網路黑貓 BlackCat dot Net

我的生活雜記, 包含了心得感想, 電腦新聞和技術探討等

2017年5月5日 星期五

Blocks 初探與 multithreading 應用

近日由於個人一項工作的緣故, 為了能在短時間內能夠加速複雜程式的運作
因此現學現賣地採用了 OpenMP 做快速的優化實作, 最後得到 3倍左右的加速

儘管 OpenMP 表現不俗, 然而在 Android 上的實作必須仰賴已經被 deprecated 的 GCC
讓現今預設使用 clang 的環境必須特別撰寫 makefile
而儘管亦能夠使用先前撰文介紹的 Grand Central Dispatch 作為方案
然而一方面 libdispatch 的 Android port 已經過舊
而官方版必須限定 Android API-level 21 外 (另外現在編譯也有問題)
另一個問題是 GCD 的實作規模不可以說小

由於 Blocks 的使用其簡潔與動態的特性對於個人而言是充滿魅力的
基於如此的動機便開始構思結合 thread pool 與 Blocks 的方式
以此來簡化 multi-threading 程式的撰寫, 並且得到能夠有效修改的實作
為此目的必須先去了解 Blocks 是如何運作的
儘管 clang 提供了 "Language Specification for Blocks" 的頁面
然而個人認為 Apple 的 Blocks Programming - Introduction 撰寫的比較淺顯易懂

基本上 Blocks 提供了將 C/C++ 的大括號內的 code 區塊轉化作為類似下列型別作為"變數"的能力

```
void (*func)(void);
```

而 Blocks 中最有趣以及最為實用的功能是對於使用 global/local variable 上數值的"擷取"
以 local variable 為例, 一般的 serial code 毫無疑問會在 stack memory 中
若使用了 Blocks 並於 Blocks 中使用了該 Block 區間外的 global/local variable
這時的流程會產生了類似 process fork 的分歧, 理解上應為未明確寫出的 call by value
Blocks 中對於外部的 global/local variable 本身的修改是不具有寫回的效果
除此之外型別為 Blocks 的變數在使用上的概念其實與一般變數無異
程式撰寫的過程中同樣地必須考量與處理 variable lifetime 的問題
這時就必須藉由使用 Block_copy/Block_release 來手動複製與釋放 Blocks 所含的內容
對於程式中 local/global variable 處理概念上的不同是 OpenMP 與 Blocks 最大差異
而兩者所使用的方式, 在應用上來說真的是各有優缺

透過閱讀上述的參考資料建立概念後
接著就動手來做類似於 GCD 的 Blocks dispatching 的功能

首先是建立等同於 GCD 中使用來作為 task dispatch 的 dispatch_block_t 的型別
接著就是增加 thread pool 的 dispatch function
基本上是將原本的介面的參數自 function pointer 與型別為 (void *) 的 argument
改為直接使用 Block 型別
可不用再撰寫型別為 void* func(void*) 的 pthread glue code 的好處不用再多說了

在建立概念之後, 動手實作上就簡單多了
為了快速而採用了現成的 thread pool 實作 - C-Thread-Pool
而 Blocks 的操作是基於 BlocksRuntime (提供了 Block_copy / Block_release)
而初步的成果我暫且名為 gunshot
請 git clone 後記得 git submodule init/update

修改的 example.c 中, 嘗試比較填入一個 buffer 數值

```
for(int pidx = 0; pidx < TEST_DEPTH; pidx++){
    int *plane = buf0 + pidx*TEST_W*TEST_H;
    for(int yidx = 0; yidx < TEST_H; yidx++){
        for(int xidx = 0; xidx < TEST_W; xidx++){
            plane[yidx*TEST_W + xidx] = pidx*4096 + (yidx + xidx);
        }
    }
}
```

而 thread pool + Blocks 的版本可以寫為

```
for(int pidx = 0; pidx < TEST_DEPTH; pidx++){
```

搜尋此網誌

搜尋

- 首頁

關於我自己

champ yen
追蹤 134

Black Cat Dot Net
網路黑貓

檢視我的完整簡介

網誌存檔

- ▼ 2017 (25)
  - ▼ 五月 (1)
    - Blocks 初探與 multithreading 應用
  - ► 四月 (2)
  - ► 三月 (4)
  - ► 二月 (3)
  - ► 一月 (15)
- ► 2014 (4)
- ► 2012 (1)
- ► 2011 (4)
- ► 2010 (25)
- ► 2009 (58)

標籤

- adreno
- Android
- architecture
- arm
- audio
- book
- bookmark
- clang
- embedded
- file system
- firefox
- FPGA
- google
- graphics
- introduction
- life
- linux
- meego
- misc
- multimedia
- multithreading
- music
- neon
- office
- omap

```
        int *plane = buf1 + pidx*TEST_W*TEST_H;
        thpool_add_block(thpool, ^{
            for(int yidx = 0; yidx < TEST_H; yidx++){
                for(int xidx = 0; xidx < TEST_W; xidx++){
                    plane[yidx*TEST_W + xidx] = pidx*4096 + (yidx + xidx);
                }
            }
        });
    }
```

在個人使用的 Quad-Core A8-5545M 平台得到了以下結果
single thread - 34832 us
4 threads - 13129 us

