vmull.s16 q12, d16, \col_d[0] @multiply col element0 by matrix col0
vmlal.s16 q12, d17, \col_d[1] @multiply-acc col element1 by matrix col1
vmlal.s16 q12, d18, \col_d[2] @multiply-acc col element2 by matrix col2
vmlal.s16 q12, d19, \col_d[3] @multiply-acc col element3 by matrix col3
vqshrns.s32 \res_d, q12, #14 @shift right and narrow accumulator into
                                @Q1.14 fixed point format,with saturation
.endm
```

بالمقارنة مع الماكرو المستخدم في نسخة الفاصلة العائمة، فإن الاختلافات الجوهرية هي:

- عرض القيم هنا 16-bit بدلا من 32-bit، لذا يمكن لمسجلات D مسك أربعة قيم دخل.
- إن نتيجة ضرب عددين كل منهما بعرض 16-bit هي عدد بعرض 32-bit. ويمكن استخدام تعليمتي VMULL و VMLAL لأنها تقوم بتخزين نتيجتها ضمن مسجلات Q. وبهذا تحافظ على جميع بتات النتيجة باستخدام عناصر مضاعفة الحجم.
- يجب أن تكون النتيجة النهائية بعرض 16-bit، لكن طول المراكم هو 32-bit. وللحصول على نتيجة بطول 16-bit تستخدم التعليمة VQRSHRN [19]. حيث تقوم هذه التعليمة بـ:
 - إضافة القيمة المدورة بشكل صحيح لكل عنصر
 - إزاحتها إلى اليمين
 - إشباع النتيجة إلى أضيق حجم عنصر جديد.

كما أن إنقاص العنصر من 32-bit إلى 16-bit له تأثير على عدد ولوجات الذاكرة. حيث يمكن تحميل وتخزين المعطيات باستخدام تعليمات أقل. يعطى الرمز اللازم من أجل مصفوفة فاصلة ثابتة كما يلي:

```
vld1.16 {d16-d19}, [r1]    @ load sixteen elements of matrix 0
vld1.16 {d0-d3}, [r2]      @ load sixteen elements of matrix 1

mul_col_s16 d4, d0          @ matrix 0 * matrix 1 col 0
mul_col_s16 d5, d1          @ matrix 0 * matrix 1 col 1
mul_col_s16 d6, d2          @ matrix 0 * matrix 1 col 2
mul_col_s16 d7, d3          @ matrix 0 * matrix 1 col 3

vst1.16 {d4-d7}, [r0]      @ store sixteen elements of result
```

ج- الجدولة Scheduling

يمكن تحسين أداء ضرب مصفوفة بأخرى من خلال جدولة التعليمات. وفي الماكرو التالي، تقوم تعليمات الضرب المتجاورة بالكتابة على نفس المسجل، لذا يجب على أنابيب NEON أن تنتظر انتهاء كل عملية ضرب قبل أن تبدأ التعليمة التالية.

ويمكن فصل التعليمات المعتمدة على نفس المسجلات من خلال جدولة التعليمات الأخرى بينهما أو من خلال إعادة ترتيب التعليمات. حيث يمكن إصدار هذه التعليمات أثناء تنفيذ التعليمات الأخرى بالخلف. يقوم المثال التالي بإعادة ترتيب الرموز لإتاحة مساحة لنفاذات accesses مسجلات المراكم.

```
vmul.f32 q12, q8, d0[0] @ rslt col0 = (mat0 col0) * (mat1 col0 elt0)
vmul.f32 q13, q8, d2[0] @ rslt col1 = (mat0 col0) * (mat1 col1 elt0)
vmul.f32 q14, q8, d4[0] @ rslt col2 = (mat0 col0) * (mat1 col2 elt0)
vmul.f32 q15, q8, d6[0] @ rslt col3 = (mat0 col0) * (mat1 col3 elt0)
```

```
vmla.f32 q12, q9, d0[1] @ rslt col0 += (mat0 col1) * (mat1 col0 elt1)
vmla.f32 q13, q9, d2[1] @ rslt col1 += (mat0 col1) * (mat1 col1 elt1)
...
...
```

وباستخدام هذا الإصدار من الرماز، يتضاعف تقريباً أداء ضرب مصفوفة بمصفوفة على نظم Cortex-A8.

3.4. عمليات الإزاحة Shift operations

تقدم هذه الفقرة عمليات إزاحة NEON، وكيفية استخدامها لتحويل معطيات الصورة بين أعماق الألوان المستخدمة بشكل شائع.

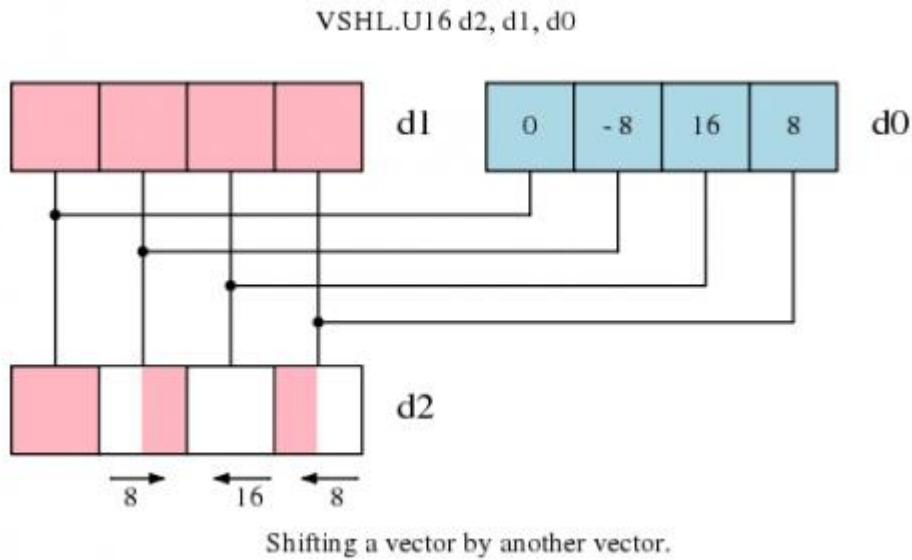
يقدم المجال القوي من تعليمات الإزاحة المزود من قبل حزمة تعليمات NEON:

- عمليات ضرب وقسمة سريعة للأشعة من قوة 2، مع التقريب rounding والإشباع saturation.
- عمليات إزاحة ونسخ للبتات من شعاع إلى آخر.
- حسابات مؤقتة عند الدقة العالية high precision ومراكمة النتائج عند الدقة الأدنى lower precision.

3.4.1. إزاحة الأشعة Shifting Vectors

تشابه عملية إزاحة NEON إلى شكل كبير عمليات الإزاحة الموجودة ضمن رماز ARM القابل للتوسعة. حيث تقوم الإزاحة بتحريك البتات ضمن كل عنصر من عناصر الشعاع يميناً أو يساراً. يتم تجاهل أي بت تسقط من يمين أو يسار كل عنصر، حيث لا يتم إزاحتهم إلى العناصر المجاورة.

يمكن توصيف مقدار الإزاحة من خلال حرف مرمز ضمن التعليمات، أو من خلال شعاع إزاحة إضافي. عند استخدام شعاع إزاحة يتم تطبيق الإزاحة إلى كل عنصر من شعاع الدخل اعتماداً على قيمة العنصر الموافق في شعاع الإزاحة. يتم التعامل مع عناصر أشعة الإزاحة كقيمة مؤشرة signed value، وبالتالي فإن الإزاحة إلى اليمين وإلى اليسار وإلى الصفر تكون ممكنة، على أساس كل عنصر per-element basis. الشكل 15-0

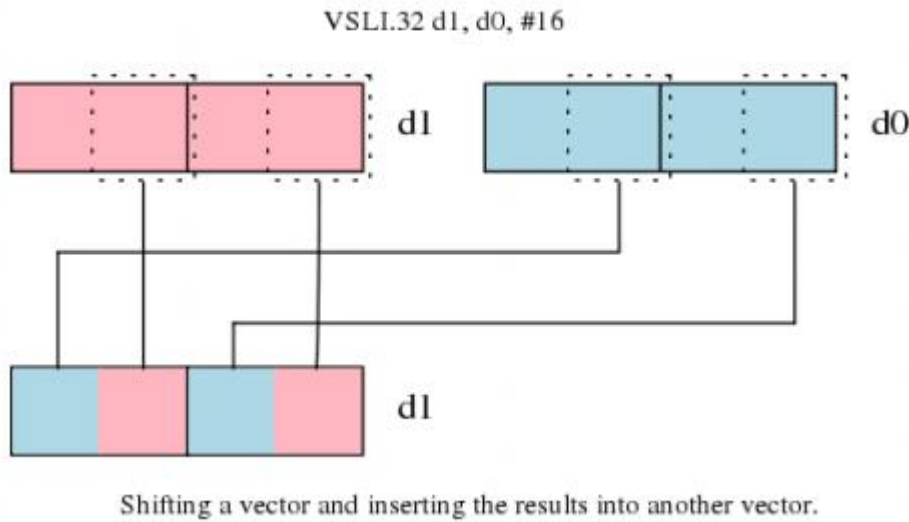


الشكل 15-0 إزاحة شعاع باستخدام شعاع الإزاحة

إن عمل إزاحة إلى اليمين على شعاع من عناصر مؤشرة، محدد من النوع المرفق للتعليمية، ستمدد إشارة كل عنصر. وهذا يكافئ إلى الإزاحة الحسابية arithmetic shift ضمن رماز ARM. بينما الإزاحة المطبقة على الأشعة غير المؤشرة لا تمدد الإشارة sign extend.

3. 4. 2. الإزاحة والإدخال Shifting and inserting

تزود تعليمات إزاحة NEON مع الإدخال طريقة لدمج البتات بين شعاعين. على سبيل المثال تقوم تعليمية الإزاحة إلى اليسار ثم الإدخال (VLSI) بإزاحة كل عنصر من عناصر شعاع المصدر source vector إلى اليسار. حيث أن البتات الجديدة المدخلة إلى يمين كل عنصر هي البتات الموافقة من شعاع الوجهة destination vector. الشكل 16-0



الشكل 16-0 إزاحة شعاع وإدخال النتائج إلى شعاع آخر

3.4.3. الإزاحة والمراكمة Shifting and accumulating

يوجد تعليمات NEON لإزاحة عناصر الشعاع يميناً ثم مراكمة النتائج ضمن شعاع آخر. وهي مفيدة في الحالات التي يتم عمل حسابات مؤقتة بدقة عالية، قبل دمج النتيجة مع مراكم بدقة منخفضة.

3.4.4. معدّلات التعليمات Instruction Modifiers

يمكن لكل تعليمة إزاحة أخذ معدّل واحد أو أكثر. وهذه المعدّلات لا تغير من عملية الإزاحة بذاتها، لكن يتم ضبط المدخلات والمخرجات لإزالة الانحياز bias أو لإشباعها ضمن مجال ما. يوجد 5 معدّلات لتعليمات الإزاحة:

- التقريب Rounding، تدل بـ R
- ضيق Narrow، تدل من خلال N
- عريض Long، تدل من خلال L
- الإشباع Saturating، تدل من خلال Q
- الإشباع غير المؤشر، تدل من خلال السابقة Q واللاحقة U. وهو مشابه لمعدّل الإشباع، لكن النتيجة هي مشبعة لمجال غير مؤشر عندما يتم إعطاؤه قيم دخل مؤشرة أو غير مؤشرة.

لا تصف بعض تركيبات هذه المعدّلات العمليات بشكل مفيد، وبالتالي لا يوجد أي تعليمة NEON لها. على سبيل المثال، إن الإزاحة المشبعة لليمين (والتي يمكن تسميتها VQSHR) ليست ضرورية، لأن الإزاحة لليمين تعطي نتائج أصغر، بالإضافة لعدم إمكانية تجاوز قيمتها للمجال متاح.

3.4.5. جدول بتعليمات الإزاحة المتاحة Table of shifts available

يعرض الجدول 1-0 جميع تعليمات إزاحة NEON مرتبة حسب المعدّلات التي تم ذكرها سابقاً. بحيث إذا حدث التباس بماهية عمل كل معدّل، يمكن عندها العودة إلى الجدول لاختيار التعليمة اللازمة.

جدول 1-0 عمليات الإزاحة ضمن NEON

Rounding						
		Long	Narrow		Long	Narrow
Left (Imm)	VSHL	VSHLL				
Left (Reg)	VSHL			VRSHL		
Right (Imm)	VSHR		VSHRN	VRSHR		VRSHRN
Left insert (Imm)	VSLI					
Right insert (Imm)	VSRI					
Right accumulate (Imm)	VSRA			VRRA		

Saturating						
		Long	Narrow		Long	Narrow
Left (Imm)	VQSHL					
Left (Reg)	VQSHL			VQRSHL		
Right (Imm)			VQSHRN			VQRSHRN

Unsigned Saturating						
		Long	Narrow		Long	Narrow
Left (Imm)	VQSHLU					
Left (Reg)						
Right (Imm)			VQSHRUN			VQRSHRUN

Shifts available on NEON. (Imm) is a literal shift, (Reg) is Register specified shift.

3.4.6. مثال تطبيقي: تحويل عمق اللون Color Depth Conversion

يعدّ التحويل بين أعماق اللون عملية متكررة في معالجة الرسوميات. حيث عادة ما تكون معطيات الدخل أو الخرج في صيغة لون 16-bit RGB565. لكن المعطيات بصيغة RGB888 أسهل للعمل لا سيما في حال استخدام وحدة NEON، حيث لا يوجد أي دعم طبيعي لنوع المعطيات من الشكل RGB565.



RGB888 and RGB565 color formats.

الشكل 0-17 صيغ اللون RGB888 و RGB565

ومع ذلك تستطيع وحدة NEON التعامل مع معطيات RGB565 بشكل أكثر كفاءة باستخدام عمليات إزاحة الأشعة.

أ- التحويل من صيغة 565 إلى صيغة 888

يتم أولاً التحويل من صيغة RGB565 إلى صيغة RGB888. يفترض المثال التالي أن المسجل q0 يحتوي على ثمانية بيكسلات بعرض 16-bit وبصيغة RGB565. يفصل الرمز البيكسلات الحمراء، والخضراء والزرقاء إلى عناصر بثمان بتات على طول المسجلات d2 إلى d4.

vshr.u8	q1, q0, #3	@ shift red elements right by three bits, @ discarding the green bits at the bottom of @ the red 8-bit elements.
vshrn.i16	d2, q1, #5	@ shift red elements right and narrow, @ discarding the blue and green bits.
vshrn.i16	d3, q0, #5	@ shift green elements right and narrow, @ discarding the blue bits and some red bits @ due to narrowing.
vshl.i18	d3, d3, #2	@ shift green elements left, discarding the @ remaining red bits, and placing green bits @ in the correct place.
vshl.i16	q0, q0, #3	@ shift blue elements left to most-significant @ bits of 8-bit color channel.
vmovn.i16	d4, q0	@ remove remaining red and green bits by @ narrowing to 8 bits.

يتم وصف تأثير كل تعليمة ضمن التعليقات أعلاه، لكن يمكن شرح العملية المنفذة على كل قناة بشكل مختصر كما يلي:

- إزالة معطيات اللون للقنوات المتجاورة باستخدام عمليات الإزاحة لدفع البتات إلى كل من طرفي العنصر.
- إجراء إزاحة ثنائية لتوزيع معطيات اللون في البتات الأعلى وزناً لكل عنصر.
- استخدام عملية التضييق Narrowing لإنقاص حجم العنصر من 16-bit إلى 8-bit.

ملاحظة: تستخدم أحجام العنصر ضمن هذا التتابع لعنونة العناصر 8-bit و 16-bit، لإنجاز بعض عمليات التقنيع masking operations.

جودة اللون الأبيض: بعد التحويل من صيغة RGB565 إلى صيغة RGB888 باستخدام الرمز السابق، فإن اللون الأبيض لن يكون أبيضاً بشكل كافٍ، وهذا بسبب أن البتّين أو الثلاثة الدنيا ضمن كل قناة تكون أصفراً بدلاً من الواحدات. إن الأبيض الممثل بصيغة RGB565 هو [0x1F, 0x3F, 0x1F]. يقوم الرمز السابق بتحويله إلى [0xF8, 0xFC, 0xF8] في صيغة RGB888، ويمكن حل هذه المشكلة باستخدام الإزاحة أثناء العمل لوضع بعض من البتات الأكثر وزناً ضمن البتات الأقل وزناً.

ب- التحويل من صيغة 888 إلى صيغة 565

يظهر الرمز في هذا المثال كيفية التحويل من صيغة RGB888 إلى صيغة RGB565. حيث يعدّ أن معطيات RGB888 هي من الصيغة المولدة من الرمز السابق. وبالتالي تكون بيكسلات اللون الأحمر والأخضر والأزرق مفصولة على ثلاثة مسجلات من d0 إلى d2، مع احتواء كل مسجل على ثمان عناصر. ويتم تخزين النتيجة كثمانية عناصر بـ 16-bit بصيغة RGB565 ضمن المسجل q2.

