**運用CUDA加速影像處理演算法(Accelerate Image Processing with CUDA)**

Extended Abstract†

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 林宸皜 National Chiao Tung University Department of Civil Engineering, Team Information Technology Taiwan lin021393.cv05g@nctu.edu.tw | | 蔡文魁 National Chiao Tung University Department of Civil Engineering, Team Information Technology Taiwan .cv05g@nctu.edu.tw | | 徐梓芸 National Chiao Tung University Department of Civil Engineering, Team Information Technology Taiwan .cv06g@nctu.edu.tw | |

**ABSTRACT**

最早的動機是實作一個影像辨識的演算法，並且用Compute Unified Device Architecture (CUDA) [1]加速它。影像辨識的演算法都不容易實踐，但在許多案例中，如Fig. 1所示之物理的干涉圖形案例 [2]，它能以很簡潔的影像處理技術分析物理的干涉圖形，我們的想法是去實踐如物理的干涉圖形這般非過分複雜的影像處理演算法，便能很仔細地分析我們選擇的影像處理演算法其所能平行化的每個部分，使之加速與平行化。



Figure 1: 由Zeeman效應引發的光譜線分裂情形

**–—————————————————————**

\*Produces the permission block, and copyright information

†The full version of the author’s guide is available as acmart.pdf document

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

January 2018, Hsinchu city, Hsinchu county Taiwan

© 2018 Copyright held by the owner/author(s). 123-4567-24-567/08/06. . . $15.00

DOI: 10.1145/123\_4

**CCS CONCEPTS**

• **Operating system** → **Parallel programming**;

**KEYWORDS**

CUDA, GPU, parallel programming, optimization

**ACM Reference format:**

林宸皜, 蔡文魁 and 徐梓芸. 2018. SIG Proceedings Paper in word Format. In *Proceedings of ACM Woodstock conference, Hsinchu City, Hsinchu County Taiwan, January 2018*, 9 pages.

DOI: 10.1145/123 4

**1 INTRODUCTION**

此專題討論之題目是以影像處理演算法來實踐錢幣計數，我們將每個錢幣其圓圈的邊緣圈取出來，接著，可以計算出錢幣的半徑與圓心，這便是此專題要達成之目標。

此專題所使用之演算法流程非常簡潔，如 Fig.2所示，先以大津二值化演算法(Otsu's method) [3]，透過二值化辨識圖片中所有錢幣的所在範圍，彷彿圈選出圖片中各個錢幣所在的位置與範圍，接著，會使用侵蝕(Eroding)與膨脹(Eroding) [4]，盡可能地抹除所有錢幣上的白點，再來，對圖片做Canny邊緣偵測器(Canny Edge Detector) [5]，提取出各個錢幣的邊緣，最後以Hough轉換(Hough Transform) [6]計算出各個錢幣的圓心位置與半徑。