得到了 2.65 倍的加速

於 5月 05, 2017 沒有留言: 這篇文章的連結

標籤： multithreading, optimization, programming

2017年4月19日 星期三

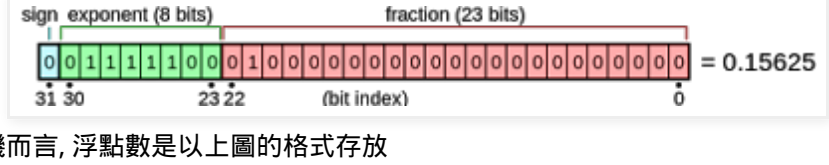
浮點數的美麗與哀愁

這幾年個人在影像處理程式優化的領域打滾, 如果問到感到棘手的工作, floating point 的處理應該可以排上很前面的名次

在許多演算來說由於同時對於 precision 與 dynamic range 的需求, 因此在計算過程中對於浮點數的使用是非常常見的 (若要避免使用會有很高的專業與困難度), floating 主要優點在於可以表示極大與極小值, 相較整數能大幅避免 overflow 與 underflow, 缺點是有效位數的減少, 而且現今多數的計算單元都俱備 floating 的支援, 已經讓一些人疏於了使用浮點的問題, (包含486與之前的時代 FPU是高檔貨, ARM 也自 ARMv7 才列標配)
然而若橫跨了 PC 與 手機, CPU 與 GPU, CPU 與 DSP, 甚至於三者 (PC, 手機, GPU), floating point 就變成非常難以考量與處理的負擔, 而為了區分是程式錯誤或是誤差就必須耗費相當的心力

為了簡化問題, 因此文中談到 floating point 若無指名, 一律是指 32bit single precision, 但相同的問題 64bit double precision 中一樣存在

IEEE 754
首先必須要談的是問題的核心 - IEEE 754



對於計算機而言, 浮點數是以上圖的格式存放
fraction 一共 23 bits 存放一個介於 1~2 之間的數目, 這 b22 ~ b0 存放的是二進位小數以下的部分, 也就是說 fraction 所表示的數值為:

    1 + b22*(2^-1) + b21*(2^-2) + ... + b0*(2^-23)

而 exponent 代表著指數, 一樣以 2 的只數次方表示, 具有 8bit, 因此值域為 0~255, 但是預設會減去 127, 所以即為 -127 ~ 128, 因此對於 exponent 本身所表示的數值為:

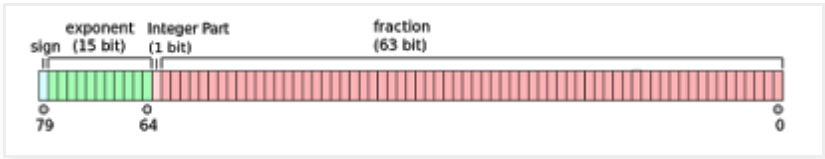
    2^(exponent - 127)

而 sign 就不用多談了, 這是用以表示正負

然而 IEEE 754 定義的不僅僅只是 format 而已, 還有著 rounding mode, required operations 以及 exception handling, 符合了 IEEE 754 的規範下, 才可能有相同的輸出結果 (這當然只是一個最低門檻)

Format 本身的問題
扣除浮點數因格式問題不可能表示全部的數目外
格式本身最大的問題是因為 dynamic range 的移動, 像是 (A+B)+C 與 A+(B+C) 單以代數考量這無疑是相等的, 但是若以 floating 格式去思考, 你就會意會到輸出結果很有可能會不同, 而原始演算實作所採用的累加或相乘的順序, 必須在優化實作上努力維持才能產生一致的輸出結果, 這樣的問題對於程式優化影響最大的部分是平行計算, 無論 TLP or ILP, 因為平行度優化考量, 都會有分割與不同面向個別累計的需求, 如此勢必都會產生一定的誤差

CPU 間的問題
或許有些人會認為只使用 CPU 那麼就不會碰到浮點數問題了, 這樣說只算對了一半, 而且你還是必須要只使用一種平台以及指令集, 對於多數演算法設計工程師而言, 他們很慣於使用 PC 平台, 甚至會使用 MATLAB, x86 PC 上的程式預設會使用 x87 浮點數協同單元 指令集, 而這是許多問題的開始 - x87 內部使用 80bit 浮點數表示



通常 x86 CPU 是在 x87 FPU 的內部以 80b 計算得到結果後, 再 truncate 為 IEEE-754 的 float(32b) or double(64b), 這就表示這與 IEEE 754 FPU 的輸出結果會有微小的差異, 目前常見的手機的 ARM 架構, 即為 IEEE 754 compatible FPU, 所以光是 ARM 與 x86 PC 相同的程式碼其輸出結果基本上就會有所不同, 而對於 ARM 與 x86 的部分, 就必須仰賴以 IEEE 754 設計的 SSE2 指令集, 若是 gcc 與 clang 很早就可透過 -mfpmath=sse2 的編譯參數來達到, 但是 Visual Studio 必須是 2013 版後才有正確的 code generation 實作, (也就是說 Windows 的使用者要安裝 VS 2012以後的版本 才有可能透過 /arch:sse2 有一致的輸出)

對於 ARM 與 x86 平台的一致性方案反而又揭露了另一個層面問題:

> 就算使用了單一CPU架構, 在 ISA 指令集間的支援還是會有所不一致!