```
vshll.u8 q2, d0, #8 @ shift red elements left to most-significant
@ bits of wider 16-bit elements.
vshll.u8 q3, d1, #8 @ shift green elements left to most-significant
@ bits of wider 16-bit elements.
vsri.16 q2, q3, #5 @ shift green elements right and insert into
@ red elements.
vshll.u8 q3, d2, #8 @ shift blue elements left to most-significant
@ bits of wider 16-bit elements.
vsri.16 q2, q3, #11 @ shift blue elements right and insert into
@ red and green elements.
```

بشكل مختصر، من أجل كل قناة:

- يتم عمل طول كل عنصر 16-bit ثم إزاحة معطيات اللون إلى البتات الأكثر وزناً.
- إجراء عملية الإزاحة إلى اليمين مع الإدخال لتوضيح كل قناة لونية ضمن مسجل النتيجة.

3.5. تعليمات مبادلة الأشعة Instructions to permute vectors

3.5.1 مقدمة Introduction

تعتبر عمليات المبادلة permutations، أو ما يعني تغيير ترتيب العناصر ضمن الشعاع، ضرورية أحياناً في معالجة الأشعة عند عدم تطابق التعليمات الحسابية الموجودة مع صيغة المعطيات الموجودة

ضمن المسجلات. حيث تقوم باختيار عناصر غير مترابطة إما من مسجل واحد أو من خلال مسجلات متعددة لتشكيل شعاع جديد بحيث يطابق بشكل أفضل تعليمات NEON التي يزودها المعالج.

تشابه تعليمات المبادلة Permutation تعليمات النقل move، من حيث أنها تقوم بتهيئة وإعادة ترتيب المعطيات أكثر من تعديل قيمها. يمكن للخوارزميات المصممة بشكل جيد أن تلغي الحاجة إلى إعادة ترتيب المعطيات. وبالتالي يجب الأخذ بالحسبان ما إذا كان هناك حاجة لاستخدام تعليمات permutation ضمن الرماز [19].

3.5.2 البدائل Alternatives

هناك العديد من الطرائق البديلة لتجنب إعادة الترتيب غير الضرورية:

- إعادة ترتيب معطيات الدخل. وتعدّ من دون تكلفة أن تتم عملية تخزين المعطيات ضمن صيغة ملائمة، حيث تلغى الحاجة إلى إعادة الترتيب عند التحميل والتخزين. ومع ذلك يجب الأخذ بالحسبان موقع المعطيات والتأثير على أداء الخابية قبل تغيير بنى المعطيات.
- إعادة تصميم الخوارزمية. حيث يمكن أن توجد خوارزمية مختلفة والتي تستخدم عدد مماثل من خطوات المعالجة لكن تتعامل مع المعطيات بصيغة مختلفة.
- تعديل درجة المعالجة السابقة. يمكن بإجراء تغيير طفيف لدرجة معالجة سابقة، كضبط الطريقة التي يتم تخزين المعطيات في الذاكرة، أن يقلل أو يلغي الحاجة إلى عمليات إعادة الترتيب.
- استخدام التحميلات والتخزينات المتشابهة. وذلك باستخدام تعليمات تحميل وتخزين قادرة على التشابك وإلغاء التشابك. فإذا لم تلغ الحاجة إلى إعادة التكرار يمكنها عندئذ أن تقلل عدد التعليمات الإضافية اللازمة.
- دمج المقاربات. حيث أن استخدام هذه التقنيات بشكل كبير يمكن أن يكون أكثر كفاءة من تعليمات إعادة الترتيب الإضافية.

إذا لم يتم الحصول على الصيغة المناسبة بالرغم من أخذ النتائج السابقة جميعها بالحسبان، عندها يجب استخدام تعليمات المبادلة.

أ- النقل Move والتبديل VMOV and VSWP Swap

تعدّ التعليمتان VMOV و VSWP أبسط تعليمات المبادلة permute instructions، حيث تقوم بنسخ محتويات كامل المسجل إلى مسجل آخر، أو تقوم بتبديل swap القيم بين زوج من المسجلات. على الرغم من ذلك يمكن عدم النظر إليها على أنها تعليمات إعادة ترتيب، حيث يمكن استخدامها لتغيير القيم ضمن مسجلات D التي تشكل مسجل Q. فعلى سبيل المثال تقوم VSWP d0, d1 بمبادلة البتات الـ 64-bit الأكثر والأقل وزناً من q0.

ب- العكس VREV Reverse

تقوم VREV بعكس ترتيب عناصر 8-bit، 16-bit، و 32-bit ضمن الشعاع، ويوجد لها ثلاث أشكال مختلفة:

VREV16 تعكس كل زوج من العناصر الفرعية 8-bit مشكلة عناصر 16-bit ضمن الشعاع.

VREV32 تعكس أربعة عناصر فرعية من 8-bit أو اثنين من 16-bit مشكلة عناصر 32-bit ضمن الشعاع.

VREV64 تعكس ثمانية عناصر 8-bit، أو أربعة 16-bit أو عنصرين 32-bit ضمن الشعاع.

تستخدم VREV لعكس نهايات endianness المعطيات، أو لمبادلة مركبات الألوان أو لتبادل القنوات لعينات الصوت.

ج- الاستخراج VEXT Extract

تستخرج VEXT شعاع بايتات جديد من زوج من الأشعة الفعلية. حيث أن البايتات في الشعاع الجديد هي من أعلى المعامل الأول، ومن أسفل المعامل الثاني. وهي تسمح بتوليد شعاع جديد يحتوي على عناصر ناتجة عن مبادلة أشعة موجودة.

يمكن استخدام VEXT لإنجاز نافذة متحركة على معطيات من شعاعين، وهي مفيدة في مرشحات FIR. من أجل المبادلة permutation، يمكن استخدامها لمحاكاة عمليات الدوران المتعلقة بالبايتات byte-wise rotate operation عندما يتم استخدام نفس الشعاع لكل من معاملي الدخل.

د - تبديل الموضع VTRN Transpose

تقوم VTRN بتبديل موضع العناصر 8-bit، و 16-bit و 32-bit بين زوج من الأشعة. وتعامل عناصر الأشعة كمصفوفات 2x2 وتقوم بتبديل موضع كل مصفوفة.

يمكن استخدام عدة تعليمات VTRN لتبديل موضع المصفوفات الأكبر. على سبيل المثال تتكون مصفوفة 4x4 من عناصر 16-bit يمكن تبديل موضعها باستخدام تعليمات VTRN.

وهي العملية نفسها المطبقة من قبل كل من VLD4 و VST4 بعد التحميل أو قبل تخزين الأشعة. وبسبب طلبها لتعليمات أقل، ينصح باستخدام ميزات بنى ولوج الذاكرة هذه لتخصيص تتابع من تعليمات VTRN عندما يكون ذلك ممكناً.

هـ - الضغط Zip وفك الضغط VZIP and VUZP Unzip

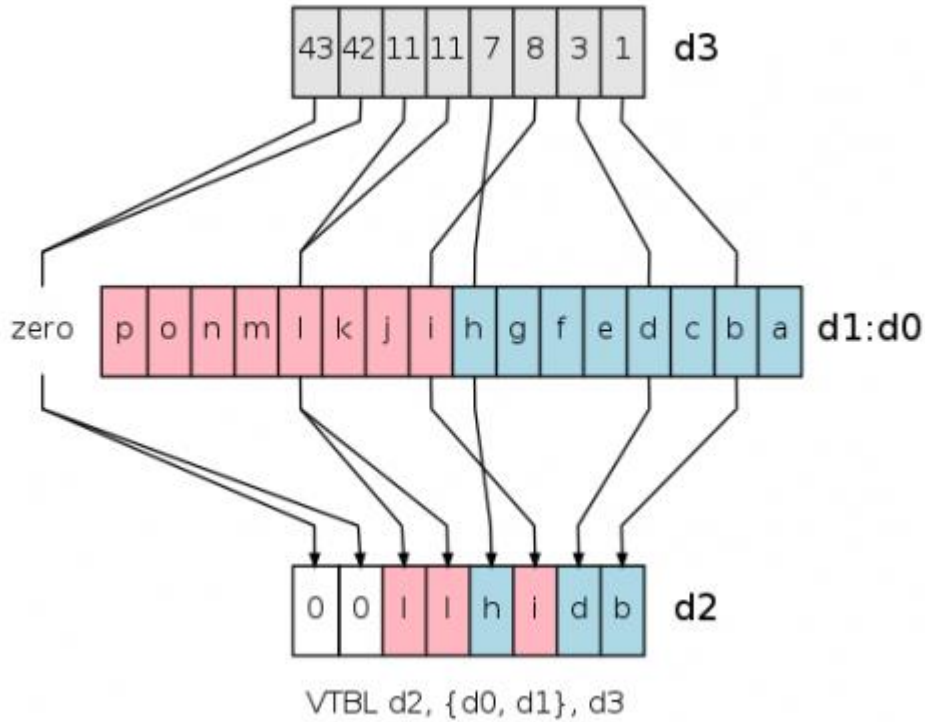
تقوم VZIP بإلغاء تشابك عناصر 8-bit، أو 16-bit أو 32-bit لزوج من الأشعة. وهذه العملية هي نفسها المنفذة من قبل VST2 قبل التخزين، لذا يمكن استخدام VST2 بدلاً من VZIP إذا لزم ضغط المعطيات مباشرة قبل كتابتها إلى الذاكرة.

تعمل VUZP عكس VZIP، وذلك بإلغاء تشابك عناصر 8-bit، أو 16-bit أو 32-bit من أزواج الأشعة. وهذه العملية هي نفسها المنفذة من قبل VLD2 بعد التحميل من الذاكرة.

و - الجدول Table والجدول الموسّع VTBL, VTBX Table Extend

تقوم VTBL بإنشاء شعاع جديد من جدول من الأشعة بالإضافة إلى شعاع الفهرس index vector. وهي عملية بحث بالجدول متعلقة بالبايتات byte-wised table lookup operation.

يتكون الجدول من مسجل إلى أربعة مسجلات D متجاورة، ويستخدم كل بايت ضمن شعاع الفهرس لفهرسة بايت ضمن جدول الأشعة. حيث يتم إدخال القيمة المفهرسة إلى شعاع النتيجة ضمن الموضع الموافق لموقع الدليل الأساسي ضمن شعاع الفهرس.



الشكل 18-0 مبدأ عمل مسجل الفهرسة

تختلف كل من VTBL و VTBX في طريقة التعامل مع الفهارس خارج النطاق out-of-range indexes. فإذا تجاوز الفهرس طول الجدول، تقوم VTBL بإدخال قيمة صفرية للمكان المتعلق ضمن شعاع النتيجة، بينما تترك VTBX القيمة الموجودة ضمن شعاع النتيجة دون أي تغيير. الشكل 18-0

في حال استُخدم شعاع مصدر وحيد كجدول، فإن VTBL تسمح في إنجاز مبادلة الشعاع بشكل اعتباطي، وذلك على حساب اعداد مسجل الفهرسة index register. وإذا تم استخدام العملية ضمن حلقة ولم يتغير نوع إعادة الترتيب، عندها يمكن تهيئة مسجل الفهرسة خارج الحلقة، وإلغاء تكاليف التهيئة غير المباشرة.

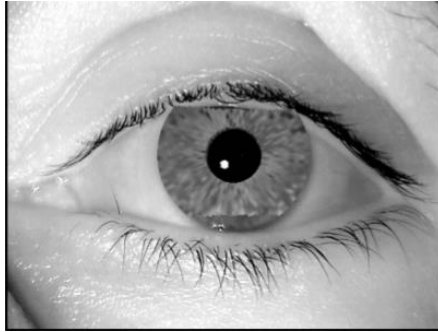
الفصل الرابع

التعرّف على قزحية العين

Iris Recognition

4.1. مدخل إلى تحديد مكان قزحية العين Introduction to Iris Localization

التعرف على قزحية العين أو Iris recognition هي تقنية مؤتمتة من نظم تحديد الهوية البيومترية biometric identification والتي تستخدم تقنيات أنماط التعرف الرياضية - mathematical pattern-recognition techniques على صور القزحيات لأعين الأفراد، والتي تحتوي على أنماط عشوائية معقدة فريدة، ويمكن رؤيتها من مسافة قريبة. وميزة استخدام قزحية العين كسمة بيومترية، أنها عضو داخلي محمي مع بقاء نمطها النسيجي العشوائي من دون تغيير خلال كامل حياة الفرد. يبين الشكل 1-0 صورة عن عينة قزحية العين.



الشكل 1-0 صورة عينة لقزحية العين

تستخدم تقنية التعرف على قزحية العين تكنولوجيا الكاميرا ذات إضاءة دقيقة بالأشعة تحت الحمراء للحصول على صور بتفاصيل عالية من بنى قزحية العين المعقدة. تسمح القوالب الرقمية المرمزة من هذه الأنماط من خلال الخوارزميات الرياضية والإحصائية بتحديد هوية الفرد. يتم البحث ضمن قواعد المعطيات للقوالب المدخلة من خلال محركات المطابقة matcher engines بسرعات تقاس بملايين القوالب بالثانية بوحدة معالجة مركزية واحدة CPU، وبمعدلات خطأ بالمطابقة متناهية في الصغر.

تم إدخال قزحيات أعين الملايين من الأشخاص من بلدان مختلفة حول العالم إلى نظم التعرف على قزحية العين لأغراض السهولة والراحة، مثل نظم عبور الحدود المؤتمتة بدون الحاجة لاستخدام جواز السفر، بالإضافة إلى تطبيقها ضمن نظم تحديد الهوية الوطنية المعتمدة على هذه التكنولوجيا. إن الميزة الأساسية لتقنية التعرف على قزحية العين، إلى جانب سرعتها في المطابقة وممانعتها العالية لأخطاء التتابق، هي مقاومة القزحية كونها عضو داخلي من العين، محمي وظاهر خارجياً.

تشتهر قزحية العين كسمة بيو مترية لاحتمالية عدم تغيرها مع تقدم السن والحالة النفسية بالإضافة إلى عدد كبير من العوامل الخارجية. كما تستخدم لتحديد هوية الأشخاص على مستوى قواعد المعطيات الوطنية والدولية كمقياس أمني لعمليات التحقق والمصادقة البيو مترية. بينما تتأثر السمات البيو مترية الأخرى مثل التوقيع signatures والكلام speech بالعوامل النفسية، وهنا يبرز دور القزحية البيو متري لحل المشكلة لثبات نسيجها العشوائي على امتداد دورة حياة الفرد. علاوة على ذلك فهي عضو محمي خالي من التخريب الخارجي وبالتالي فهي أكثر وثوقية.

انتشر حالياً استخدام خوارزميات التعرف على قزحية العين عالمياً ضمن العديد من الأجهزة المتنوعة مروراً بالحواسب الشخصية إلى ماسحات قزحية العين المحمولة portable Iris scanners. وتستخدم هذه الأجهزة أنظمة تعتمد على وحدة المعالجة المركزية CPU التي تم تصميم الخوارزميات عليها بالأساس.

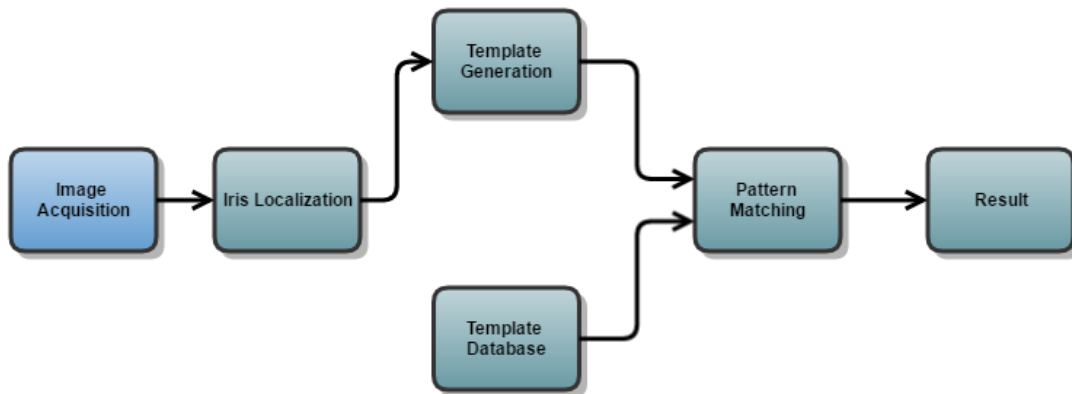
يُنظر للنظم المعتمدة على وحدة المعالجة المركزية على أنها آلات ذات استخدام عام، مصممة لجميع أنواع التطبيقات. تعرف كتجهيزات معالجة تتابعية sequential processing devices. وفيها يتم جلب التعليمات أولاً، ثم فك ترميزها وفي النهاية تنفيذها من خلال وحدات الحساب والمنطق ALUs. تحوي المعالجات المتطورة أكثر من وحدة حساب ومنطق ALU واحدة، وبالتالي يمكن تنفيذ عدد من التعليمات على التوازي. ومع ذلك تكون المعالجات الحديثة مقيدة بعدد وحدات الحساب والمنطق ALU التي تملكها، ومعظم وحدات المعالجة المركزية في يومنا هذا لا تملك أكثر من أربع معالجات وبالتالي فإن استخدام وحدات معالجة الرسوميات GPUs يمكن أن يكون حلاً بديلاً لتسريع التطبيق.

تحديد هوية القزحية البيو متري Iris biometric identification هو عملية تعرّف على نمط pattern recognition تتطوي على الخطوات التقليدية لمراحل ما قبل المعالجة preprocessing، استخراج الخصائص feature extraction، تمثيل الخصائص feature representation، وتحديد الهوية identification. وجميع هذه الخطوات تزيد من تعقيد الحوسبة computation complexity، إلا أنه يمكن تحسين كفاءتها بشكل جذري باستخدام الخوارزميات والمعماريات المتوازية. وبالتالي تزداد أهميتها عندما يتطلب تنفيذ عمليات تحديد الهوية أو المصادقة على مستوى قواعد المعطيات الوطنية. يمكن لقواعد المعطيات ذات المستوى الوطني أن تحتوي على الملايين من قوالب المطابقة، حيث تستهلك كل واحدة منها معالجة كم هائل من البيكسلات، وبالتالي تكون الحوسبة المتوازية هي الحل الوحيد البديل.

لا ينحصر استخدام وحدات معالجة الرسوميات GPUs الحديثة فقط على تصيير الرسوميات Graphics Rendering والتصور visualization بل تستخدم أيضاً للتعامل مع حسابات الأغراض العامة الأخرى التي تتطلب كم ثقيل من الحوسبة. وبالتالي تسمى أحياناً بوحدات معالجة الرسوميات للأغراض العامة General Purpose Graphics Processing Units أو GPGPUs. وتظهر أهمية الـ GPGPUs في الأبحاث التي يحتاج تنفيذ خوارزمياتها العامة زمناً كبيراً يحول دون الحصول على نتائج سريعة.

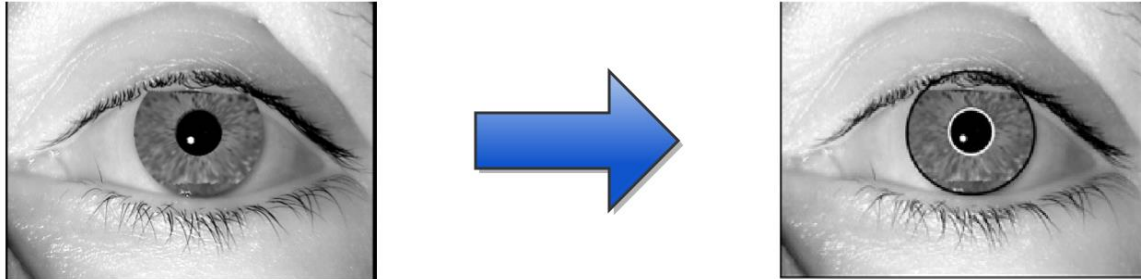
تقسم عملية التعرف على القزحية بشكل أساسي إلى مراحل تحصيل الصورة Image acquisition، تحديد مكان القزحية Iris Localization، توليد القالب Template generation ثم المطابقة Matching كما هو موضح في الشكل 2-0.

يعتمد الأداء الكلي في التعرف على قزحية العين بشكل كبير على أداء خوارزمية تحديد المكان والتي تشكل المرحلة الأولية في العملية من أجل كشف التطابق للصورة المأخوذة أمام القوالب المتوفرة في قاعدة المعطيات. تتكون مرحلة توليد القوالب Template generation من توليد شعاع خصائص feature vector فريد للصورة وذلك من خلال تحليل أنماط النسيج المتواجدة ضمن صورة القزحية. وتشمل مرحلة تحديد مكان القزحية، إيجاد الحدود الداخلية والخارجية للقزحية من أجل فصل مناطق الصورة المأخوذة والتي ليس لها صلة في المعالجة التالية.



الشكل 2-0 المراحل المستخدمة ضمن التعرف على قزحية العين

يظهر الشكل 0-3 كلاً من صورة العينة الأصلية على اليسار وصورة القزحية المحدد موقعها على اليمين. حيث أن الجزء ذو الصلة من الصورة هو المتوضع بين خارج البؤبؤ وداخل الحدود الخارجية للقزحية. كما ينبغي أيضاً إزالة جزء القزحية المغطى بالجفون.



الشكل 0-3 نتائج تحديد موقع القزحية بدون إزالة الجزء المغطى من قبل الجفون

4.2. دراسة مرجعية

في عام 1987 تم تقديم مفهوم التعرف الآلي على قزحية العين automated Iris recognition من قبل ليونارد فلوم وآران سفير [20]، وهما أستاذان في طب العيون، وحصلوا في وقت لاحق على براءة اختراع بعنوان "تقانة التعرف على قزحية العين" "Iris Recognition Technology". اقترحا بعدها على الأستاذ جون دوغمان [21] [22] الأستاذ بجامعة هارفارد كتابة خوارزمية تعتمد على مبدئهما الذي قام بإنشائها وحصل عليها على براءة اختراع بعد عدة سنوات. وبعد انتهاء براءة الاختراع تم إنشاء عدة خوارزميات تعتمد على طريقة دوغمان، حقق بعضها أداءً أفضل من خوارزمية دوغمان.

4.3. التعرف على قزحية العين Iris Recognition

تحتوي قزحية عين الإنسان على نسيج ذو أشكال نمطية معقدة complex random texture patterns والتي يمكن استخراجها من الصورة الرقمية لعين الإنسان ثم ترميزها ضمن قالب template باستخدام تقنيات معالجة الصورة الرقمية. حيث يمكن تخزين هذا القالب داخل قاعدة معطيات. ويحتوي القالب على تمثيل رياضي mathematical representation للمعلومات الفريدة المخزنة ضمن القزحية Iris وهذا يسمح بالمقارنة مع القوالب من قاعدة المعطيات.

تم تطوير أول نظام قابل للعمل operational للتعرف على قزحية العين من قبل دوغمان في جامعة كامبردج عام 1994، حيث استخدم دوغمان مرشح موجيات غابور ثنائية البعد التريبيعية متعددة القياس multi-scale quadrature 2D Gabor wavelets لاستخراج معلومات النسيج من أجل توليد نمط شكلي

ذو 2048 بت يدعى رمز القزحية IrisCode [21]. وتعتمد الأنظمة التجارية الموضوعة في الخدمة جميعها حالياً على هذه الطريقة.

كملخص يمكن إنجاز طريقة دوغمان من خلال:

- معامل التفاضل التكاملي لكشف الحدود الداخلية والخارجية An Integro differential operator to detect the inner and outer boundaries
- مرشح موجات غابور ثنائية البعد التريبية متعددة القياس لاستخراج معلومات النسيج بصيغة أشعة إثنائية فريدة مشكّلة رمّاز القزحية Multi-scale quadrature 2D Gabor wavelets to extract texture information in form of unique binary vectors constituting the IrisCode
- مطابق إحصائي (معامل XOR منطقي) والذي يقوم بحساب مسافة هامينغ بين رمّاز القزحية المدخل ورمّازات قزحية العين في قاعدة المعطيات (logical XOR operator) which finds out the hamming distance between the input IrisCode and the IrisCodes in the database

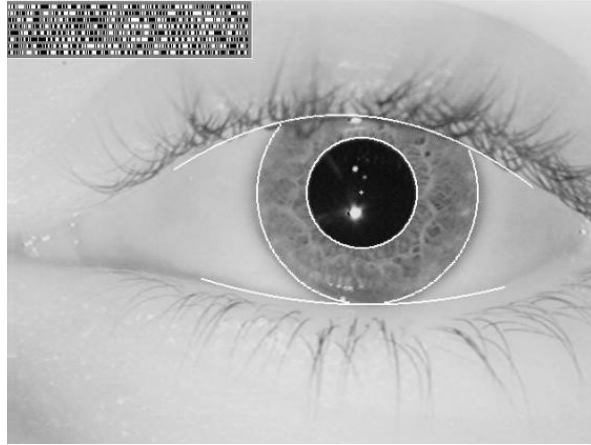
ومن الأعمال المهمة بعد عمل دوغمان، قام وايلد وآخرون [23] بتمثيل النسيج المعروض ضمن قزحية العين من خلال إنشاء أهرامات لابلاسيان laplacian pyramids مع قيم تمييزية مختلفة different resolution values. حيث قاموا باستخدام الترابط المنتظم normalized correlation لتبيان فيما إذا كانت الصورة المدخلة من صف class صورة النموذج model image نفسه.

يتكون حل R.Wilde من الخطوات التالية:

- تحويل هوف لتحديد مكان القزحية A Hough Transform for Iris localization
- هرم لابلاسيان (التحلل متعدد القياس) لتمثيل الخصائص المكانية المميزة لقزحية عين الإنسان Laplacian pyramid (multi-scale decomposition) to represent distinctive spatial characteristics of human Iris
- عملية الترابط المنتظم المعدّل من أجل خطوة الموائمة Modified normalized correlation process for the matching step.

4.4. تحديد مكان قزحية العين Iris Localization

يشكل تحديد مكان قزحية العين المرحلة الأولى من عملية التعرف على قزحية العين Iris recognition، بهدف عزل منطقة الاهتمام region of interest الممثلة ضمن العين. وتمثل عادةً جفون eyelids ورموش العين eyelashes ضمن الأجزاء الساترة occluding parts لمنطقة القزحية Iris region. يبين الشكل 4-0 صورة قزحية عين إنسان محدّدة تم الحصول عليها بعد إزالة الجفون و الرموش.



الشكل 4-0 تحديد مكان قزحية العين بعد إزالة الجفون

كانت الخوارزميات الأصلية المستخدمة في التعرف على قزحية العين في الأساس تسلسلية، أي تم تصميمها للعمل على الأنظمة التقليدية المعتمدة على وحدة المعالجة المركزية CPU، من قبل دوغمان، ثم اتخذ مناخ مختلفة حتى الوقت الحالي.

تتألف عملية التعرف على قزحية العين بحسب طريقة دوغمان من ثلاث خطوات:

الخطوة الأولى: تحصيل صورة العين باستخدام كاميرا فيديو رقمية مع استخدام اضاءة قريبة من الأشعة تحت الحمراء لإظهار تفاصيل أكثر بالنسبة للقزحيات الداكنة [21].

الخطوة الثانية: تحديد موقع قزحية العين Iris localization والتي تقوم باستخراج قزحية العين من الصورة الخام، سيتم عرض هذه التقنية في فقرات اللاحقة مع بعض الطرق الأخرى التي تم مناقشتها. تستخدم بعدها موجات غابور Gabor wavelets لاستخراج الخصائص feature extraction لتوليد رمّاز اثنائي من 2048 بت اسمه رمّاز قزحية العين IrisCode.