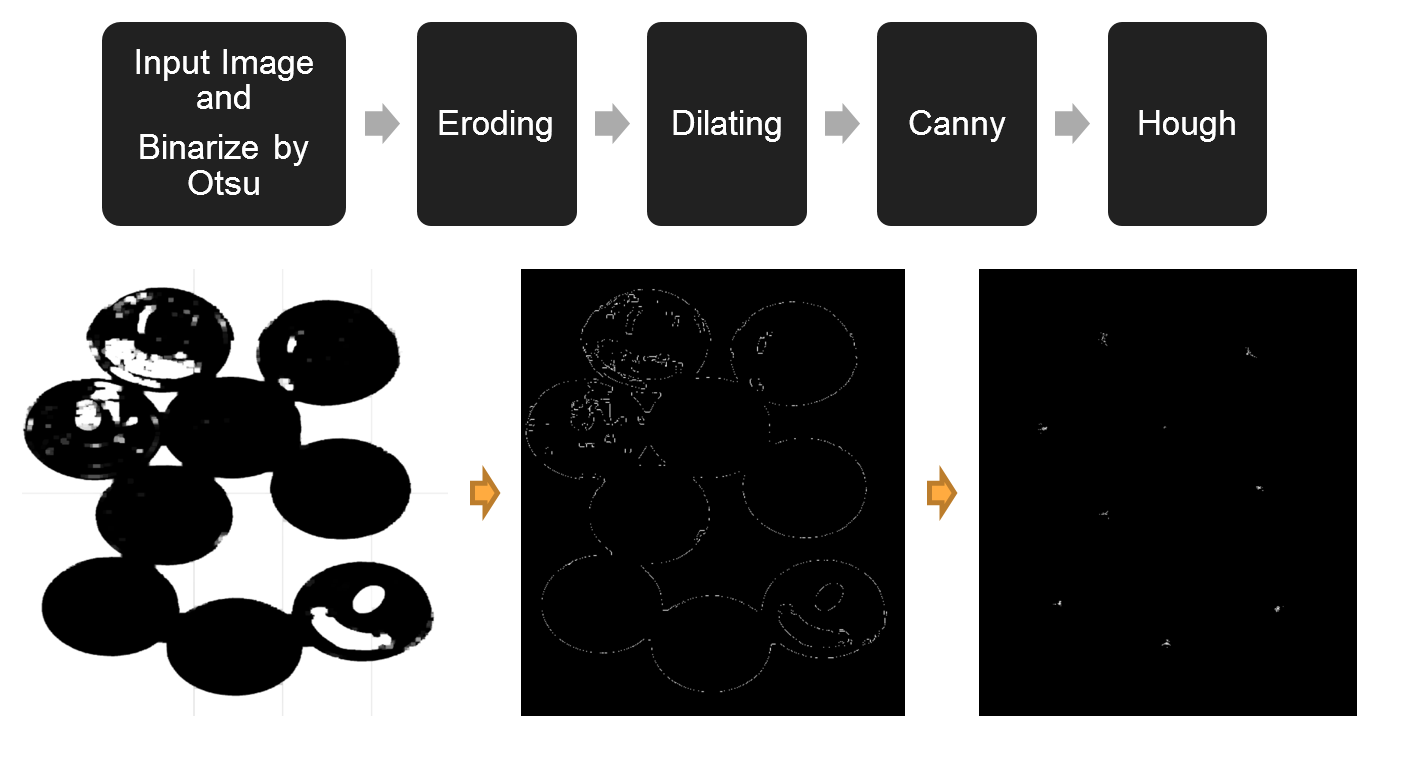


Figure 2: 錢幣計數的影像處理流程

關於Canny邊緣偵測器方面， Canny邊緣偵測器的實質內容是取梯度，即為執行摺積(convolution) [7]，執行摺積的過程中，將不斷計算像素(pixel)邊緣的值，對接下來內容中的效能檢測會有明顯的影響，將在後續內容做詳細的解釋。  
 除此之外，Hough轉換的實質內容等同於對每一個邊緣再畫一個圓，接著去做累加，故Hough轉換也會影響到邊緣的值。

但是，Canny邊緣偵測器與Hough轉換影響邊緣的值之程度強弱，兩者明顯不同，此點會影響程式效率，至此，可以想像Hough轉換是畫一個圓圈，Canny邊緣偵測器則是周圍的每個點都會被其影響到。  
 最後，此專題選擇以CUDA加速上述之影像處理演算法整體流程。

**2 PROPOSED SOLUTION**

我們選擇以 CUDA 程式語言來實作我們的專題平行化部份，相對於 CPU base的語言，如: Pthread [8]與 OpenMP [9]，我們認為 GPU 對於影像處理演算法之平行性質會有較高的優化表現。

我們以一個 OpenCV [10] 版本的影像辨識程式作為參考，實作一個功能相似的 Cpp 版本，讀取影像與結果呈現仍然使用 OpenCV API，我們的焦點在於對影像處理演算法的研究，再以 CUDA 進行加速，由於影像處理演算法基本上每個像素皆為獨立的，因此每個像素分配一個 GPU thread 進行運算，threadblock 與 thread 都用一維得以方便資料傳輸，CUDA 加速以每個演算法獨立實作進行，共享儲存於global memory的資料。

**3 EXPERIMENTAL METHODOLOGY**

**3.1 測試平台**

Table 1: 測試環境中裝置的細節資料

|  |  |
| --- | --- |
| GPU | NVIDIA GeForce GTX 1070 8GB |
| CPU | Intel Xeon E5620 @ 2.40GHz 16C/32T |
| OS | Ubuntu 16.04 |
| CUDA | version 7.5 |
| Other  CUDA compiler | GPUCC |
| Comparison | OpenCV / Cpp / CUDA / GPUCC |