類似於 x86 平台上 x87 指令與 SSE2 指令有著不同輸出, 同樣地 ARM VFP 指令(IEEE 754 相容) 與 NEON 指令(非完全 IEEE 754 相容) 也可能會有輸出結果不同, 而這樣的問題還會再帶到 libmath 的實作方式, 讓要處理一致性的問題再度的變得更嚴重

GPU
GPU 本身有著龐大的浮點數計算能力, 但是通常為了能達到更高的吞吐量以及加速上的考量, 在計算結果與 IEEE-754 可能存在差異, 不同代的 GPU 或是不同架構都有可能有所不同. 像是 CUDA 是在 compute compatibilty v2.0 之後才完備了 IEEE 754 的支援, 除此之外許多硬體加速的數學函數的輸出上也不保證與 CPU 一致, 這點 Nvidia 在 2011 GTC 中給的 Floating Point and IEEE-754 Compliance for Nvidia GPUs 簡報中有很詳盡的說明. 對於其他GPU 以及各 CPU/GPU 平台上的 OpenCL 中的 built-in functions 的實作與支援也有著相同的道理.

DSP
對於 DSP 而言這樣的痛苦並不在於 IEEE 754 本身, 而是多數的多媒體面向的 DSP 為了考量計算能力與面積, 結果多半是直接不俱備 floating 能力的, 像是 Qualcomm S82x 中的 Hexagon 680 HVX 就不俱備 floating 運算的 SIMD 指令, 而通常的處理作法是採用 fixed-point (quantization) 的浮點模擬, 然而若採用靜態位數的方式容易失真, 而動態的方式有著實作上的複雜度以及多餘計算的負擔. 而數學函數上的實作若難以避免則通常必須透過相當紆迴的方式.

Lookup-Table or Frame-based Parameters
對於跨裝置的正確性驗證, 由單一裝置輸出的 Lookup Table(單一的 math function 像是 sin, cos, log, exp 等等) 或是一整張預先透過單一裝置計算的 Frame-based Parameters(複雜的並結合多個 math function 的運算) , 是常用來確認誤差單純是由 floating 計算造成的技巧. 以此來確保實作上的流程與邏輯無誤.

延伸閱讀: The pitfalls of verifying floating-point computations

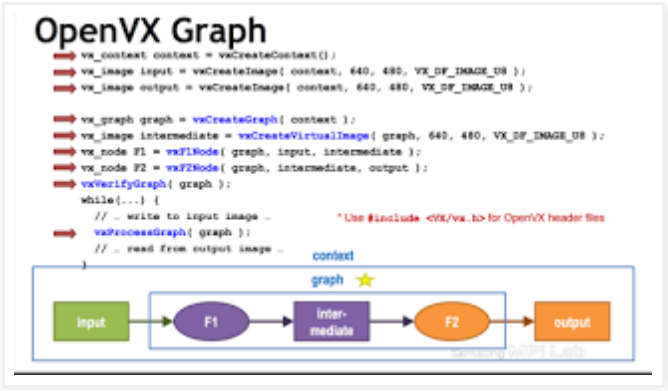
於 4月 19, 2017　4 則留言:　這篇文章的連結

標籤： optimization, programming

2017年4月2日 星期日

"ARM Compute Library for computer vision and machine learning" III - 總結

在實作上核心的實作是在各功能的 Kernel 類別實作中
因此若想瞭解可以去研讀各個繼承 IKernel 的 CL/NEON 實作
由系列文 II 多少可以了解 ARM Compute Library 是如何的工具
在官方的介紹也說明了這是 - a collection of low-level software functions
這樣的好處是設計簡單且易於使用, 若所需要功能不複雜, 其所提供的工具也相當堪用
但是若一個目的是需要使用多個 Kernel 的串連來達成
如此的應用就需要更進階的方式來作優化
以 OpenVX 來說即為其 Graph pipeline
基本上需要透過一個更為高階的抽象層
為問題帶入各個 stage 的分割與相關排程的分析
對於 ARM Compute Library 而言每個 function 需要 I/O image buffer
能接近這樣的方式在於兩個 stage 間以 Window + Thread 的 Tiling 方式
如此也僅是有限度地利用 data locality 的特性增加 cache 的有效性
況且對於 Load/Store 指令可是一個都沒能因此節省
(這需要能作 operation fusion 的 compiler)
這即是 ARM Compute Library 在效能與進階功能上的局限



然而若需要進一步解決上述的局限
會需要能針對 temporal/spatial scheduling 作 dynamic code generation 的 compiler以及 runtime
實作複雜度亦會大幅增加 (即 OpenVX/Halide 或類似的實作)

於 4月 02, 2017　2 則留言:　這篇文章的連結

在 Google 上推薦這個網址

標籤： arm, neon, OpenCL, programming, simd, study

2017年3月28日 星期二

"ARM Compute Library for computer vision and machine learning" II - Framework 篇

ARM Compute Library 的使用上的 class 主要有二類分別是針對 data 以及 task/workload
data 類別為: Image/Tensor, TensorInfo
task/workload 類別為: Kernel, Window 與 Function
下列的內容主要為 ARM Compute Library: Documentation 所描述