الخطوة الثالثة: التطابق matching.

4.4.1. طريقة دوغمان Daugman's Method

اقترح دوغمان مؤثر التفاضل التكاملي integro-differential operator بهدف تحديد موقع قزحية العين Iris localization الذي يقوم بالبحث عن القيم العظمى ضمن المشتقات الجزئية غير الواضحة blurred partial derivative للصورة على طول القوس الدائري لنصف القطر r والمركز (x_0, y_0) ، مع الأخذ بالحسبان نصف قطر محيط التكامل المنتظم normalized contour integral للصورة [24] [22].

$$\max(r, x_0, y_0) | G_\sigma(r) * \frac{\partial}{\partial r} \int_{(r, x_0, y_0)} \frac{I(x, y)}{2\pi r} ds | \quad (2-4)$$

حيث $I(x, y)$ تمثل الصورة و $G_\sigma(r)$ نواة التنعيم smoothing kernel، يمكن إعطاء مستويات مختلفة من التنعيم من خلال قيم مختلفة لـ σ . يتم معالجة حقيقة أن بؤبؤ العين لا يكون دائماً داكناً من قزحية العين من خلال أخذ القيمة المتوسطة للمشتق الجزئي. يتصرف المعامل بشكل كلي كما لو كان كاشف حافة دائري circular edge detector. يوجد أيضاً بعض من طرق التقطيع التي اكتسبت شهرة واسعة [25] [26].

لا تأخذ عملية التعرف على قزحية العين لون القزحية بالحسبان على الرغم من وجود تفاوت كبير في لون القزحية يتراوح من الأسود إلى البني إضافة إلى اللون المزرق إلى المخضر، وبدلاً من ذلك تركز طريقة دوغمان على أنماط النسيج مثل الأخاديد furrows، والخبايا crypts، والخواتم rings والإكليل corona. بعد مرحلة تحديد مكان القزحية Iris Localization، يتم حساب رماز 2048 بت من أجل قزحية العين Iris ثم مقارنته مع القوالب الممثلة ضمن قاعدة المعطيات. حيث يتم ترميز الأنماط باستخدام كشف تعديل غابور ثنائي البعد 2D Gabor demodulation. يتألف رماز القزحية IrisCode من 2048 بت من المعطيات بالإضافة إلى بتات الأفعنة الـ 2048 مشكّلةً بذلك رماز القزحية IrisCode بـ 512 بايت.

4.4.2. مخطط وايلد وآخرون Wilde et al.'s Scheme

في مخطط وايلد [23] [27]، يتم تحويل معلومات كثافة الصورة image intensity information إلى خريطة حافة إثنائية binary edge map أولاً. بعدئذٍ تقوم نقاط الحافة بالتصويت لاستنتاج قيم مُوسط الحافة الفعلية particular contour parameter values. ويتم الحصول على خارطة الحافة باستخدام

مخطط كشف الحافة الذي يعتمد على التدرج gradient based edge detection scheme من خلال استيفاء الحد الأدنى لمطال تدرج كثافة الصورة image intensity gradient

$$\nabla G(x, y) * I(x, y), \quad \text{where } \nabla = (\partial/\partial x, \partial/\partial y) \quad (2-4)$$

و

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}} \quad (3-4)$$

وينفذ التصويت voting procedure باستخدام تحويلات هوف Hough Transforms [28] [29] [30] على المحددات المُوسَّطة لحواف القزحية المحيطية iris boundary contours.

يعطى تحويل هوف بالمعادلة التالية:

$$H(x_c, y_c, r) = \sum_{j=1}^n h(x_j, y_j, x_c, y_c, r) \quad (4-4)$$

حيث:

$$h(x_j, y_j, x_c, y_c, r) = \begin{cases} 1 & \text{if } g(x_j, y_j, x_c, y_c, r) = 0 \\ 0 & \text{otherwise} \end{cases} \quad (5-4)$$

و

$$g(x_j, y_j, x_c, y_c, r) = (x_j - x_c)^2 + (y_j - y_c)^2 - r^2 \quad (6-4)$$

4. 3. مخططات أخرى Other Schemes

يستخدم ريتز وآخرون [31] نماذج الإطار الفعّال active contour models لتحديد مكان بؤبؤ العين. وعمل بعض المختصين على زيادة تحسين أداء تحديد الموقع من خلال كشف جفون ورموش العين [32] لتحسين الزمن اللازم لتحديد الموقع localization. ينظر طور ما قبل المعالجة preprocessing phase، والذي يتعامل مع تحديد الموقع localization وتقييس normalization صورة القزحية، وبؤبؤ العين والحدود الخارجية للقزحية يجب أن تكون دائرية الشكل. وقد أجريت أعمال كثيرة لكشف جفون ورموش العين.

اقترح خوان وآخرون [33] تقنية لتحسين زمن تحديد موقع القزحية iris localization وذلك من خلال كشف الحدود الخارجية من القزحية ضمن الصورة المعاد توسيعها rescaled image، حيث أن إعادة ضبط أبعاد الصورة للجزء الأمثل من الصورة الأصلية يقوم بأداء أفضل من الصورة الأصلية. كما قام العديد من الباحثين الآخرين باقتراح طرائق لكشف جفون ورموش العين، حيث يتضمن كشف الجفون البحث عن منحنيين، واللذان يحققان معادلة كثيرة الحدود من الصيغة:

$$x(t) = at^2 + bt + c, \quad t \in [0,1] \quad (7-4)$$

يمكن كشف رموش العين من خلال فحص التباين variance لكل كتلة. وقام أيضاً بعض الباحثون باستخدام المناهج المعتمدة على العتبة threshold based approaches لإيجاد التحديد الخشن لبؤبؤ العين coarse localization of pupil. كون البؤبؤ هو المنطقة الداكنة، يمكن فصل حدود البؤبؤ بتطبيق عتبة محددة certain threshold على الكثافة intensity في الصورة.

واقترح كل من كونغ وجانغ [34] طريقة لكشف رموش العين. حيث قاما بتقسيم رموش العين إلى نوعين. النوع الأول هو الرموش القابلة للفصل separable eyelashes والتي يمكن فصلها بسهولة من الصورة، أما النوع الثاني فهو الرموش المتعددة multiple eyelashes والتي تكون متزاحمة bunched معاً ومتداخلة overlapped في صورة العين. يمكن استخدام مرشحات غابور أحادية البعد one dimensional Gabor filters لكشف الرموش القابلة للفصل لأن التنعيم الغوسي Gaussian smoothing للرموش القابلة للفصل يؤدي إلى قيمة خرج منخفضة low output value. وهكذا فإن النقاط الناتجة تكون أصغر من النقاط الممثلة لرموش العين. تستخدم طريقة تنوع الكثافة variation of intensity لكشف الرموش المتعددة. حيث أنه إذا كان التباين في قيم الكثافة ضمن النافذة أقل من قيمة العتبة عندها مركز النافذة هو نقطة على رموش العين.

4.5. منهج موازاة التعرف على قزحية العين Parallel

Approach to Iris Recognition

بدأ العمل في التعرف على قزحية العين المتوازي parallel iris recognition بشكل متأخر جداً. حيث حصل على دفعة boost مع تطور البنية الأساسية للحوسبة المتوازية parallel computing infrastructure مثل معمارية النوى المتعددة multicore architecture والتقنيات الجديدة لتطوير البرامج المتوازية parallel programs. ولقد اقترح دوغمان نفسه تقديم الموازاة parallelism ضمن مرحلة المطابقة لرماز القزحية IrisCode المولدة من الصورة مع قوالب رمازات القزحية المقدمة في قاعدة المعطيات.

تتضمن عملية المطابقة matching process مقارنة رمازات القزحية IrisCodes من خلال استخدام عمليات XOR المنطقية لحساب تابع يدعى مسافة هامينغ Hamming distance، والذي يقوم بإيجاد مقدار عدم التماثل dissimilarity بين أي من رمازي القزحية [21] [24].

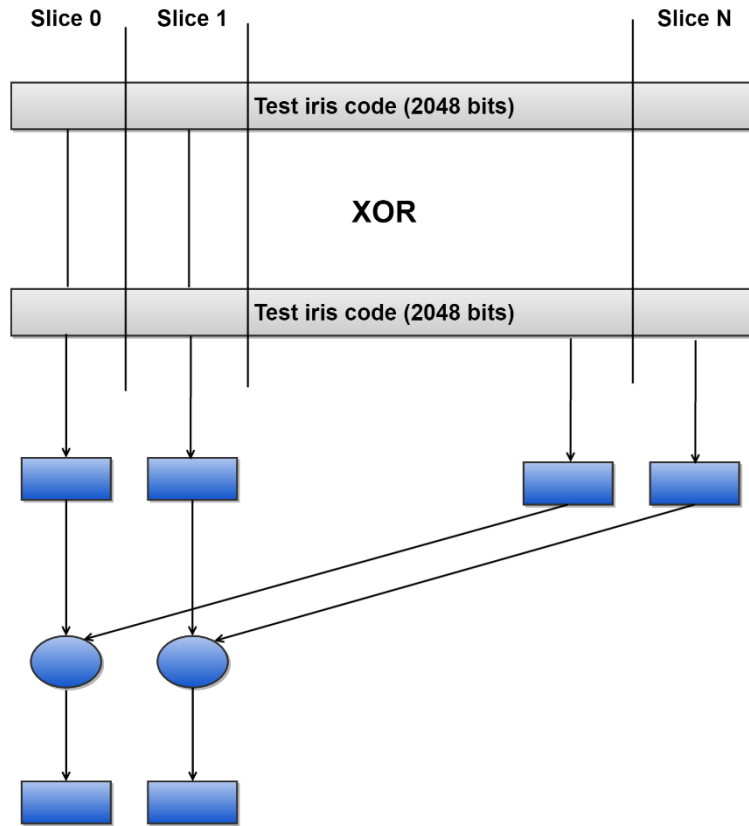
$$HammingDistance = \frac{\|(CodeA \oplus CodeB) \cap MaskA \cap MaskB\|}{\|MaskA \cap MaskB\|} \quad (8-4)$$

يمكن حساب مسافة هامينغ بسرعة كبيرة جداً نظراً لطريقة الحاسوب في تنفيذ العمليات عتادياً بت مقابل بت bitwise operations. تنفذ العمليات المختصة بالبتات مباشرةً على مستوى الكيان الصلب hardware level وهذه الطريقة فعالة جداً very efficient. حيث تحتاج عملية مقارنة رماز القزحية مع الملايين من القوالب إلى 1.7 ثانية باستخدام حاسوب تردد عمله 2.2 GHz، وهذا المستوى من الأداء يسمح بتطبيق نظم التعرف على قزحية العين على الكثير من التطبيقات large scale applications حيث ينبغي أن يتم مقارنة رماز القزحية IrisCode مع الملايين من القوالب. وطالما أن حجم رماز القزحية IrisCode هو 2048 بت فيمكن تنفيذ عمليات المقارنة على التوازي بمقدار قطع chunks بقياس 32 بت.

ظهر مؤخراً العديد من الأعمال الملحوظة في تطبيق منهج الموازة parallel approach بهدف حل مشاكل التعرف على قزحية العين. إذ قدمت فاطمة زكي صقر وآخرون [35] [36] في عام 2011 طريقة بديلة استخدمت المعالجة المتوازية مباشرة direct parallel processing على وحدات معالجة الرسومات Graphical Processing Units (GPUs) للتعرف على قزحية العين عالي الأداء وذلك بعد تفكيك المراحل الأكثر استهلاكاً للوقت والمشاركة في عملية التعرف على القزحية. وقاموا بالتركيز على مراحل المطابقة matching وتحديد الهوية identification الذي يتطلب قدراً كبيراً من الحوسبة إذا كان حجم قاعدة المعطيات اللازمة للمقارنة عالياً جداً. وقاموا بتطبيق خوارزمية دوغمان ضمن كل خطوة من العملية، وبلغ مقدار التسريع الذي تم الحصول عليه 10 إلى 15 مرة في طوري المطابقة وتحديد الهوية، وحوالي 1.3 مرة عند أخذ جميع مراحل التعرف على القزحية بالحسبان. وأنجزوا النظام على وحدة معالجة الرسومات NVidia GTX 460 ذات 336 نواة.

تتألف عملية تحديد الهوية identification process وفقاً لمنهجية دوغمان من مطابقة الـ 2048 بت لرمازات القزحية IrisCodes للصورة وذلك من خلال حساب مسافة هامينغ بين الرماز المدخل ورماز القوالب. استكشف [35] هذه الحقيقة من خلال مطابقة الرمازات على التوازي. حيث تم تقسيم بتات

الرماز إلى شرائح bit slices وتمت مقارنة كل شريحة من الشرائح مكونة من 32 بت على معالج وحيد بعملية XOR بين الرماز المدخل وبين رماز القوالب كما هو موضح في الشكل 5-0.



الشكل 5-0 حوسبة متوازية لمسافة هامينغ على وحدة معالجة الرسومات GPU

وفي عام 2012، توصلوا في بحث آخر بعنوان "تسريع خوارزمية التعرف على قزحية العين باستخدام وحدة معالجة الرسومات" "Accelerating Iris Recognition Algorithm on GPU's" ركزوا فيه على مرحلتي تحديد الموقع والاستخراج الأكثر استهلاكاً للوقت مقارنة بمرحلة المطابقة. وقاموا مرة ثانية بإنجاز نظام التعرف على قزحية العين اعتماداً على نظام دوغمان استخدموا فيه وحدة معالجة الرسومات السابقة نفسها، وحصلوا على تسريع بمقدار 9.6 و 12.4 مرة لمرحلتي المعالجة على التوالي وبتسريع كلي بمقدار 12.4 مرة عند أخذ جميع العمليات بالحسبان.

وفي منهج آخر، قدم ريان وآخرون [37] [38] معالجة متوازية بديلة باستخدام صفائف البوابات القابلة للبرمجة حقيقياً (FPGAs) التي سرعت التطبيق لمستوى أفضل بكثير. حيث قاموا باستخدام المعالجة المتوازية لتسريع الخطوات الأكثر استهلاكاً للزمن المتعلقة في عملية التعرف على قزحية العين، وهي خطوات تحديد مكان قزحية العين، وتوليد القالب والمطابقة والتي

تتطلب العديد من مراحل الحوسبة، لذا تم إنجاز الموازة على نظام FPGA مقدماً، وحصلوا على تسريع مرحلة تحديد الموقع بمعدل 9.6 مرة، و 324 لمرحلة توليد القالب و 19 مرة لمرحلة المطابقة مقارنة بنظام يستخدم وحدة المعالجة المركزية CPU. إن شرائح FPGA هي عبارة عن تجهيزات منطقية قابلة للبرمجة حقلياً يمكن برمجتها حسب المتطلبات اللازمة.

4.6. الخلاصة Summary

هناك الكثير من الأعمال المنجزة ضمن مجال التعرف على قزحية العين اعتمد معظمها على الأنظمة التتابعية المبنية على وحدة المعالجة المركزية CPU. وبالمحصلة تم استعراض الكثير من الأبحاث مع التركيز على الاستفادة القصوى من وحدة معالجة الرسومات GPU لغرض تحديد الموقع localization واستخراج الخصائص feature extraction كونها تمثل الأجزاء التي تحتاج إلى الكثير من عمليات الحوسبة المركزة computation intensive والتي تحتاج إلى وقت كبير في عملية التعرف على قزحية العين.

الفصل الخامس

تحديد مكان قزحية العين المتوازي

Parallel Iris Localization

الهدف

تعدّ معظم تقنيات تحديد مكان قزحية العين Iris Localization techniques المطورة إلى الآن تتابعية sequential بطبيعتها. والهدف من هذه الأطروحة هو إنجاز تقنية مستخدمة في معالجة الإشارة الرقمية، حيث تم اختيار تقنية تحديد مكان قزحية العين ضمن أنظمة التعرف على القزحية، على نظام غير متجانس نُقال لتسريع المهمة. لم يتم أخذ دقة الخوارزمية بالحسبان طالما تم اختبارها من قبل مطوريها. تم اختيار طريقة تحويل هوف Hough transform من أجل عملية تحديد مكان القزحية ومن أجل كشف الحدود الدائرية circular boundaries، وباستخدام أداة التطوير Jetson TK1 من شركة NVidia، وهي نظام مضمّن غير متجانس ذو استهلاك منخفض للطاقة ويحتوي على وحدة معالجة مركزية بأربع نوى ARM A15 ووحدة معالجة رسومات GPU بمعمارية CUDA، كنظام غير متجانس من أجل الإنجاز. ويعود السبب لاختيار تحويل هوف كونه طريقة متينة وفعالة عند التعامل مع معطيات ناقصة وملئية بالضجيج على عكس الطرائق المعتمدة على العتبة threshold.

5.1. تحويل هوف التقليدي Traditional Hough Transform

يعدّ تحويل هوف التقليدي Traditional Hough Transform طريقة فعّالة جداً عندما يتعلق الأمر بتعيين هوية الأشكال البسيطة المقدمة في الصورة. في البداية تم استخدام تحويل هوف لإيجاد الخطوط lines ضمن الصورة فقط، لكن تم بعد ذلك لتعيين لتعيين هوية أشكال أخرى مثل الدائرة والقطع الناقص. تم طرح مفهوم تحويل هوف المعمّم من قبل كل من ريتشارد دودا وبيتر هارت [28] في عام 1972، حيث وسمياه تحويل هوف المعمّم Generalized Hough Transform.

يتم تغذية خرج خوارزمية كشف الحافة edge detection algorithm كدخل لمرحلة تحويل هوف، حيث تقوم هذه الخوارزمية بكشف الحواف الحادة للأشكال المختلفة الممثلة ضمن الصورة من خلال تطبيق اختبارات التباين على الصورة. بعدها تقوم الخوارزمية اللاحقة بكشف الشكل الخاص الممثل ضمن الصورة. ويمتاز تحويل هوف بفعاليته العالية للمعطيات المليئة بالضجيج لاحتمالية وجود عدة عيوب imperfections ضمن الصورة بالإضافة إلى خوارزمية كشف الحافة، على حساب التعقيد الحاسوبي العالي جداً.

يعطى تحويل هوف بالمعادلة التالية:

$$H(x_c, y_c, r) = \sum_{j=1}^m h(x_j, y_j, x_c, y_c, r) \quad (1-5)$$

حيث:

$$h(x_j, y_j, x_c, y_c, r) = \begin{cases} 1 & \text{if } g(x_j, y_j, x_c, y_c, r) = 0 \\ 0 & \text{otherwise} \end{cases} \quad (2-5)$$

و

$$g(x_j, y_j, x_c, y_c, r) = (x_j - x_c)^2 + (y_j - y_c)^2 - r^2 \quad (3-5)$$

تم إنجاز العديد من التحسينات على تحويل هوف التقليدي [33].

واستخدم بعض الباحثون مع تحويل هوف كواشف Canny للحافة لتحسين سرعة تحديد المكان [28]. وقام البعض الآخر باستخدام التقنيات التي تعتمد على العتبة لتحديد بؤبؤ العين لكنها لا تعطي دائماً النتيجة المرجوة بسبب التفاوت الكبير بين ألوان القزحية الذي يتراوح بين اللون الفاتح إلى اللون الداكن، إذ تزداد الصعوبة للقزحيات الداكنة أكثر.

يقوم تحويل هوف بتجميع نقاط الحواف في أشكال يمكن التعرف عليها مثل الخط، والدائرة أو القطع الناقص. ولما كان شكل القزحية والبؤبؤ دائري، لذا يتم استخدام تحويل هوف للدوائر [28] [29] لكشف البؤبؤ وحدود القزحية. يتم وصف الدائرة كلياً بواسطة ثلاث متغيرات وهي مركزها (a, b) ونصف قطرها r .

تعطى معادلة متوسطات الدائرة كما يلي:

$$x = a + r \cos \theta \quad (4-5)$$

$$y = b + r \sin \theta \quad (5-5)$$

بتغيير قيمة θ من 0 إلى 360 درجة تتشكل دائرة كاملة. في تحويل هوف الدائري يتوقع وجود ثلاثية (x, y, r) والتي يكون لها احتمالية عالية بأن تكون دائرة ضمن الصورة. على فرض أنه يراد البحث على دوائر بنصف قطر محدد، أي نصف قطر معلوم، عندها كل نقطة ضمن فضاء الصورة يكافئ دائرة ضمن فضاء هوف Hough space أو فضاء المتوسط Parameter space.

بإعادة ترتيب المعادلات السابقة يتم الحصول على المعادلات التالية:

$$a = x - r \cos \theta \quad (6-5)$$

$$a = y - r \sin \theta \quad (7-5)$$

وهي معادلات الدائرة ضمن فضاء المتوسط parameter space المقابلة للنقطة المعلومة (x, y) ضمن فضاء الصورة.

يبدأ تحويل هوف التسلسلي من خرج خوارزمية كشف الحافة كدخل له، بعدها يتم تحديد القيم العظمى والدنيا لنصف القطر r_{min} و r_{max} للبحث عن الدائرة، ويتم إنشاء صفيقة مراكز فضاء هوف $accumulator\ array\ houghspace_i$ ، لكل نصف قطر r محتمل حيث $r_{min} \leq r \leq r_{max}$ ، بحجم مساوي لعدد البيكسلات المكّمة quantized pixels ضمن الصورة. يتم تهيئة Initialized المراكز بقيمة ابتدائية صفرية غير مهمة، بعدها يتم رسم جميع الدوائر المحتملة بقيمة نصف القطر r المعينة ضمن فضاء الموصل. بعدها تضاف الترشيحات إلى خلايا المراكز من أجل كل البيكسلات التي تقع على الدائرة المرسومة ضمن فضاء الموصل إذا كان مركز الدائرة المرسومة هو نقطة حافة في فضاء الصورة. تمثل القيمة العالية في المراكز أن البيكسل تقع ضمن نقطة تقاطع بين عدة دوائر ضمن فضاء الموصل. وتمثل نقطة الالتقاء هذه نقطة محتملة لمركز دائرة ضمن فضاء الصورة.

ونظراً لأن نصف قطر الدائرة التي يتم البحث عنها غير معلوم مسبقاً، فإنه يتم استخدام حل بسيط لتخمين الدوائر لجميع أنصاف الأقطار المحتملة، وذلك بجعل القيمة العظمى لنصف القطر مساوية للمسافة القطرية للصورة، لأنه لا يمكن لأي دائرة أن تمتلك نصف قطر بطول أكبر من طول قطر الصورة. يتم البحث عن الدوائر بجميع أنصاف الأقطار المحتملة وتحجز مراكزها لكل عمليات البحث، أي صفيقة مراكز واحدة لكل عملية بحث عن دائرة بنصف قطر معين، ثم تجمع القيم العظمى من كل خلية للمراكز وتوضع ضمن قائمة أخرى، عندها تمثل القيم العظمى ضمن القائمة الدوائر المحتملة. تظهر الخوارزمية رقم 1 الخطوات التفصيلية المتبعة ضمن هذه العملية.

5.2. خوارزمية التوازي Parallel Algorithm

تشرح الخوارزمية رقم 2 الخطوات التفصيلية المتبعة أثناء عملية مقابلة mapping الخوارزمية على وحدة معالجة الرسومات GPU. حيث أن الخطوات الأولى هي نفسها المتبعة ضمن الخوارزمية التسلسلية. يتم إنشاء شبك CUDA grid، لكل قيمة نصف قطر متوقع، بحجم كتل يساوي

$$(width - 2 * r) * (height - 2 * r) \quad (8-5)$$

حيث أن حجم كل كتلة يساوي 72 نيسب، ويقوم كل نيسب بحساب موضع البيكسل ضمن فضاء هوف، بعدها يقوم بتحميل بيكسل الصورة من صورة حواف الدائرة ثم يُحدّث المراكز المخصص لنصف القطر المعني.

خوارزمية 1 تحديد الموقع التسلسلي

Algorithm 1 SerialLocalize (*Image*)