**3.2 此專題之實驗使用的測資**

Table 2: 所使用的測試資料

|  |  |
| --- | --- |
| image ID | image size |
| image1 | 1.1 MB |
| image2 | 2.1 MB |
| image3 | 4.1 MB |
| image4 | 8.0 MB |
| image5 | 16 MB |

**4 EXPERIMENTAL RESULTS**  
此專題最早先實作出OpenCV版本，緊接著再改成我們自己寫的Cpp版本，最後，則實作成CUDA版本。

如Fig. 3所示，我們也先把OpenCV列入考慮，因為OpenCV是此專題最早的執行版本，由OpenCV可以看到最後的執行層，OpenCV的執行效果會爆炸性地成長，起因於OpenCV的Hough轉換是用極其複雜的方式實作，但此專題設計的實作則並非如此複雜，此專題把將Hough轉換做完後，接著，掃描一個範圍內之Hough轉換的圖，此專題並沒有採用OpenCV內部過於複雜的方法，故此專題之實作在輸入為尺寸較大的圖時其表現會明顯較好，此專題主要是研究我們自己實作的版本與使用CUDA加速來實作的版本之間兩者的區別。

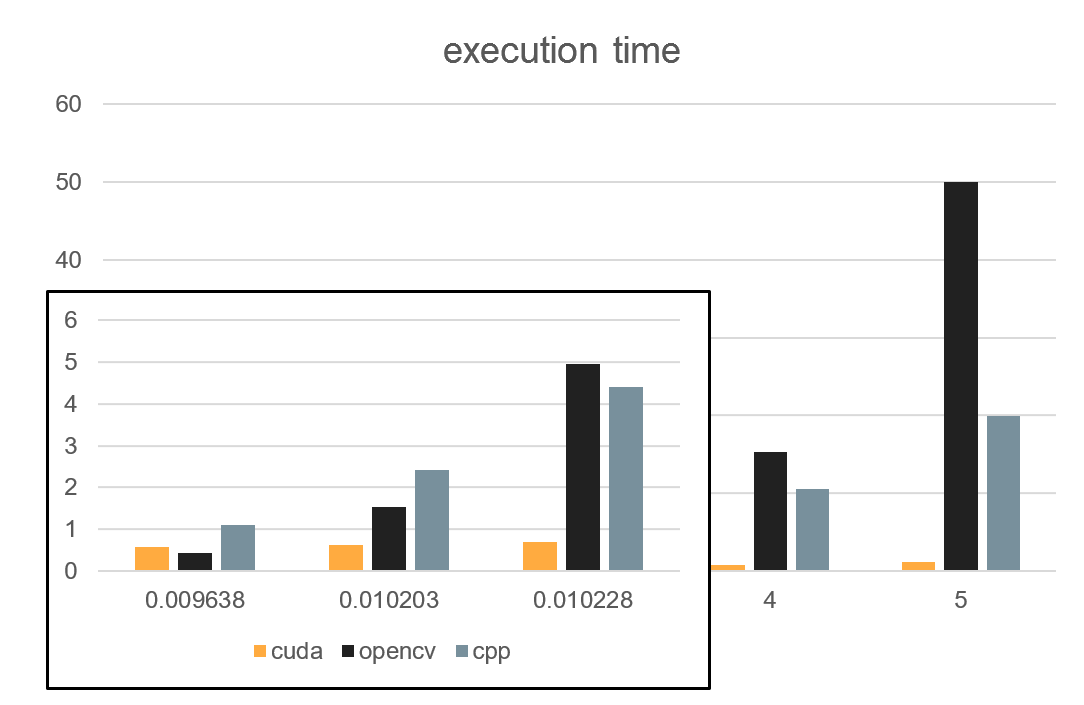


Figure 3: 實際的執行時間

因為OpenCV本身內建單指令流多資料流(Single Instruction Multiple Data, SIMD)優化與多執行緒(Multithreading)優化，所以，即使是CPU版本，除了OpenCV的製作團隊，其他人非常難完全了解OpenCV內部完整的優化內容，故我們專注於討論我們自己實作的影像處理演算法與運用CUDA加速這些影像處理演算法後完成的成果，比較兩者之間有何差異。  
 在此有前三張的圖，可以得知OpenCV版本在輸入的圖片其尺寸較小時，OpenCV的表現甚至可以比CUDA更好，可見OpenCV內部的優化程度極為精良。