並且搭配 source code 的內容做特定用途的說明
(取代文件內 MyKernel, MyFunction 的方式)

由於 ARM Computer Library 是做 Computer Vision 與 Machine Learning 應用的
因此主要處理的資料型別為 Image 及 Tensor
在 Compute Library 中基本上只是名稱不同而已
Image, Tensor, TensorInfo
在 NEON 下直接使用 Image

```
Image     src, dst;
```

而在 CL 下則使用 CLImage

```
CLImage   src, tmp, dst;
```

Image 宣告後並沒有實際的 buffer 空間, 必須進一步最配置的動作,配置的方法有二, 兩者都需要傳遞 TensorInfo 資訊, TensorInfo 基本上為 Image/Tensor 各維度的大小以及資料格式
配置的第一種方式為直接透過 Allocator 的 init() 方式

```
src.allocator()->init(TensorInfo(640, 480, Format::U8));
```

而第二種方式為先呼叫 configure() 在呼叫 Allocator 的 allocate()

```
TensorInfo dst_tensor_info(src.info()->dimension(0) / scale_factor,
src.info()->dimension(1) / scale_factor, Format::U8);
dst.allocator()->init(dst_tensor_info);
dst.allocator()->allocate();
```

Kernel, Window & Function
空間配置好之後, 就必須透過各種各樣的 Kernel 來套用對應的功能來操作 Image/Tensor
使用上的核心 class 為 Kernel, 各個 Kernel 實作了 IKernel 相關的介面

使用的第一個步驟為宣告想使用的 kernel object, 假設我們想作 image scale

```
//Create a kernel object:
NEScaleKernel scale_kernel;
```

在使用之前必須對 kernel 作 input, output 使用參數以及 padding mode 作設定

```
// Initialize the kernel with the input/output and options you want to use:
Tensor offsets;
const TensorShape shape(dst.info()->dimension(0), dst.info()->dimension(1));
TensorInfo tensor_info_offsets(shape, Format::S32);
tensor_info_offsets.auto_padding();
offsets.allocator()->init(tensor_info_offsets);
scale_kernel.configure( &src, nullptr, nullptr, &offset, &dst,
InterpolationPolicy::NEAREST_NEIGHBOR, BorderMode::UNDEFINED);
```

```
offsets.allocator()->allocate();
// compute offset
...
```

這裡使用了 NEAREST Filter, 並且使用 UNDEFINED padding 方式 (對於邊界有缺所需資料的點不做處理)

最後就是呼叫使用該功能,即是呼叫 IKernel 的 run() 介面

```
// Retrieve the execution window of the kernel:
const Window& max_window = scale_kernel.window();// Run the whole kernel
in the current thread:
scale_kernel.run( max_window ); // Run the kernel on the full window
```

這些即為 Compute Library 基本的使用方法.
對於 CL Kernel 則有稍微不同的 flow (需要特別傳入以操作 cl::CommandQueue)

或許會覺得上面例子的 max_window 很多餘, 但它是有進階應用的
Window 用途在於對 Kernel 指定要套用執行的範圍描述
在官方說明文件是以 Multi-Threading 的方式來說明 Window 的用途

```
const Window &max_window = scale_kernel->window();
const int num_iterations = max_window.num_iterations(split_dimension);
int num_threads    = std::min(num_iterations, _num_threads);

for(int t = 0; t < num_threads; ++t){
   Window win = max_window.split_window(split_dimension, t, num_threads);
   win.set_thread_id(t);
   win.set_num_threads(num_threads);
   if(t != num_threads - 1){
      _threads[t].start(kernel, win);    }else{
      scale_kernel->run(win);    }
}
```

當下列所有的條件都符合後, Window 可以用來分割 workload 為多個子 Window

- max[n].start() <= sub[n].start() < max[n].end()
- sub[n].start() < sub[n].end() <= max[n].end()
- max[n].step() == sub[n].step()
- (sub[n].start() - max[n].start()) % max[n].step() == 0
- (sub[n].end() - sub[n].start()) % max[n].step() == 0

至於 Function 的使用則是為了簡化繁雜的 Kernel, Window 的使用流程, Function 實作內部會自行配置所需的暫存 buffer, 甚至能透過上述的方式自行做 Multi-Threading

```
// Create and initialize a Scale function object:
NEScale scale;scale.configure(&src, &dst,
InterpolationPolicy::NEAREST_NEIGHBOR, BorderMode::UNDEFINED);
// Run the scale operation:
scale.run();
```

若使用的為以 CL 實作的 Kernel 最後還有個確保執行完成的額外同步動作

```
// Make sure all the OpenCL jobs are done executing:
CLScheduler::get().sync();
```