```

1: img ← LoadImage(Image)
2: Apply edge detector to produce an edge image img.
3: Define array imgValue with value 1 for all edge pixels and 0 for others.
4: Define minimum and maximum search radius as  $r_{min}$  and  $r_{max}$ 
5: Create arrays maxValues, xPositions, yPositions and radiusValues all of size
    $r_{max} - r_{min}$ 
6:  $m \leftarrow 72$ 
7:  $i \leftarrow 0$ 
8: for  $r \leftarrow r_{min}$  to  $r_{max}$  do
9:    $n \leftarrow (width - 2 * r) * (height - 2 * r)$ 
10:  Create accumulator array houghspacei of size  $n$  and initialize with all 0.
11:  for Circle  $C_1, C_2, \dots, C_n$  each of radius value  $r$  do
12:    for Points  $P_1, P_2, \dots, P_m$  on circle  $C_j$  do
13:      if Center of the circle on which  $P_k$  lies, represents an edge point then
14:        Increment houghspacei at the edge point.
15:      end if
16:    end for
17:  end for
18:  Find  $max(houghspace_i)$  and add this value to maxValues[ $i$ ] and update
   other arrays radiusValues, xpositions and yPositions according to this value.
19:   $i \leftarrow i + 1$ 
20: end for
21:  $max(maxValues)$  corresponds to a circle in image space.
```

خوارزمية 2 تحديد الموقع المتوازي

Algorithm 2 ParallelLocalize (*Image*)