如此一來 Function 比起直接使用 Kernel 簡化不少

在下一篇會進入 NEON 與 CL 內部實作方式的說明

於 3月 28, 2017　　沒有留言:　　這篇文章的連結

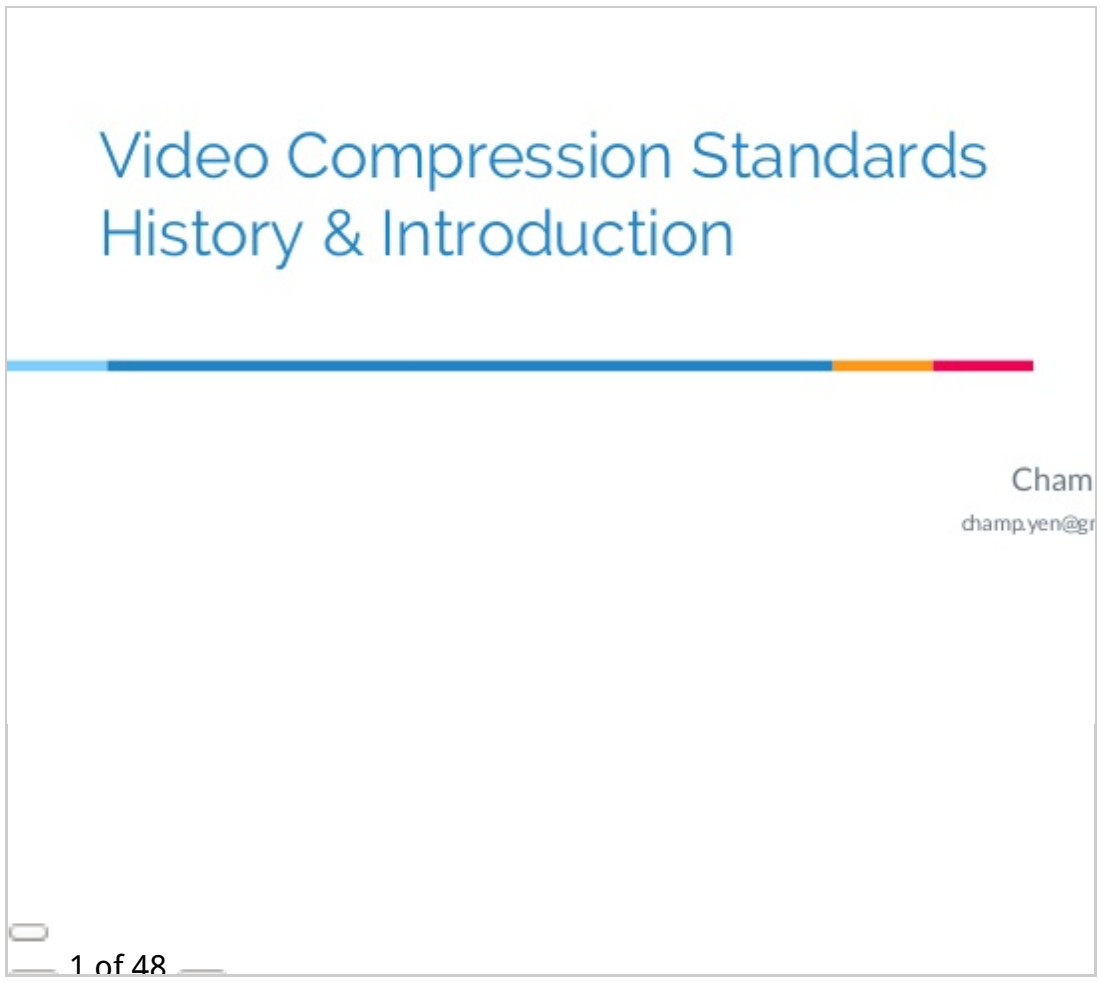
G+1 在 Google 上推薦這個網址

標籤： arm, neon, OpenCL, optimization, simd

2017年3月26日 星期日

簡報 - Video Compression Standards - History & Introduction

這份簡報是一年半前為了數位電視課程所準備, 發現 blog 沒有紀錄所以發文分享與 Link 一下



Video Compression Standards - History & Introduction from Champ Yen

於 3月 26, 2017 沒有留言: 這篇文章的連結

在 Google 上推薦這個網址

"ARM Compute Library for computer vision and machine learning" I - Overview 篇

日前 ARM 官方透過 github 並且以 MIT License 方式再次正式地釋出了 Compute Library 的原始碼(先前提供了 internal evaluation only 的 binary, 詳請請回顧當時的 release note), 這是個 low-level implementation, 而且是 pure function 的形式, Compute Library 提供了涵蓋下列的功能函式:

- 基本的運算, 數學與布林運算函式 (Basic arithmetic, mathematical, and binary operator functions)
- 色彩操作, 包含轉換, 頻道擷取與其他 (Color manipulation (conversion, channel extraction, and more))
- 捲積濾波器 (Convolution filters (Sobel, Gaussian, and more))
- Canny Edge, Harris corners, optical flow, and more
- Pyramids (such as Laplacians)
- HOG (Histogram of Oriented Gradients)
- SVM (Support Vector Machines)
- 半/全精準 通用矩陣乘法 (H/SGEMM (Half and Single precision General Matrix Multiply))
- 捲積類神經網路建構功能區塊 (Convolutional Neural Networks building blocks (Activation, Convolution, Fully connected, Locally connected, Normalization, Pooling, Soft-max))

對於 Compute Library 來說, 它屬於個人介紹過的(請參考SIMD Introduction 簡報) SIMD Programming Model 中的 SIMD Optimized Library, 概念與與提供的效能上, 可以參考 ARM 官方釋出的介紹文, 本系列文會專注於 Compute Library 內部架構與更細節的如何使用, 所提供的能力以及, 以及探討使用這樣的 Library 依然存在什麼樣的限制.

首先 ARM Compute Library 其 github 位置為 https://github.com/ARM-software/ComputeLibrary , 而相關的原文文件在 source 與網站上各有一份

以目錄結構來說下列為主要較重要的目錄:

- arm_compute/ - 放置所有 Compute Libraray 的 Headers
  - core/ - Core library 是由底層演算法的實作所組成
    - 基本共用資料型別 (Types, Window, Coordinates, Iterator, 等等)
    - 基本通用介面 (ITensor, IImage, 等等)
    - 物件 metadata 型別 (ImageInfo, TensorInfo, MultiImageInfo)
    - backend 目錄

    - runtime/ - Runtime library 是用來快速 prototyping 用途非常基本的 Core Library 的 wrapper (由於 CL/NEON 的 Programming Model, 這裡提供對應不同的 execution interface)
        - 基本通用物件介面的實作(Array, Image, Tensor, etc.)
    - 以上兩者, 內各自有 CL/CPP/NEON backedn 目錄, 提供對應 backend 定義的 kernel headers
- documentation/ - Doxygen 所產生的文件
- examples/ - 內有提供的 4 個範例程式
- include/ - 基本上只放置 OpenCL 1.2 的 Headers
    - CL/
- src/
    - core/ - 於 arm_compute/core/ 中定義的型別/介面的實作
    - runtime/ - 於 arm_compute/runtime/ 中定義的型別/介面的實作
    - 以上兩者, 內各自有 CL/CPP/NEON 目錄, 即為該 backend 實作相關原始碼

而值得一提的是在 Compute Library 中提供的功能中, 這些 Kernel 演算所使用的定義規範為 OpenVX 1.1 所制定的

以下為目前提供的 Kernel 列表, 若了解 image processing, DNN 該 function 名稱應解釋了其功用, 即不在此冗文解釋: (注明 NEON-only 表示目前尚未有 CL 實作)
AbsoluteDifferenceKernel
AccumulateKernel
ActivationLayerKernel
ArithmeticAdditionKernel
ArithmeticSubtractionKernel
BitwiseAndKernel
BitwiseNotKernel
BitwiseOrKernel
BitwiseXorKernel
Box3x3Kernel
CannyEdgeKernel
ChannelCombineKernel
ChannelExtractKernel
Col2ImKernel
ColorConvertKernel
ConvolutionKernel
ConvolutionLayerWeightsReshapeKernel
CumulativeDistributionKernel (NEON-only)
DepthConvertKernel
DerivativeKernel
DilateKernel
ErodeKernel
FastCornersKernel
FillArrayKernel (NEON-only)
FillBorderKernel
FillInnerBorderKernel (NEON-only)
Gaussian3x3Kernel
Gaussian5x5Kernel
GaussianPyramidKernel
GEMMInterleave4x4Kernel
GEMMLowpMatrixMultiplyKernel
GEMMMatrixAccumulateBiasesKernel
GEMMMatrixAdditionKernel
GEMMMatrixMultiplyKernel
GEMMTranspose1xWKernel
HarrisCornersKernel
HistogramKernel
HOGDescriptorKernel (NEON-only)
HOGDetectorKernel (NEON-only)
HOGNonMaximaSuppressionKernel (NEON-only)
Im2ColKernel
IntegralImageKernel
LKTrackerKernel
MagnitudePhaseKernel
MeanStdDevKernel
Median3x3Kernel
MinMaxLocationKernel
NonLinearFilterKernel
NonMaximaSuppression3x3Kernel
NormalizationLayerKernel
PixelWiseMultiplicationKernel
PoolingLayerKernel
RemapKernel
ScaleKernel

Scharr3x3Kernel
Sobel3x3Kernel
Sobel5x5Kernel
Sobel7x7Kernel
SoftmaxLayerKernel
TableLookupKernel
ThresholdKernel
TransposeKernel
WarpKernel (CL 細分為 WarpAffine, WarpPerspective 兩種)

下一篇將會介紹 Compute Library 中使用所需了解的基本型別, 介面, 執行方式以及範例

於 3月 26, 2017　沒有留言:　這篇文章的連結

標籤： arm, neon, OpenCL, OpenVX, programming

2017年3月19日 星期日

OpenCL Programming Tips for Qualcomm Adreno GPU 導讀

目前手機多半都有內建 OpenCL Runtime
(除了 Google 無謂地堅持 RenderScript, Do Evil 地阻礙標準的採用)
對於 OpenCL 有所了解的人, 多半清楚 OpenCL 是 function portability, 而無法做到 performance portability, 這其中的緣由主要還是在於各家的 GPU Architecture 的差異, 因此多半各家 GPU vendor 都會提供自家 OpenCL Optimization Guide, 以利開發者對自家平台優化應用性能

目前手機可能內建的 GPU 主要為三

- ARM Mali
- Imagination PowerVR
- Qualcomm Adreno

由於個人 OpenCL 經驗多半與前兩者有關，由於未曾接觸, 因此對於 Adreno 部分有很大的興趣, 因而選擇研讀與撰文. 這篇主要的內容主要來自下列公開的 Adreno OpenCL Programming Guide 文件. 有機會也會撰文介紹 ARM Mali 與 Imagination PowerVR 的 Programming Guide