```

1: img  $\leftarrow$  LoadImage(Image)
2: Apply edge detector to produce an edge image img.
3: Define array imgData with value 1 for all edge pixels and 0 for others.
4: Define minimum and maximum search radius as  $rad_{min}$  and  $r_{max}$ 
5: Create arrays maxValues, xPositions, yPositions and radiusValues all of size
    $r_{max} - r_{min}$ 
6: i  $\leftarrow$  0
7: for  $r \leftarrow r_{min}$  to  $r_{max}$  do
8:   Create accumulator houghspacei and initialize with all 0.
9:   gridSize = (width - 2 * r) * (height - 2 * r) blocks
10:  blockSize  $\leftarrow$  72 threads
11:  Call parallel Kernel(img, r, houghspacei) for each thread
12:  Find  $max(houghspace_i)$  and add this value to maxValues[i].
13:  Record corresponding radius and center
14:  Update other arrays radiusValues , xPositions and yPositions accordingly.
15:  i  $\leftarrow$  i + 1
16: end for
17:  $max(maxValues)$  corresponds to a circle in image space .

```

خوارزمية 3 نواة التوازي

Algorithm 3 Kernel (*img*, *radius*, *houghspace*)

```

1: Calculate position of a pixel drawn on a circle in the hough space
2: Load the image pixel at the corresponding center position pos from the circle
   edge image.
3: Get the pixel value pixValue at the position pos
4: if pixValue > 0 then
5:   Update houghspace at the pixel value pix in the hough space
6: end if

```

5.3. تحليل تعقيد الزمن Time Complexity Analysis

لنفترض أن حجم الصورة المعطاة يساوي $imgsize = width * height$ بيكسل. من أجل الخوارزمية التسلسلية، ففي الحالات القصوى extreme case يمكن أن تساوي القيمة الدنيا لنصف القطر إلى بيكسل واحد والقيمة العظمى طول قطر الصورة، $d = \sqrt{width^2 + height^2}$ بيكسل. وبالتالي فإن العدد الكلي لقيم نصف القطر المحتملة يساوي إلى قطر الصورة d .

من أجل كل قيمة لنصف قطر الدائرة، يتم إنشاء عدد تقريبي $width * height$ من الدوائر ضمن فضاء هوف، فعلى فرض أن حجم الصورة كبير نسبياً مقارنةً بقيم أنصاف القطر. ونقوم بحساب 72 نقطة من أجل كل دائرة.

إن تعقيد الزمن للخوارزمية التسلسلية يتناسب مع $d * imgsize * 72$

$$T_s = O(d * imgsize) \quad (9-5)$$

من أجل الخوارزمية المتوازية يتم أيضاً التعامل مع عدد d من قيم نصف القطر. من أجل كل قيمة نصف قطر يتم إنشاء عدد من الكتل مساوٍ إلى $width * height$ ، كل منها يحتوي على 72 نيسب. وكل كتلة تقوم بتقييم دائرة وحيدة ضمن فضاء هوف.

وبما أن عدد من النياسب $width * height * 72$ تعمل على التوازي، فإن تعقيد الزمن يتناسب مع التالي

$$T_p = O(d) \quad (10-5)$$

تعمل المعادلة السابقة بكفاءة إذا كان هناك عدد كاف من النوى للتعامل مع كل نيسب بشكل مستقل. في حين يصبح تعقيد الزمن من مضاعفات T_p إذا قل عدد النوى.

الفصل السادس

التنفيذ العملي والنتائج

Implementation & Results

6.1. مدخل إلى أداة التطوير Jetson TK1

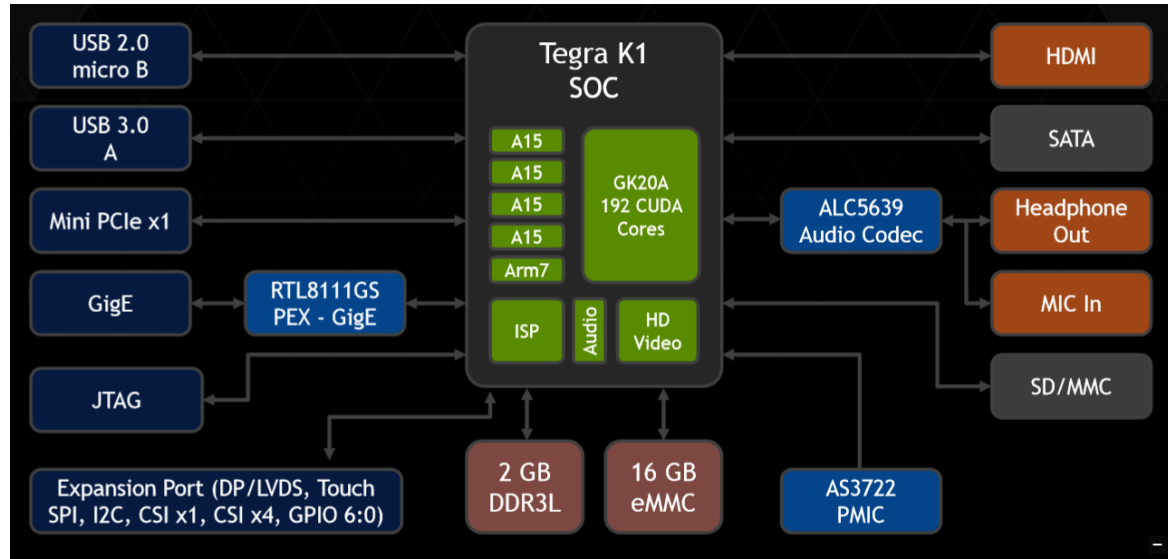
تعدّ الحوسبة المضمنة Embedded Computing المحطة التالية لوحدة معالجة الرسومات GPUs والتي يمكن أن تساعد في تسريع وتيرة الابتكار وتقدم مزايا ملحوظة في مجالات الإصدار الحاسوبي computer vision، الروبوتات robotics، السيارات ذاتية القيادة self-driving cars، معالجة الصور الرقمية digital image processing، أمن الشبكات network security، الطب، وفي العديد من المجالات الأخرى. تم تصميم أداة التطوير Jetson TK1 خصيصاً لتساهم في تطوير وتسريع التطبيقات المضمنة باستخدام وحدة معالجة الرسومات GPU-accelerated embedded applications، موفرةً بذلك إمكانية المعالجة المتوازية عالية الأداء وبكفاءة ووفر ملحوظ في استهلاك الطاقة للتطبيقات المضمنة. [39]



الشكل 1-0 أداة التطوير Jetson TK1

اعتمدت أداة التطوير Jetson TK1 في تصميمها على معالج NVIDIA Tegra K1 النقال الثوري ذو 192 نواة CUDA، مع ذاكرة وصول عشوائي RAM بسعة 2GB، وذاكرة تخزين على اللوحة EMMC بسعة 16GB بالإضافة إلى العديد من الطرفيات peripherals ومنافذ الدخل والخرج I/O ports. ولقد استُخدمت معمارية وحدة معالجة الرسومات NVIDIA Kepler GPU [40] المستخدمة في الحواسيب الخارقة وأنظمة الحوسبة العالية الأداء حول العالم نفسها في Tegra K1. وتوفّر أداة التطوير Jetson TK1 منصة NVIDIA CUDA كاملة الوظائف تتضمن حزمة تدعم اللوحة Board Support Package، OpenGL 4.4، CUDA 6.5، بالإضافة إلى أدوات التطوير NVIDIA VisionWorks toolkit. من خلال تقديم حزمة التطوير الكاملة وأدوات التتبع profiling tools، وتدعم بصورة مباشرة الكاميرات

والطرفيات الأخرى، تعدّ أداة التطوير NVIDIA Jetson TK1 منصة تطوير مثالية لتشكيل المستقبل الجديد للحوسبة المضمنة Embedded Computing.



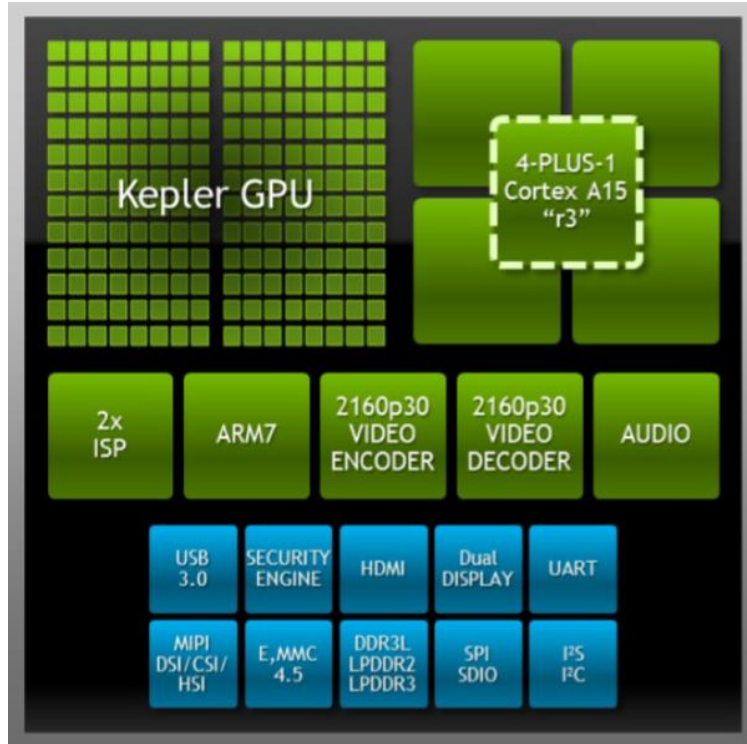
الشكل 2-0 المخطط الصندوقي لأداة التطوير Jetson TK1

6.2. معمارية NVIDIA Tegra K1

يوجد إصداران من معالجات Tegra K1 النقالة: إصدار 32-bit وإصدار 64-bit، مع توافق بالأرجل pin compatible¹. يستخدم الإصدار 32-bit وحدة المعالجة المركزية رباعية النواة نوع quad-core Cortex-A15 CPU تعمل على معدلات نبضات ساعة تصل إلى 2.3GHz، ذو قابلية Superscalar² بثلاث مراحل، وتملك خابية تعليمات بسعة 32KB L1 Instruction Cache وخابية معطيات بسعة 32KB L1 Data Cache. في حين يستخدم الإصدار 64-bit وحدة معالجة مركزية Denver CPU ثنائية النواة مخصصة تعمل على معدلات نبضات ساعة تصل إلى 2.5GHz، بسبع مراحل Superscalar، وتملك خابية تعليمات 128KB L1 Instruction Cache وخابية معطيات 64KB L1 Data Cache. نظراً لعدم نشر شركة NVIDIA لأوراق عمل إصدار نسخة 64-bit ثنائي النواة بعد، لذا تم فقط استعراض إصدار 32-bit رباعي النواة فقط في هذا القسم، وهو الذي سيستخدم للتجارب في أثناء تطبيقها على أداة التطوير Jetson TK1.

¹ يقصد بالجهاز ذو الأرجل المتوافقة pin-compatible أنه يمتلك نفس الوظائف المخصصة لنفس الأرجل المعنية

² تدل على معمارية الحاسوب والتي يمكن فيها تحميل عدة تعليمات بنفس الوقت وتنفيذها بوقت واحد، بأقصى قدر ممكن من السرعة، وبالتالي تقصير الزمن المأخوذ لتنفيذ البرنامج الكلي



الشكل 3-0 شريحة NVIDIA Tegra K1 (إصدار 32-bit)

يُظهر الشكل 3-0 المعمارية عالية المستوى لإصدار 32-bit من معالج NVIDIA Tegra K1 النقال. وفيما يلي بعض الملامح الرئيسية لمعمارية شريحة النظام Tegra K1 SoC:

- **معمارية وحدة المعالجة المركزية CPU 4-PLUS-1 Cortex A15 "r3"** والتي تقدم أداءً أعلى وكفاءة استضافة أكثر من الجيل السابق Tegra 4.
- **معمارية وحدة معالجة الرسومات Kepler GPU** تستخدم 192 نواة CUDA بقدرات متقدمة للرسومات، مع حوسبة GPU مدعومة من NVIDIA CUDA 6.5، مع تقدم خارق في كفاءة الطاقة والأداء من أجل ألعاب الجيل القادم وتطبيقات الحوسبة المسرعة باستخدام GPU.
- **نواتين لمعالجة تدفق الصورة Dual Image Stream Processing (ISP) Core** والتي تقدم 1.2 غيغا بيكسل في الثانية من قدرة المعالجة الخام والتي تدعم حساسات الكاميرا التي تصل إلى 100 ميغا بيكسل.
- **محرك إظهار متقدم Advanced Display Engine** قادر على قيادة كل من شاشة إظهار محلية بقياس 4K وشاشة عرض خارجية من خلال منفذ HDMI في آن معاً.
- **بُنيت الشريحة بتقنية TSMC 28 nm High Performance Mobile process** لتوفّر أداءً ممتازاً وكفاءة عالية في الطاقة.

بالإضافة لهذه الملامح الأساسية، يترجم المرسل/المستقبل العام غير المتزامن **Universal Asynchronous Receiver/Transmitter (UART)** بترجمة المعطيات بين الصيغ التسلسلية والمتوازية؛ بينما تمثل كل من **MIPI DSI/CSI/HSI** واجهة التخاطب التسلسلية لكل من شاشة الإظهار/الكاميرا/المنفذ التسلسلي المتزامن عالي السرعة على الترتيب لواجهة تخاطب المعالج النقل الصناعي **Mobile Industry Processor Interface (SDIO)**؛ **Secure Digital Input/Output (SDIO)** وهو نوع منافذ التخاطب مع كرت التخزين الرقمي المؤمن **Secure digital card interface**، وهو عبارة عن خط نقل معطيات متزامن يعمل بنمط **full duplex**.

6. 2. 1. وحدة المعالجة المركزية CPU

كانت الشريحة Tegra 4 أول محاولة من شركة NVIDIA لتقديم معمارية Cortex A15 رباعية النواة لسلسلة Tegra، والتي تعمل على معدلات نبضات ساعة تصل إلى 1.9GHz. استمرت شريحة Tegra K1 بنفس معمارية CPU، لكن مع قيام NVIDIA ببعض التحسينات مثل عملها على معدلات نبضات ساعة تصل إلى 2.3GHz. حيث ادعي إمكانية Tegra K1 بتسيير أداء أعلى بـ 40% عند نفس مستوى الطاقة، أو استهلاك 45% من الطاقة فقط عند نقطة أداء معينة وفق حاشية القياس SPECint2000 [41].

كما هو مبين في بداية هذه الفقرة، يستخدم الإصدار 32-bit من Tegra K1 وحدة معالجة مركزية نوع Cortex-A15 رباعية النواة، والتي تعمل بمعدلات نبضات ساعة تصل إلى 2.3GHz، Superscalar بثلاث مراحل، ولها خابية تعليمات 32KB L1 وخابية معطيات 32KB L1. تتشارك الأربع نوى Cortex A15 خابية L2 حجم 2MB ترابطية بـ 16 طريقة (كما هو مبين في الشكل 0-3). يوجد نواة وحدة معالجة مركزية إضافية من نوع A15 ذو تحسين مثالي من أجل توفير الطاقة "battery saver". وهذا الأمر يقدم مرونة من أجل الانتقال لهذه النواة الخاصة للتعامل مع المهام منخفضة الأداء وبالتالي زيادة عمر البطارية.

6.2.2. وحدة معالجة الرسومات GPU

إن معمارية وحدة معالجة الرسومات نوع Kepler GPU الموجودة ضمن الشريحة Tegra K1 هي مطابقة افتراضياً لمعمارية وحدة معالجة الرسومات Kepler GPU المستخدمة في النظم عالية الأداء. يلخص جدول 16- بعض الخصائص القياسية لكل من NVIDIA Tesla K40C و Tegra K1.

جدول 6-1 مقارنة القياسات بين Tesla K40C و Tegra K1 من NVIDIA. حيث تقيس "power consumption" متوسط الطاقة المستهلك من قبل GPU أثناء تشغيل مجموعة من ألعاب النّقال المشهورة.

	K40C	Tegra K1
Memory Clock Rate [Hz]	3GHz	950MHz
Peak single FP perf. [FLOPS]	4.29TFLOPS	364GFLOPS
Memory Bandwidth [GB/s]	288	17
Memory Size [GB]	12	1.7
# of SMX unit	15	1
# of CUDA cores	2880	192
Power consumption [W]	~100	Sub 2

6.3. تنفيذ التجربة

يخدم جزء التطبيق التتابعي المنجز على وحدة المعالجة الرسومية CPU كأساس لإنجاز الخوارزمية المتوازية ولمقارنة أدائها أمام الخوارزمية التسلسلية. كما يعدّ إنجاز CPU بسيطاً جداً وفقاً للخوارزمية المناقشة سابقاً. والفكرة الأساسية وراء التنفيذ المتوازي لخوارزمية تحديد موقع قزحية العين Iris Localization algorithm في الحقيقة هي إمكانية تحليل decomposed المعطيات إلى أجزاء صغيرة والتي يمكن تنفيذها على التوازي.

6.3.1. بيئة التنفيذ Implementation Environment

من أجل مقارنة الأداء بين الحاسوب العادي المتوسط والأجهزة غير المتجانسة النّقالة تم أولاً تطبيق الخوارزمية التسلسلية على جهاز حاسوب محمول بمواصفات متوسطة ثم إعادة ترجمتها وتنفيذها كما هي دون أي تغيير على أداة التطوير NVIDIA Jetson TK1 ومقارنة النتائج من حيث سرعة التنفيذ واستهلاك الطاقة، ثم تم تنفيذ الخوارزمية المتوازية ومقارنة مقدار التسريع على نفس أداة التطوير عند القيام باستثمار وحدة معالجة الرسومات GPU ووحدة NEON لتعمل كنظام غير متجانس مضمن نّقال Embedded mobile heterogeneous system.

يلعب العدد الكلي من نوى CUDA دوراً ملحوظاً في زمن تنفيذ الخوارزمية. حيث أن العدد الأعلى من النوى قادر على إعطاء نتائج أفضل طبعاً مع وجود بعض التقييدات. كما يلعب عرض حزمة ذاكرة التخاطب بين نوى GPU وذاكرة GPU دوراً هاماً. يظهر جدول 2-0 تفاصيل بيئة العمل والكيان الصلب المستخدم من أجل التطبيق.

جدول 2-0 تفاصيل بيئة التنفيذ

Platform	Samsung Ultrabook Series 5	NVIDIA Jetson TK1
GPU	NVIDIA GeForce 620M GPU	NVIDIA® Kepler™ Architecture
GPU Specifications	96 CUDA cores, 1GB GPU Memory	192 CUDA cores, 2GB Shared Memory
Processor	Intel Core i5 3370M, 1.70 GHz	NVIDIA 4-Plus-1™ Quad-Core ARM Cortex-A15 "r3"
System Memory	12 GB DDR3	2 GB DDR3L
Operating System	Windows 10	Ubuntu 14.04 L4T

6.3.2. النتائج Results

يظهر الجدول 3-6 زمن تنفيذ الخوارزمية من أجل كشف الحدود الدائرية (البؤبؤ والقزحية) من صور عين عشوائية مأخوذة من قاعدة معطيات CASIA V4 القياسية. حيث أن معدل التسريع Speedup المحصل من وحدة معالجة الرسوميات GPU هو 5.48x من أجل العينة المأخوذة من قاعدة معطيات CASIA [42]. وبالرغم من أن عدد نوى معالج وحدة الرسوميات GPU المستخدم هو 192 فإن مقدار التسريع Speedup لا يساوي 192 نظراً لتأثره بالعديد من العوامل مثل حجم الرماز التتابعي وتلبث الاتصال communication latency التي تم مناقشته في الفصل الأول. إلا أن هذه النتيجة تعدّ تحسناً ملحوظاً لأن قيمة التسريع الفعلي سيكون أفضل من الأنظمة الحالية المخصصة لغرض التعرف على قزحية العين، على الرغم من تأثير بيئة العمل المختارة للتنفيذ بالعديد من العوامل كتنفيذ المئات من العمليات على المعالج ضمن أي زمن معطى بالإضافة إلى العديد من العوامل الأخرى.

أظهرت التجربة نتيجة هامة أخرى تتعلق بحجم الصورة وهي زيادة كفاءة التسريع المأخوذ من أحجام مختلفة للصور بزيادة حجم الصورة. حيث تظهر النتائج أن الصور بقياسات مختلفة والمأخوذة من خلال إعادة تقييس الصور الأصلية من قاعدة معطيات CASIA، زيادة التسريع بشكل أسي. وهذا يعني أنه من أجل مهمة ذو كثافة حوسبة أقل، مثل التعامل مع صور أصغر، تبقى وحدة معالجة الرسوميات أقل استعمالاً. يعرض جدول 4-6 أزمان التنفيذ على CPU و GPU للصور بأحجام مختلفة.

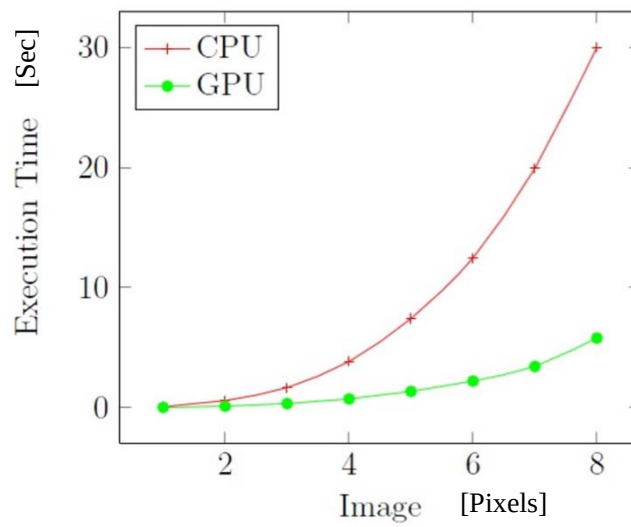
جدول 3-6 أزمان التنفيذ التسلسلية و المتوازية لخوارزمية تحديد موقع القزحية على عينة من الصور العشوائية

بقياس 320x280 مأخوذة من قاعدة معطيات CASIA

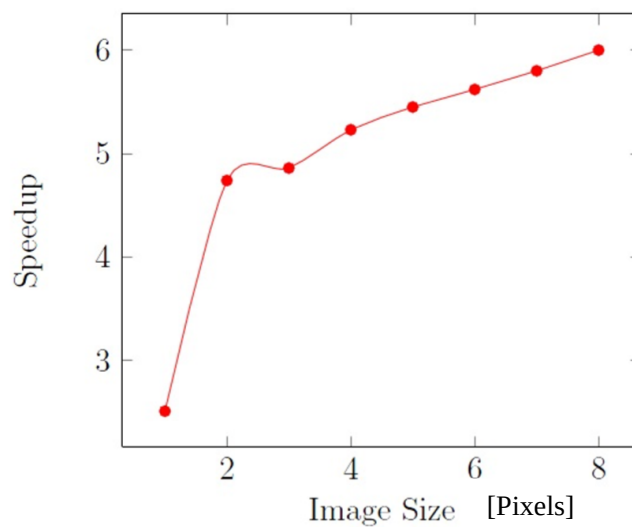
Sample No.	$t_{\text{sequential}}$ (sec)	t_{parallel} (sec)	Speedup
1	7.424	1.362	5.45
2	7.391	1.344	5.50
3	7.256	1.354	5.36
4	7.364	1.299	5.67
5	7.448	1.350	5.52
6	7.420	1.341	5.53
7	7.334	1.339	5.48
8	7.329	1.352	5.42
9	7.452	1.358	5.49
10	7.399	1.365	5.42

جدول 4-6 أزمان تنفيذ خوارزمية تحديد مكان قزحية العين على صور قزحيات عين بأحجام مختلفة

SI No.	Image Size(px)	$t_{\text{sequential}}$ (sec)	t_{parallel} (sec)	Speedup
1	64x56	0.078	0.031	2.51
2	128x112	0.593	0.125	4.74
3	192x168	1.672	0.344	4.86
4	256x224	3.844	0.735	5.23
5	320x280	7.424	1.362	5.45
6	384x336	12.469	2.219	5.62
7	448x392	19.939	3.437	5.80
8	512x448	29.996	5.801	6.00



الشكل 4-0 زمن التنفيذ مقابل حجم الصورة بالاعتماد على نتائج جدول 4-6



الشكل 4-5 اختلاف مقدار التسريع وفقاً لحجم الصورة بالاعتماد على نتائج جدول 4-6

الخلاصة والآفاق المستقبلية:

الخلاصة:

تعتمد الخوارزميات المستخدمة في عمليات معالجة الإشارة الرقمية Digital Signal Processing، كمعالجة الصور Image Processing وكشف الكائنات Objects Detection، والتي تستخدم في أبحاث وأعمال هامة على الصعيدين العلمي والطبي، على عمليات حسابية معقدة وطويلة، وغالباً ما تنفذ باستخدام أجهزة حاسوبية خارقة Super Computers ضخمة، ومتواجدة في مكان ثابت، وبحجم واستهلاك للطاقة كبيرين، وبسعر مرتفع ولا تكون متاحة لجميع الباحثين وخصوصاً في منطقة الشرق الأوسط.

تم في هذا البحث دراسة آلية استخدام وحدات المعالجة الرسومية Graphical Processing Units (GPUs) كأدوات للبرمجة عامة الغرض General Purpose ضمن الحوسبة غير المتجانسة Heterogeneous Computing، لمعرفة مدى تأثيرها على سرعة وأداء الخوارزميات والعمليات الحسابية المستخدمة في معالجة الإشارة الرقمية Digital Signal Processing. ثم تم التطرق للحوسبة غير المتجانسة النقالة Mobile Heterogeneous Computing ومقارنة النتائج مع الحواسيب غير المتجانسة التقليدية من حيث الأداء والتكلفة واستهلاك الطاقة والحجم حيث تم إثبات إمكانية استخدامها كبديل فعلي للدراسة والاختبار بدلاً من صرف المبالغ الباهظة لتوفير الحواسيب الخارقة الكلاسيكية بالإضافة إلى محدودية موارد الطاقة الكهربائية التي تحتاج إلى تشغيلها، وبالتالي يمكن العمل عليها لفترات زمنية تساوي إلى 20x من زمن تشغيل حاسوب متوسط الأداء بالإضافة إلى استهلاك طاقة منخفض يعادل 4% من استهلاك الحواسيب العادية.

كتطبيق عملي على البحث تم دراسة التقنيات المثبتة في أنظمة كشف قزحية العين Iris Recognition والتركيز على مرحلة تحديد حدود القزحية Iris Localization وكيفية تسريعها ضمن النظام وذلك بتحويل الرمز التسلسلي Serial Code المستخدم إلى رمز متوازي Parallel Code ثم مقارنة النتائج من حيث سرعة التنفيذ عندما يتم استخدام الرمز على حاسوب عادي محمول بمواصفات متوسطة وعلى أداة التطوير Jetson TK1 من شركة NVIDIA حيث وجدنا أن أداء الخوارزمية التسلسلي في حال تنفيذها على أداة التطوير النقالة يضاهي أداءها في حال تنفيذها على حاسوب عادي دون أي تغيير على الرمز بالإضافة إلى التحسين العالي في السرعة في حال موازاة الرمز وإنجازه على وحدة معالجة الرسومات GPU.

الآفاق المستقبلية:

يمكن تطوير نظام العمل كما يلي:

- تنفيذ جميع مراحل عمل نظام التعرف على قزحية العين على التوازي للحصول على تكامل عالٍ وتسريع واستقلالية تامة في عمل النظام غير المتجانس المضمّن النقل.
- استخدام التعلم العميق Deep Learning ومكاتب الشبكات العصبونية البرمجية المتوفرة على أداة التطوير Jetson TK1 لتسريع عملية البحث عن الهوية والمقارنة من قاعدة المعطيات.
- إنجاز النظام باستخدام جهاز لوحى رخيص التكلفة يستخدم الشريحة المستخدمة في أداة التطوير Jetson TK1 نفسها وباستخدام بيئة البرمجة NDK لنظام تشغيل أندرويد.
- تسريع خوارزميات أعقد مستخدمة في نظم التشفير باستخدام أداة التطوير الجديدة Jetson TX1.
- دراسة واجهة البرمجة OpenCL لتوسيع العمل ليشمل الأجهزة النقّالة الذكية بجميع الشرائح المتوفرة غير شرائح Tegra من NVidia.

:المراجع (References)

- [1] J. Clemons, "Computer architectures for mobile computer vision systems," Michigan, 2013.
- [2] V. Ferná'ndez, J. Orduna and P. Morillo, "Performance characterization of mobile phones in augmented reality marker tracking," in *Int. Conf. Computational and Mathematical Methods in Science and Engineering (CMMSE)*, 2012.
- [3] L. SE, Z. Y, F. Z, S. S, I. R and N. D, "Accelerating mobile augmented reality on a handheld platform," in *Computer design (ICCD)*, 2009.
- [4] M. B. Lopez, H. Nykanen, J. Hannukselaa, O. Silven and M. Vehvilainen, "Accelerating image recognition on mobile devices using GPGPU," in *In: Proc. of SPIE*, 2011.
- [5] K.-T. Cheng and Y.-C. Wang, "Using mobile GPU for general-purpose computing - a case study of face recognition on smartphones," in *In: Proc. IEEE Int. Symp. VLSI Design, Automation and Test (VLSI-DAT)*, 2011.
- [6] "Implementation and optimization of image processing algorithms on handheld GPU," in *IEEE International Conference on Image Processing*, 2010.
- [7] Khronos Group, "OpenGL ES 2.0 Specification," [Online]. Available: <http://www.khronos.org/opengles> .
- [8] Khronos Group, "Open Computing Language (OpenCL) Specification," [Online]. Available: <http://www.khronos.org/opencl>.
- [9] Y. Su, S. Shan, X. Chen and W. Gao, "Hierarchical Ensemble of Global and Local Classifiers for Face Recognition," *IEEE Transactions on Image Processing*, vol. no.8, no. vol.18, pp. 1885-1896, 2009.
- [10] P. J. Phillips, H. Moon, S. A. Rizvi and P. J. Rauss, "The FERET Evaluation Methodology for Race-Recognition Algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1090-1104, 06 August 2002.
- [11] T. Sumanaweera and D. Liu, "Medical image reconstruction with the FFT," *GPU Gems 2*, pp. 765-784, 2005.
- [12] Open Handset Alliance, "Android Software Development Kit 2.2," 2010. [Online]. Available: <http://www.openhandsetalliance.com/>.
- [13] J. Ström and T. Akenine-Möller, "i PACKMAN: High-quality, low-complexity texture compression for mobile phones," in *Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Los Angeles, California, 2005.

- [14] NVIDIA Corporation, "NVIDIA Jetson TK1 Embedded Development Kit," 6 2014. [Online]. Available: <http://www.nvidia.co.uk/object/jetson-tk1-embedded-dev-kit-uk.html>.
- [15] NVIDIA Corporation, "JETSON TX1 Embedded Systems Module," 2016. [Online]. Available: <http://www.nvidia.com/object/jetson-tx1-module.html>.
- [16] A. Grama, A. Gupta, G. Karypis and Vipin, *Introduction to Parallel Computing*, Second Edition ed., Pearson Education, 2007.
- [17] H. Kai, *ADVANCED COMPUTER ARCHITECTURE: Parallelism. Scalability, Programmability*, First Edition ed., Addison-Wesley Professional, 2004.
- [18] NVIDIA Corporation, "CUDA C Programming Guide," 2017. [Online]. Available: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [19] ARM, "NEON Programmer's Guide," 2013. [Online]. Available: <https://developer.arm.com/docs/den0018/latest/neontm-version-10-programmers-guide>.
- [20] L. Flom and A. Safir, "IRIS RECOGNITION SYSTEM". USA Patent 4,641,349, 3 Feb 1987.
- [21] J. G. Daugman, "High confidence visual recognition of persons by a test of statistical independence," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1148-1161, 1993.
- [22] J. G. Daugman, "The importance of being random: statistical principles of iris recognition," *Pattern Recognition*, no. Vol 36, pp. 279-291, 2003.
- [23] R. P. Wildes, J. C. Asmuth, G. L. Green, S. C. Hsu, R. J. Kolczynski, J. R. Matey and S. E. McBride, "A machine-vision system for iris recognition," *Machine Vision and Applications*, pp. 1-8, January 1996.
- [24] J. G. Daugman, "How iris recognition works," *IEEE Transactions on Circuits and Systems for Video Technology*, no. Vol 14, pp. 21-30, 2004.
- [25] J. Huang, Y. Wang, T. Tan and J. Cui, "A new iris segmentation method for recognition," in *17th International Conference on Pattern Recognition*, 2004.
- [26] A. Bachoo and J. Tapamo, "A segmentation method to improve iris based person identification," in *proceedings of 7th Africon Conference*, 2004.
- [27] R. P. Wildes, "Iris recognition: an emerging biometric technology," *Proceedings of the IEEE*, vol. 9, no. Vol. 85, pp. 1348-1363, 1997.

- [28] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Graphics and Image processing, Communications of the ACM*, no. Vol 15, pp. 11-15, January 1972.
- [29] H. P.V.C, "Method and Means for Recognizing Complex Patterns". USA Patent 3069654, 1962.
- [30] J. Illingworth and J. Kittler, "A survey of the Hough transform," *Computer Vision, Graphics and Image Processing*, no. Vol. 44, pp. 87-116, 1988.
- [31] N. Ritter, R. Owens, J. Cooper and P. V. Saarloos, "Location of the pupil-iris border in slit-lamp images of the cornea," in *International Conference on Image Analysis and Processing*, 1999.
- [32] K. W. Bowyer, K. Hollingsworth and P. J. Flynn, "Image understanding for iris biometrics: A survey," *Computer Vision and Image Understanding*, vol. 2, no. Vol. 110, pp. 281-307, 2008.
- [33] Y.-P. Huang, S.-W. Luo and E.-Y. Chen, "An efficient iris recognition system," in *International Conference on Machine Learning and Cybernetics*, 2002.
- [34] W. K. Kong and D. Zhang, "Accurate iris segmentation based on novel reflection and eyelash detection model," in *International Symposium on Intelligent Multimedia, Video and Speech Processing*, Hong Kong, 2001.
- [35] F. Z. Sakr, M. Taher and A. M. Wahba, "High Performance Iris Recognition System On GPU," 2011.
- [36] F. Z. Sakr, M. Taher, A. M. Ei-Bilaly and A. M. Wahba, "Accelerating Iris Recognition algorithms on GPUs," in *IEEE CIBEC*, Cairo, 2012.
- [37] R. N. Rakvie, U. J. Bradley, R. P. Broussard, R. W. Ives and N. Steiner, "Parallelizing Iris Recognition," *IEEE transactions on Information Forensics and Security*, vol. 4, no. 4, December 2009.
- [38] R. N. Rakvic, B. J. Ullis, R. P. Broussard and R. W. Ives, "Iris template generation with parallel logic," in *in Proceeding 42nd Annual Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, 2008.
- [39] NVIDIA, "Jetson TK1 Development Kit User Guide," November 2014. [Online]. Available: http://developer.download.nvidia.com/embedded/jetson/TK1/docs/2_GetStart/Jetson_TK1_User_Guide.pdf.
- [40] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012. [Online]. Available: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-ArchitectureWhitepaper.pdf>.

- [41] "CINT2000 (Integer Component of SPEC CPU2000)," 2003. [Online]. Available: <http://www.spec.org/cpu2000/CINT2000>.
- [42] Chinese Academy of Sciences Institute of Automation, " Database of 756 Greyscale Eye Images," [Online]. Available: <http://www.sinobiometrics.com>.
- [43] J. G. Daugman, "Biometric personal identification system based on iris analysis". USA Patent 5,291,560, 1 Mar 1994.
- [44] D. G. Lowe, "Distinctive image features from scale invariant keypoints," *International Journal of Computer Vision*, no. Vol 60, pp. 91-110, 2004.
- [45] Q.-C. Tian, Q. Pan and Y.-M. Cheng, "Fast algorithm and application of Hough transform in iris segmentation," in *International Conference on Machine Learning and Cybernetics*, 2004.
- [46] Y. Liu, S. Yuan, X. Zhu and Q. Cui, "A practical iris acquisition system and a fast edges locating algorithm in iris recognition," in *20th IEEE Conference on Instrumentation and Measurement Technology*, 2003.

الملاحق

Appendences

hough2d.h