Adreno OpenCL Programming Tips

Memory

其第一個章節即是 "Memory", 對於計算架構來說 Memory 幾乎是效能上的關鍵, 對於 GPU 亦不例外, 而在這份文件中 Memory 章節佔了一半的份量, 可見其重要性, 對於 Adreno GPU 而言, 其考量點有五:

- Vectorization and coalescing

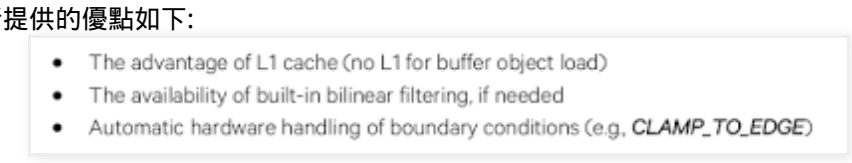
對於 memory access 而言, Coalescing 是常用的方式, 儘管一些 compiler 會透過 auto-vectorization 嘗試去優化 workgroup 內更有效率的存取, 但是 programmer 自行作 vectorization 並控制 access pattern 對於後續的 finetune 是重要的. 一旦 vectorization 後, OpenCL 提供兩個介面做 vector loading, 一者為 vector-based pointer arithmetic, 另一為 vloadn 的方式, 這裡 Adreno 建議以 vloadn 並且不建議 n > 4. (這必定也只是一個 common rule, 程式的流程與記憶體的行為也會有差異)


Since the bit width of Adreno is 128, we recommend that you load/store in batches of 128 bits per transaction.

There is no benefit to using *vload8* or *vload16* over a multiple of *vload4*.

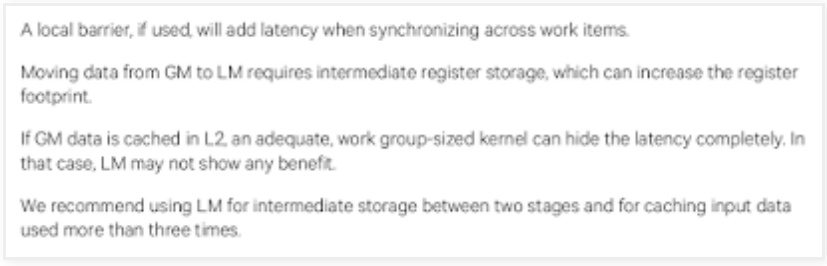
- Image vs. buffer memory objects

OpenCL 的記憶體使用分為兩種 abstract object, 一者為 Buffer 另一為 Image Object, 對於 Image 所提供的優點如下:


- The advantage of L1 cache (no L1 for buffer object load)
- The availability of built-in bilinear filtering, if needed
- Automatic hardware handling of boundary conditions (e.g, *CLAMP_TO_EDGE*)

主要是相較於 Buffer 多了 L1 cache (這是 Texture Unit 的特性所增加), 另外就是硬體加速的線性內插的計算, 以及最後是能夠透過硬體自動處理邊界的問題(以 Buffer 而言需要增加 GPU 所不擅長的 if/else 的 code)

- Global memory (GM) vs. local memory (LM)


A local barrier, if used, will add latency when synchronizing across work items.

Moving data from GM to LM requires intermediate register storage, which can increase the register footprint.

If GM data is cached in L2, an adequate, work group-sized kernel can hide the latency completely. In that case, LM may not show any benefit.

We recommend using LM for intermediate storage between two stages and for caching input data used more than three times.

在 Global Memory(GM) 與 Local Memory(LM) 間應注意的事情有三, barrier 的使用, 再者為搬移資料需要考量的 cost, 最後是將資料存放在 LM 的條件

對於 Kernel 撰寫實作熟悉者, 應該對於 barrier 的使用會有相當的 overhead 不陌生, 但是對於一些有 data dependency stages 的實作這又是必要的, 減少 barrier 的使用幾乎是所有 GPU 平台一致要納入考量的點.
儘管 LM 有著較低的 latency 但對於 Adreno GPU 來說 GM 到 LM 的途徑中需要使用 GPU 內部的暫存器, 隨著需要搬移的資料數目耗費的暫存器可能會引起 register spilling 問題, 另外 Adreno GPU 對於 GM 有 L2 cache, 因此並不是直接將資料放置於 LM 就能獲得效益.
對於需要放置到 LM 的條件, 這裡建議 LM 存放介於兩個 stage 中間的資料, 或是會被使用至少三次的 input data.

- Private memory

> If possible, the compiler will store private memory objects in registers. Otherwise, the compiler will spill to local memory and finally to system memory. At each stage of spilling, performance is lost.
>
> For private arrays, we recommend trying LM instead of private memory.