```
#include <iostream>
#include <math.h>

#define GRAYLEVELS 256

// CUDA functions
extern "C" void Hough2D_CUDA(float* img, int width, int height, int
radMin, int radMax, int* posX, int* posY, int* maxVal, int* resRad);
extern "C" void imadjustCUDA(unsigned char *inImg, unsigned char
*outImg, int width, int height, float lowPerc, float highPerc);
extern "C" void adjustGammaCUDA(unsigned char *inImg, unsigned char
*outImg, int width, int height, float gamma);
```

hough2d.cu

```
#include "hough2d.h"
#include "main.h"
#include <sm_32_atomic_functions.h>

#define BLOCK_SIZE 512
#define BLOCK_SIZE_HOUGH 360
#define STEP_SIZE 5
#define NUMBER_OF_STEPS 360/STEP_SIZE

// Circ mask kernel storage
__constant__ int maskKernelX[NUMBER_OF_STEPS];
__constant__ int maskKernelY[NUMBER_OF_STEPS];

// Function to set precalculated relative coordinates for circle
boundary coordinates
extern "C" void setMaskKernel(int *maskX, int *maskY)
{
    cudaMemcpyToSymbol(maskKernelX, maskX, NUMBER_OF_STEPS*sizeof(int));
    cudaMemcpyToSymbol(maskKernelY, maskY, NUMBER_OF_STEPS*sizeof(int));
}

// Kernel to set all pixel values to specified value
__global__ void setAllValuesKernel(int* houghSpace, int height, int
width, float value)
{
    int const index = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    if (index < height*width) {
        houghSpace[index] = value;
    }
    __syncthreads();
}

extern "C" void setAllValuesToCUDA(int* houghSpace, int height, int
width, float value)
{
    //cout << "Setting all values to " << value << "..." << endl;
    dim3 dimGrid = (ceil((float)width*height/(float)BLOCK_SIZE));
    dim3 dimBlock = (BLOCK_SIZE);
    setAllValuesKernel<<<dimGrid, dimBlock>>>(houghSpace, height, width,
value);
    cudaThreadSynchronize();
}
```

```

// Kernel to perform circular Hough transform
__global__ void houghTransformKernel(int* cudaHough, float* img, int
height, int width, int radius)
{
    if (threadIdx.x < BLOCK_SIZE_HOUGH) {
        // Arrays to hold coordinates for circle pixels
        __shared__ float circVals[NUMBER_OF_STEPS];

        // There are 10 hough pixels calculated in each block
        int whichPixel = (int)threadIdx.x / NUMBER_OF_STEPS;
        // Calculate position for pixel in hough space
        int cpixIDy = (int)((float)(blockIdx.x*STEP_SIZE+whichPixel))
/ (float)(width-(radius*2)) + radius;
        int cpixIDx = (blockIdx.x*10+whichPixel) % (width-(radius*2)) +
radius;

        // Load image pixel from circle edge
        int xVal = cpixIDx + maskKernelX[threadIdx.x % NUMBER_OF_STEPS];
        int yVal = cpixIDy + maskKernelY[threadIdx.x % NUMBER_OF_STEPS];
        // Get the pixel value from the image
        float pixVal = img[yVal*width + xVal];
        //float pixVal = img[cpixIDy*width + cpixIDx]; // ## TO DELETE,
INCORRECT

        //int houghVal = cudaHough[cpixIDy*width + cpixIDx];
        __syncthreads();

        if (pixVal > 0) {
            atomicAdd(cudaHough + cpixIDy*width + cpixIDx, 1);
        }
        __syncthreads();
    }
}

// Calls the Hough transform kernel
extern "C" void performHoughTransformCUDA(int* cudaHough, float* img,
int height, int width, int radius)
{
    // Define grid and block dimensions
    dim3 dimGrid = ( ceil((float)(width-(2*radius)) * (height-
(2*radius)) / (float)STEP_SIZE) );
    dim3 dimBlock = (BLOCK_SIZE_HOUGH);

    // Perform Hough transform and sync threads to get the final result
    houghTransformKernel<<<dimGrid, dimBlock>>>(cudaHough, img, height,
width, radius);
    cudaThreadSynchronize();
}

// Analyse the defined image area for circles using Hough Transform
extern "C" void Hough2D_CUDA(float* img, int width, int height, int
radMin, int radMax, int* posX, int* posY, int* maxVal, int* resRad)
{
    int* houghSpace;
    houghSpace = (int*)malloc(width*height*sizeof(int));

    // Arrays for results
    int *posxArray, *posyArray, *maxValArray, *radArray;

```

```

// Allocate correct memory for arrays
posxArray = (int*)malloc((radMax-radMin)*sizeof(int));
posyArray = (int*)malloc((radMax-radMin)*sizeof(int));
maxValArray = (int*)malloc((radMax-radMin)*sizeof(int));
radArray = (int*)malloc((radMax-radMin)*sizeof(int));

// Allocate memory for CUDA images and matrices
float *cudaImg;
int *cudaHough;

cudaMalloc((void **)&cudaImg, width*height*sizeof(float));
cudaMalloc((void **)&cudaHough, width*height*sizeof(int));
// Copy image from host to device
cudaMemcpy(cudaImg, img, width*height*sizeof(float),
cudaMemcpyHostToDevice);

int ctrArr = 0, radius;
for (int i=radMin; i < radMax; i++) {
    // Set all elements to zero
    setAllValuesToCUDA(cudaHough, height, width, 0);

    // Precalculate relX and relY
    radius = i;
    int ctr = 0;
    int* relX, *relY;
    relX = (int*)malloc(NUMBER_OF_STEPS*sizeof(int));
    relY = (int*)malloc(NUMBER_OF_STEPS*sizeof(int));
    for (int theta=0; theta < 360; theta+=STEP_SIZE) {
        // Calculate x and y coordinates
        float angle = (theta*PI) / 180;
        relX[ctr] = (int)(-radius*cos(angle));
        relY[ctr] = (int)(-radius*sin(angle));
        ctr++;
    }

    // Set mask coordinates for circle
    setMaskKernel(relX, relY);

    // Free memory
    free(relX);
    free(relY);

    //performHoughTransformCUDA(cudaHough, cudaImg, height, width,
radius, relX, relY, angleNum);
    performHoughTransformCUDA(cudaHough, cudaImg, height, width,
radius);

    // Copy matrix from device to host
    cudaMemcpy(houghSpace, cudaHough, width*height*sizeof(float),
cudaMemcpyDeviceToHost);

    // Find max value in the houghSpace
    *maxVal = 0;
    int index;
    int tempPosX, tempPosY, tempMaxVal = 0;

    for (int y=0; y < height; y++) {
        for (int x=0; x < width; x++) {
            //index = radius*width*height + y*width + x;
            index = y*width + x;

```

```

        if (tempMaxVal < houghSpace[index]) {
            tempMaxVal = houghSpace[index];
            tempPosX = x;
            tempPosY = y;
        }
    }

    // Write results to arrays
    posXArray[ctrArr] = tempPosX;
    posYArray[ctrArr] = tempPosY;
    maxValArray[ctrArr] = tempMaxVal;
    radArray[ctrArr] = i;

    cout << "Current (radius: " << i << ") MaxVal: " <<
maxValArray[ctrArr] << " ctr: " << ctrArr << endl;

    ctrArr++;
} // end for

// Find the maximum value from arrays
*maxVal = 0;
for (int j=0; j < (radMax-radMin); j++) {
    cout << "MaxValArray: " << maxValArray[j] << " " << posXArray[j]
<< " " << posYArray[j] << " " << radArray[j] << endl;
    if (*maxVal < maxValArray[j]) {
        *maxVal = maxValArray[j];
        *posX = posXArray[j];
        *posY = posYArray[j];
        *resRad = radArray[j];
    }
}

// Free cuda memory
cudaFree(cudaImg);
cudaFree(cudaHough);

// Free array memory
free(posxArray);
free(posyArray);
free(maxValArray);
free(radArray);
}

// #####
// ##### IMADJUST #####
// #####

__global__ void AdjustImageIntensityKernel(float *imgOut, float *imgIn,
int width, int height, float lowin, float lowout, float scale)
{
    __shared__ float bufData[BLOCK_SIZE];

    // Get the index of pixel
    const int index = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    // Load data to shared variable
    bufData[threadIdx.x] = imgIn[index];

```

```

    // Check that it's not out of bounds
    if (index < (height*width)) {

        // Find the according multiplier
        float tempLevel = ( bufData[threadIdx.x] - lowin)*scale +
lowout;

        // Check that it's within required range
        if (tempLevel < 0) {
            bufData[threadIdx.x] = 0;
        }
        else if (tempLevel > 1) {
            bufData[threadIdx.x] = 1;
        }
        else {
            bufData[threadIdx.x] = tempLevel;
        }

        // Write data back
        imgOut[index] = bufData[threadIdx.x];
    }

    // Synchronise threads to have the whole image fully processed for
output
    __syncthreads();
}

// Resize the image
__global__ void ImageScalingKernel(float *imgOut, float *imgIn, int
width, int height)
{
    __shared__ float inData[BLOCK_SIZE];
    // Get the index of pixel
    const int index = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    // Load data to shared variable
    inData[threadIdx.x] = imgIn[index];

    if ( index < (width*height) ) {
        imgOut[index] = inData[threadIdx.x] / (float)255;
    }

    __syncthreads();
}

void ImageHistogramCUDA(float *pSrc, int width, int height, int *
imghist)
{
    const int GrayThres = 256;
    for (int i=0; i< GrayThres; i++) imghist[i] = 0;
    for (int i=0; i< width*height; i++) {
        int level = (int) (pSrc[i]*255);
        imghist[level]++;
    }
}

```

```

// Stretch limit
void ImageStretchLimitCUDA(float *pSrc, int width, int height,
                           float tol_low, float tol_high, float *low, float
                           *high )
{
    const int GrayThres = 256;
    int imghist[256];

    double cdf[GrayThres], sum;
    int i;
    bool bLowFound=false, bHighFound=false;

    //histogram
    ImageHistogramCUDA(pSrc,width,height,imghist);

    // the below segment can be implemented on CPU only;
    //*****
    // cdf
    cdf[0]=imghist[0];
    for (i=1;i<GrayThres;i++)
        cdf[i] = cdf[i-1] + imghist[i];
    sum = cdf[GrayThres-1];
    for (i=0;i<GrayThres;i++)
        cdf[i] /= sum;

    // find low and high
    for (i=0;i<GrayThres;i++)
    {
        if (cdf[i]>=tol_low && (bLowFound == false))
        {
            *low = (float)(i);
            bLowFound = true;
        }

        if (cdf[i]>=tol_high && (bHighFound == false))
        {
            *high = (float)(i);
            bHighFound = true;
        }
    }
    // convert to range [0 1]
    *low /= (GrayThres-1);
    *high /= (GrayThres-1);
}

// Adjusts image intensity depending on the current gray levels of the
// image (histogram stretching)
extern "C" void imadjustCUDA(unsigned char *inImg, unsigned char
*outImg, int width, int height, float lowPerc, float highPerc)
{
    const int grayLevels = 256;
    float lowin, highin;
    float *tempBuffer = new float[width*height];
    float *imgInput, *imgBuffer, *imgOutput;
    clock_t init, final_gpu;

    // Convert input image to float
    for (int i=0; i < (width*height); i++) {
        tempBuffer[i] = (float) inImg[i];
    }

```

```

// ### ALLOCATE CUDA ARRAYS ###
cudaMalloc((void **)&imgInput, width * height * sizeof(float));
cudaMalloc((void **)&imgBuffer, width * height * sizeof(float));
cudaMalloc((void **)&imgOutput, width * height * sizeof(float));

// ### COPY TO CUDA MEMORY ###
cudaMemcpy(imgInput, tempBuffer, width * height * sizeof(float),
cudaMemcpyHostToDevice);

// Get number of blocks
int gridSize = ceil( (float)(height*width) / (float)BLOCK_SIZE );

// Assign sizes
dim3 blocks( gridSize );
dim3 threads( BLOCK_SIZE );

// Image scaling Kernel
ImageScalingKernel<<<blocks, threads>>>(imgBuffer, imgInput, width,
height);
cudaThreadSynchronize();

// Copy image buffer back to host memory (for ImageStretchLimit
function)
cudaMemcpy(tempBuffer, imgBuffer, width * height * sizeof(float),
cudaMemcpyDeviceToHost);

// find out the 1% pixel intensity value and set it to "low"
// find out the 99% pixel intensiy valeu and set it to "high"
//ImageStretchLimitCUDA(tempBuffer, width, height,
0.01f,0.99f,&lowin,&highin);
ImageStretchLimitCUDA(tempBuffer, width, height,
lowPerc,highPerc,&lowin,&highin);

// Adjust image intensity
float lowout = 0, highout = 1;
float range = highin-lowin;
float rangeout = highout-lowout;
float scale = rangeout/range;

printf("Adjusting image intensities on GPU (CUDA)...\n");
// Start timer
init = clock();
// Call the adjust image intensity kernel
AdjustImageIntensityKernel<<<blocks, threads>>>(imgOutput,
imgBuffer, width, height, lowin, lowout, scale);
cudaThreadSynchronize();

// Copy the result back
cudaMemcpy(tempBuffer, imgOutput, width * height * sizeof(float),
cudaMemcpyDeviceToHost);

// Take time
final_gpu=clock()-init;
printf("Time taken for imadjust on GPU (CUDA): %f sec\n",
(double)final_gpu / ((double)CLOCKS_PER_SEC));

// convert it back to unsigned char
for (int i =0; i< width*height; i++) {
    outImg[i] = (unsigned char) (tempBuffer[i]*255);
}

```

```

    // Free memory
    cudaFree(imgInput);
    cudaFree(imgBuffer);
    cudaFree(imgOutput);
}

// #####
// ##### ADJUSTGAMMA #####
// #####

__global__ void AdjustGammaKernel(float *imgOut, float *imgIn, int
width, int height, float gamma, float minVal, float maxVal)
{
    __shared__ float bufData[BLOCK_SIZE];

    // Get the index of pixel
    const int index = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    // Load data to shared variable
    bufData[threadIdx.x] = imgIn[index];

    // Check that it's not out of bounds
    if (index < (height*width)) {

        // Find the according multiplier
        float tempLevel = ( bufData[threadIdx.x] - minVal) / maxVal;

        tempLevel = powf(tempLevel, (double)1/gamma);

        // Check that it's within required range
        if (tempLevel < 0) {
            bufData[threadIdx.x] = 0;
        }
        else if (tempLevel > 1) {
            bufData[threadIdx.x] = 1;
        }
        else {
            bufData[threadIdx.x] = tempLevel;
        }

        // Write data back
        imgOut[index] = bufData[threadIdx.x];
    }

    // Synchronise threads to have the whole image fully processed for
    output
    __syncthreads();
}

extern "C" void adjustGammaCUDA(unsigned char *inImg, unsigned char
*outImg, int width, int height, float gamma)
{
    const int grayLevels = 256;
    float lowin, highin;
    float *tempBuffer = new float[width*height];
    float *imgInput, *imgOutput;
    clock_t init, final_gpu;

    float minVal = 1000, maxVal = 0;

```

```

// Convert input image to float
for (int i=0; i < (width*height); i++) {
    tempBuffer[i] = (float) inImg[i];

    // Calculate min and max values in the image ## CAN BE ADDED TO
    DO ON CUDA LATER ON ##
    if (minVal > tempBuffer[i]) {
        minVal = tempBuffer[i];
    }
    if (maxVal < tempBuffer[i]) {
        maxVal = tempBuffer[i];
    }
}

// ### ALLOCATE CUDA ARRAYS ###
cudaMalloc((void **)&imgInput, width * height * sizeof(float));
cudaMalloc((void **)&imgOutput, width * height * sizeof(float));

// ### COPY TO CUDA MEMORY ###
cudaMemcpy(imgInput, tempBuffer, width * height * sizeof(float),
cudaMemcpyHostToDevice);

// Get number of blocks
int gridSize = ceil( (float)(height*width) / (float)BLOCK_SIZE );

// Assign sizes
dim3 blocks( gridSize );
dim3 threads( BLOCK_SIZE );

printf("Adjusting gamma on GPU (CUDA)...\n");
// Start timer
init = clock();
// Image scaling Kernel
AdjustGammaKernel<<<blocks, threads>>>(imgOutput, imgInput, width,
height, gamma, minVal, maxVal);
cudaThreadSynchronize();

// Copy the result back
cudaMemcpy(tempBuffer, imgOutput, width * height * sizeof(float),
cudaMemcpyDeviceToHost);

// Take time
final_gpu=clock()-init;
printf("Time taken for gamma adjustment on GPU (CUDA): %f sec\n",
(double)final_gpu / ((double)CLOCKS_PER_SEC));

// convert it back to unsigned char
for (int i =0; i< width*height; i++) {
    outImg[i] = (unsigned char) (tempBuffer[i]*255);
}

// Free memory
cudaFree(imgInput);
cudaFree(imgOutput);
}

```

main.h

```
#include <iostream>
#include <math.h>
#include <time.h>

// File reading
#include <fstream>

// OpenCV headers
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>

// CUDA headers
// #include <cutil_inline.h>

#define CONST_RADIUS 10
#define PI 3.141592653589793
```

main.c

```
#include "main.h"
#include "hough2d.h"
// #include "imadjustCuda.h"

using namespace cv;

// Define scale, best accuracy/speed ratio achieved using 0.6 for test images
#define SCALE 0.6

// Function prototypes CPU
int Hough2D_CPU(int* img, int width, int height, int radius, int* posX,
int* posY, int* maxVal);
int findCircleHoughCPU(IplImage *edgeImg, int radMin, int radMax, int*
resX, int *resY, int *resRad, int *resVal);

// Function prototypes CUDA
int findCircleHoughCUDA(IplImage *edgeImg, int radMin, int radMax, int*
resX, int *resY, int *resRad, int *resVal);
int imadjustInput(IplImage *cvImg, float lowPerc, float highPerc);

int main()
{
    // For timer
    clock_t init, final;

    // Open the file.
    IplImage *img =
cvLoadImage("../images/test_3.bmp", CV_LOAD_IMAGE_GRAYSCALE);
    if (!img) {
        cout << "Error: Couldn't open the image file." << endl;
        return 1;
    }
}
```

```

//IplImage *imgResult = cvLoadImage("eye_edge2.png");
IplImage *imgResult = cvLoadImage("../images/test_3.bmp");
if (!img) {
    cout << "Error: Couldn't open the image file." << endl;
    return 1;
}

// Blur the image
cout << "Gaussian blur on CPU..." << endl;
// Start timer
init=clock();
cvSmooth( img, img, CV_GAUSSIAN, 3, 3 );
// Stop timer
final=clock()-init;
cout << "Time taken to perform Gaussian blurring CPU:" <<
(double)final / ((double)CLOCKS_PER_SEC) << "s" << endl << endl;

// Resize the image to SCALE % of the size
IplImage *img_small =
cvCreateImage( cvSize((int)(img->width*SCALE) ,
(int)(img->height*SCALE) ),img->depth, img->nChannels );
cvResize(img, img_small);

// Adjust the intensity of the image
imadjustInput(img_small, 0.01f, 0.99f);

// Create new image for storing edge image
IplImage *edgeImg = cvCreateImage( cvGetSize(img_small),
IPL_DEPTH_8U, 1 );

// Canny edge detection
cout << "Canny edge detection on CPU..." << endl;
// Start timer
init=clock();
cvCanny( img_small, edgeImg, 230, 250, 3);
// Stop timer
final=clock()-init;
cout << "Time taken to perform Canny on CPU:" << (double)final /
((double)CLOCKS_PER_SEC) << "s" << endl << endl;

// Dilate image to thicken edges
//cvDilate( edgeImg, edgeImg, 0, 1);

// Display edge image
cvNamedWindow("Edge Image:", CV_WINDOW_AUTOSIZE);
cvShowImage("Edge Image:", edgeImg);

// Wait for the user to press a key in the GUI window.
//cvWaitKey(0);

int posX, posY, maxVal, radius;

// Define minimum and maximum radius
int radMin = (int)30 * SCALE;
int radMax = (int)50 * SCALE;

cout << "Finding pupil in the image using Hough transform..." <<
endl;
// Start timer

```

```

    init=clock();
    // Find circles
    //findCircleHoughCPU(edgeImg, radMin, radMax, &posX, &posY, &radius,
    &maxVal);
    // Stop timer
    final=clock()-init;
    cout << "Time taken to perform Hough on CPU:" << (double)final /
    ((double)CLOCKS_PER_SEC) << "s" << endl << endl;
    // Print out results
    //cout << "CPU results: MaxVal(" << maxVal << ") was found a
    position (" << posX << "," << posY << ") with radius: " << radius <<
    endl;

    // Start timer
    init=clock();
    cout << "Finding pupil in the image using Hough transform on
    CUDA..." << endl;

    findCircleHoughCUDA(edgeImg, radMin, radMax, &posX, &posY, &radius,
    &maxVal);
    // Stop timer
    final=clock()-init;
    cout << "Time taken to perform Hough on CUDA:" << (double)final /
    ((double)CLOCKS_PER_SEC) << "s" << endl << endl;

    // Rescale values to match full size image
    posX = (int)posX/SCALE;
    posY = (int)posY/SCALE;
    radius = (int)radius/SCALE;

    // Print out results
    cout << "CUDA results on pupil: MaxVal(" << maxVal << ") was found a
    position (" << posX << "," << posY << ") with radius: " << radius <<
    endl;

    // Draw the circle
    cvCircle(imgResult, cvPoint(posX, posY), radius, CV_RGB(0,0,255),
    4);
    cvShowImage("Orig Image:", imgResult);

    cvWaitKey(0);

    // #####
    // ### FINDING IRIS ###
    // #####

    // Calculate area where iris is located
    int tl_x = posX - 4*radius;
    int tl_y = posY - 4*radius;
    int dim = 8*radius;

    // Redefine radMin and radMax
    radMin = (int)radius*2*SCALE;
    radMax = (int)radius*3.5*SCALE;

    int posX_i, posY_i, maxVal_i, radius_i;

    // Crop the image
    cvSetImageROI(img, cvRect(tl_x, tl_y, dim, dim));

```

```

    IplImage *img_cropped = cvCreateImage(cvGetSize(img), img->depth,
img->nChannels);
    cvCopy(img, img_cropped, NULL);
    // Reset ROI
    cvResetImageROI( img );
    IplImage *img_cropped_s =
cvCreateImage( cvSize((int)(img_cropped->width*SCALE) ,
(int)(img_cropped->height*SCALE) ),img_cropped->depth,
img_cropped->nChannels );
    cvResize(img_cropped, img_cropped_s);

    // Adjust the intensity of the image
    imadjustInput(img_cropped_s, 0.3f, 0.5f);

    // Create new image for storing edge image
    IplImage *edgeImgCropped = cvCreateImage( cvGetSize(img_cropped_s),
IPL_DEPTH_8U, 1 );

    // Canny edge detection
    cout << "Canny edge detection on CPU..." << endl;
    // Start timer
    init=clock();
    cvCanny( img_cropped_s, edgeImgCropped, 210, 250, 3);
    // Stop timer
    final=clock()-init;
    cout << "Time taken to perform Canny on CPU:" << (double)final /
((double)CLOCKS_PER_SEC) << "s" << endl << endl;

    // Dilate image to thicken edges
    cvDilate( edgeImgCropped, edgeImgCropped, 0, 1);

    // Display edge image
    cvNamedWindow("Edge Image:", CV_WINDOW_AUTOSIZE);
    cvShowImage("Edge Image:", edgeImgCropped);

    // Start timer
    init=clock();
    cout << "Finding iris in the image using Hough transform on CUDA..."
<< endl;

    findCircleHoughCUDA(edgeImgCropped, radMin, radMax, &posX_i,
&posY_i, &radius_i, &maxVal_i);
    // Stop timer
    final=clock()-init;
    cout << "Time taken to perform Hough on CUDA:" << (double)final /
((double)CLOCKS_PER_SEC) << "s" << endl << endl;

    // Rescale values to match full size image
    posX_i = (int)posX_i/SCALE;
    posY_i = (int)posY_i/SCALE;
    radius_i = (int)radius_i/SCALE;

    // Print out results
    cout << "CUDA results on iris: MaxVal(" << maxVal_i << ") was found
a position (" << posX_i << ", " << posY_i << ") with radius: " <<
radius_i << endl;

    // ### TEMP ### Display the image
    cvCircle(img_cropped, cvPoint(posX_i, posY_i), radius_i,
CV_RGB(0,0,255), 4);

```

```

cvNamedWindow("ROI Image:", CV_WINDOW_AUTOSIZE);
cvShowImage("ROI Image:", img_cropped);

cvWaitKey(0);

// Free the resources.
cvDestroyWindow("Orig Image:");
//cvReleaseImage(&img);

cvDestroyWindow("Edge Image:");
cvReleaseImage(&edgeImg);

return 0;
}

int findCircleHoughCPU(IplImage *edgeImg, int radMin, int radMax, int*
resX, int *resY, int *resRad, int *resVal)
{
    // Get dimensions of the image
    int width = edgeImg->width;
    int height = edgeImg->height;

    cout << "Width: " << width << ", Height: " << height << endl;
    cout << "Rad min: " << radMin << ", Rad max: " << radMax << endl;

    // Convert IplImage to an array
    uchar *data;
    cvGetRawData(edgeImg, (uchar**)&data);
    int* imgData;
    imgData = (int*)malloc(width*height*sizeof(int));
    for (int i=0; i < width*height; i++) {
        if (data[i] > 0) {
            imgData[i] = 1;
        }
        else {
            imgData[i] = 0;
        }
    }

    // Image array for CUDA
    float* imgDataCUDA;
    imgDataCUDA = (float*)malloc(width*height*sizeof(float));
    for (int i=0; i < width*height; i++) {
        if (data[i] > 0) {
            imgDataCUDA[i] = 1;
        }
        else {
            imgDataCUDA[i] = 0;
        }
    }

    // Variables for Hough
    int radius;
    int posX, posY, maxVal;
    // Arrays for results
    int *posxArray, *posyArray, *maxValArray, *radArray;
    // Allocate correct memory for arrays
    posxArray = (int*)malloc((radMax-radMin)*sizeof(int));
    posyArray = (int*)malloc((radMax-radMin)*sizeof(int));
    maxValArray = (int*)malloc((radMax-radMin)*sizeof(int));
    radArray = (int*)malloc((radMax-radMin)*sizeof(int));

```

```

    // Perform task for all the radius
    int index = 0;
    for (int r=radMin; r <= radMax; r++) {
        // Perform Hough transform
        radius = r;
        Hough2D_CPU(imgData, width, height, radius, &posX, &posY,
&maxVal);
        // Write results to arrays
        posXArray[index] = posX;
        posYArray[index] = posY;
        maxValArray[index] = maxVal;
        radArray[index] = radius;

        index++;
    }

    // Find the maximum value from arrays
    maxVal = 0;
    for (int i=0; i < (radMax-radMin); i++) {
        if (maxVal < maxValArray[i]) {
            maxVal = maxValArray[i];
            *resX = posXArray[i];
            *resY = posYArray[i];
            *resRad = radArray[i];
            *resVal = maxVal;
        }
    }

    cout << "maxVal: " << maxVal << endl;

    return 0;
}

int Hough2D_CPU(int* img, int width, int height, int radius, int* posX,
int* posY, int* maxVal)
{
    int* houghSpace;
    houghSpace = (int*)malloc(width*height*sizeof(int));
    // Set all elements to zero
    for (int i=0; i < width*height; i++) {
        houghSpace[i] = 0;
    }

    // Perform hough transform
    for (int x=radius; x < width-radius; x++) {
        for (int y=radius; y < height-radius; y++) {
            // 'Draw' a circle
            for (int theta=0; theta < 360; theta+=5) {
                // Calculate x and y coordinates
                float angle = (theta*PI) / 180;
                int tempX = (int)(x - radius*cos(angle));
                int tempY = (int)(y - radius*sin(angle));
                // If edge is detected, add vote
                if (img[tempY*width+tempX] == 1) {
                    houghSpace[y*width+x] = houghSpace[y*width+x] + 1;
                }
            }
        }
    }
}

```

```

// Find max value in the houghSpace
*maxVal = 0;
int index;

for (int y=0; y < height; y++) {
    for (int x=0; x < width; x++) {
        //index = radius*width*height + y*width + x;
        index = y*width + x;
        if (*maxVal < houghSpace[index]) {
            *maxVal = houghSpace[index];
            *posX = x;
            *posY = y;
        }
    }
}

return 0;
}

int findCircleHoughCUDA(IplImage *edgeImg, int radMin, int radMax, int*
resX, int *resY, int *resRad, int *resVal)
{
    // Get dimensions of the image
    int width = edgeImg->width;
    int height = edgeImg->height;

    // ### Convert IplImage to an array ###
    unsigned char *inImg, *temp2;
    inImg = new unsigned
char[edgeImg->width*edgeImg->height*edgeImg->nChannels];
    temp2 = inImg;

    // pointer to imageData
    unsigned char *temp1 = (unsigned char*) edgeImg->imageData;
    // copy imagedata to buffer row by row
    for(int i=0; i<edgeImg->height; i++)
    {
        // memory copy
        memcpy(temp2, temp1, edgeImg->width*edgeImg->nChannels);
        // imageData jump to next line
        temp1 = temp1 + edgeImg->widthStep;
        // buffer jump to next line
        temp2 = temp2+ edgeImg->width*edgeImg->nChannels;
    }

    // Saving the matrix to file
    ofstream fout;
    fout.open("houghinput.dat");

    for (int y=0; y < height; y++) {
        for (int x=0; x < width; x++) {
            fout << (int)inImg[y*width + x] << " ";
        }
        fout << endl;
    }
    fout.close(); // File closed
    // Image array for CUDA
    float* imgDataCUDA;
    imgDataCUDA = (float*)malloc(width*height*sizeof(float));
    for (int i=0; i < width*height; i++) {

```

```

        if (inImg[i] > 0) {
            imgDataCUDA[i] = 1;
        }
        else {
            imgDataCUDA[i] = 0;
        }
    }
    // Variables for Hough
    int radius;
    // Arrays for results
    int *posxArray, *posyArray, *maxValArray, *radArray;
    // Allocate correct memory for arrays
    posxArray = (int*)malloc((radMax-radMin)*sizeof(int));
    posyArray = (int*)malloc((radMax-radMin)*sizeof(int));
    maxValArray = (int*)malloc((radMax-radMin)*sizeof(int));
    radArray = (int*)malloc((radMax-radMin)*sizeof(int));

    // Perform task for all the radius
    Hough2D_CUDA(imgDataCUDA, width, height, radMin, radMax, resX, resY,
resVal, resRad);

    return 0;
}

int imadjustInput(IplImage *cvImg, float lowPerc, float highPerc)
{
    // Get dimensions of the image
    int width = cvImg->width;
    int height = cvImg->height;

    // ### Convert IplImage to an array ###
    unsigned char *inImg, *temp2, *outImg;
    inImg = new unsigned
char[cvImg->width*cvImg->height*cvImg->nChannels];
    outImg = new unsigned
char[cvImg->width*cvImg->height*cvImg->nChannels];
    temp2 = inImg;

    // pointer to imageData
    unsigned char *temp1 = (unsigned char*) cvImg->imageData;
    // copy imagedata to buffer row by row
    for(int i=0;i<cvImg->height;i++)
    {
        // memory copy
        memcpy(temp2, temp1, cvImg->width*cvImg->nChannels);
        // imageData jump to next line
        temp1 = temp1 + cvImg->widthStep;
        // buffer jump to next line
        temp2 = temp2+ cvImg->width*cvImg->nChannels;
    }

    // Adjust image intensity
    imadjustCUDA(inImg, outImg, width, height, lowPerc, highPerc);

    // Convert the image back to openCV format
    cvSetData(cvImg, outImg, width);
    return 0;
}

```

Abstract

The algorithms used in Digital Signal Processing, such as Image Processing, Object Detection fields are used in important works research, thus require highly intensive and complex computing which usually executed on Supercomputers located in big research centers. These supercomputers consume a lot of power and came with prohibitive cost which can't be afforded by most of researchers especially in MENA region.

Offering low-cost Mobile Heterogeneous Computer for scientist and researchers to be used as supercomputer will accelerate scientific experiments, thus implementation results can be achieved with an acceptable time rather than waiting a long time or stay in a theoretical study only.

In this research, a study on how to use Graphical Processing Units GPUs as a general purpose programming tool GPGPU involved in heterogeneous computing has been made, and how much affect the speed and performance of algorithms and computation which used in Digital signal processing. Then Mobile Heterogeneous Computing had introduced by giving examples on how to use the NEON module in ARM-based SoCs to speed up the calculation for some arithmetic operations, then the result compared against normal heterogeneous computers in important keys like performance, cost, power consumption and the size.

For the real implementation, techniques used in Iris Recognition systems have been studied, and the focus was on Iris Localization step to accelerate it by converting available serial code to parallel code, then the result had checked to show how much speedup it can get after porting the code to Jetson TK1 development kit from NVIDIA.

Damascus University

Faculty of Mechanical & Electrical Engineering

Electronic & Communications Engineering Department



Improving the performance of algorithms used in digital signal processing using mobile heterogeneous computing

Scientific Research prepared to obtain Master Degree in
Applied Electronics Engineering

By

Eng. Loay Abdolsaheb Yari

Supervised by

Prof. Dr. Eng. Jabr Al-Romhein

Academic Year

2016 - 2017