對於多數的 GPU 架構 private memory 對應到的地方是暫存器, 對於 Adreno 亦不例外, 由於 general register 數目有限, 若不良的 coding style 會造成 register spilling, 而一旦這樣的情況發生, Adreno 會嘗試自 LM 調度, 若 LM 不足則最後會調度到 GM, 這樣的流程若頻繁發生, 可以預期的是效能會大幅降低. 因此對於想要宣告為 array 的變數, 建議直接使用 LM. 其實除了這樣之外, 可以做 in-function 的 multi-stages 方式, 積極地將 kernel 透過多個 stage subroutine 來實作, 最後手段是透過精準的 variable life-scope 控制(也就是加入 {} 大括號, 主動提供 compiler 資訊) 來減少 register 的使用.

- Constant memory

> Up to 3KB of constant memory can be stored in dedicated, constant RAM, which is directly accessible by ALUs. The rest is stored in system memory.
>
> The compiler will attempt to promote constant variables and arrays to constant RAM. However, due to space limitations some constants may not get promoted.

Adreno 提供了相對特別的 constant memory (經驗上來說常看到架構會使用 LM 作為存放 constant 的地方), 然而大小為 3KB, 超過的部分系統會放在 GM 中, 對於透過 Kernel Argument 傳入的 constant buffer 需要透過屬性的設置來預期會被放置於 constant memory.(詳細請參考 1. 文件)

Zero memory copy

> To avoid memory copy when sharing data between different external components and the GPU, use ION memory and the extension ***cl_ion_qcom_host_ptr***. Alternatively, use Android native buffers and the extension ***cl_qcom_android_native_buffer_host_ptr*** to avoid memory copy. Android native buffers are based on ION.

如同其他 OpenCL Runtime, 若需要同時 CPU/GPU 能夠存取, Buffer/Image 的配置要透過 CL_MEM_ALLOC_HOST_PTR 這個 flag 來取得能夠讓 CPU/GPU 無需 data copy 的空間, 在透過 Map/Unmap 的方式來使用. 然而對於除了 CPU, GPU 外的硬體需要使用, 需適當地選擇 cl_ion_qcom_host_ptr 或 cl_qcom_android_native_buffer_host_ptr 這兩個 flag 來使用

Work group size and shape

> Do not expect ***local_work_size=NULL*** to result in the best performance in all cases. The driver attempts to pick a reasonable work group size but that will rarely be the optimal size.

儘管多數 OpenCL 教科書建議在 Kernel 執行時輸入將 local work size 參數傳入 NULL, 讓 Runtime 自動配置最適當的 WorkGroup size, 但是 Adreno 還是提醒這樣的作法其實並不總是最佳的(事實上所有的平台都不是), programmer 應該嘗試尋找適當的 WorkGroup size.

Data type and bit width

> We recommend shorter data types. For example, use ***short*** instead of ***int***. A shorter data type reduces data storage and bandwidth as well.
>
> Regarding half vs. float, 16-bit float (half) is natively supported in Adreno and offers double the throughput of 32-bit float.

資料型別的使用上 Adreno 上建議使用較短的資料型別, 除了能夠減少資料存放的大小與減少頻寬, 像是對於 half (16位元浮點數) 與 float 而言, half 還提供了兩倍的計算能力.

Math functions

> Avoid integer divide or modulo, if possible.
>
> Use ***mul24*** or ***mad24*** if precision is sufficient. 32-bit integer multiplication is emulated using three instructions.

這裡 Adreno 上應避免除法與餘數的計算, 此外若 mul24/mad24 (24位元乘法/乘加) 足夠精準度的話, 應該儘量使用, 這主因是一個32位元的乘法計算在 Adreno 內部是透過3個指令來組成的.

後續會再探討

- Matrix Multiply on Adreno GPUs – Part 1: OpenCL Optimization
- Matrix Multiply on Adreno GPUs – Part 2: Host Code and Kernel

於 3月 19, 2017　沒有留言:　這篇文章的連結

標籤： adreno, OpenCL, programming

首頁 較舊的文章

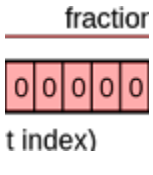
訂閱： 文章 (Atom)

Blocks 初探與 multithreading 應用

近日由於個人 一項工作 的緣故, 為了能在短時間內能夠加速複雜程式的運作 因此現學現賣地採用了 OpenMP 做快速的優化實作, 最後得到 3倍左右的加速 儘管 OpenMP 表現不俗, 然而在 Android 上的實作必須仰賴已經被 deprecated 的 GCC 讓...

解析 Qualcomm Hexagon 680 架構 I - V6x 架構

Qualcomm S835 於不久前發佈 其中內建了強大的 Hexagon 682 DSP 雖然在 S835 的發佈宣傳文 中僅說明了新增了對於 TensorFlow 與 Halide Language 的支援 但是還是有 發佈專文 說明 Hexagon 682 DSP ...



浮點數的美麗與哀愁

這幾年個人在影像處理程式優化的領域打滾, 如果問到感到棘手的工作, floating point 的處理應該可以排上很前面的名次 在許多演算來說由於同時對於 precision 與 dynamic range 的需求, 因此在計算過程中對於浮點數的使用是非常常見的 (若要避免...

"ARM Compute Library for computer vision and machine learning" I - Overview 篇

日前 ARM 官方透過 github 並且以 MIT License 方式再次正式地釋出了 Compute Library 的原始碼(先前提供了 internal evaluation only 的 binary, 詳請請回顧當時的 release note), 這是個 lo...