接下來，是此專題對Cpp版本加速的說明，如Fig. 4所示，此圖的Y軸代表加速的倍率，是指總時間加速的倍率，而X軸表示圖片的ID，對應image 1至image 5五張不同大小的input圖片，從Fig. 4可以得知，也即是輸入圖片為最大圖時可以達到18倍的加速，在輸入圖片為最小圖的狀況下，也至少有2倍的加速。

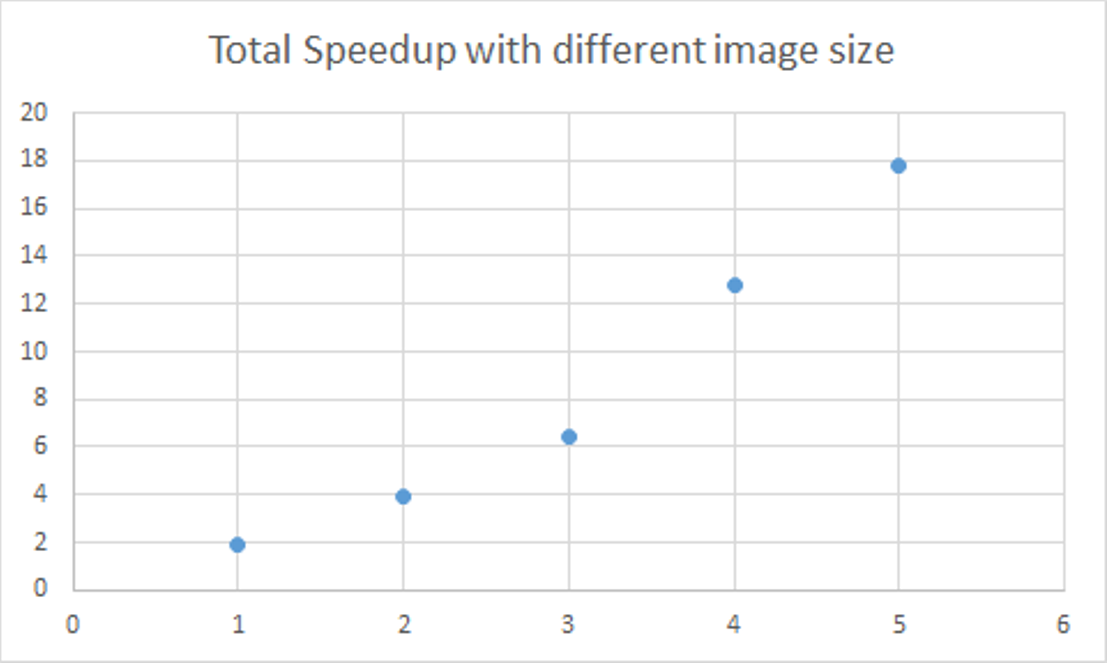


Figure 4: Cpp版本下的總時間加速成果

但是，我們對CUDA的加速程度只有十幾倍左右的狀況抱持著高度懷疑，起因於CUDA擁有非常多核心，此專題的作法是一個點以一個執行緒(thread)找，理論上來說，應要有上百倍甚至上千倍的加速成果，所以，我們將所有的函式都轉化成CUDA版本，也將所有函式都拆開來做測量，在此處要測量所有的函式時，要注意一點，在CUDA程式裡測時間，我們的作法是先把計時器放在程式裡面，接著需要使用一名為cudaDeviceSynchronize的函式 [11]，使cudaDeviceSynchronize在核心(Kernel)執行完的時候，先回到CPU按一下計時中的碼錶，按完碼錶後才會跑至下一個核心，否則核心啟動完後cudaDeviceSynchronize就會直接跳至下一個CPU，而會漏按一下計時中的碼錶。另外，可得知Canny邊緣偵測器會有atomicAdd執行，如前述所提，Canny邊緣偵測器也會掃描旁邊的點，故我們要保證跟旁邊的點互相做加減時，只會有一個函式更改記憶體裡的數值，故此時Canny邊緣偵測器與Hough 轉換都會發生此情況。因此，Canny邊緣偵測器的優化效果便明顯不如侵蝕與膨脹，如Fig. 5所示。

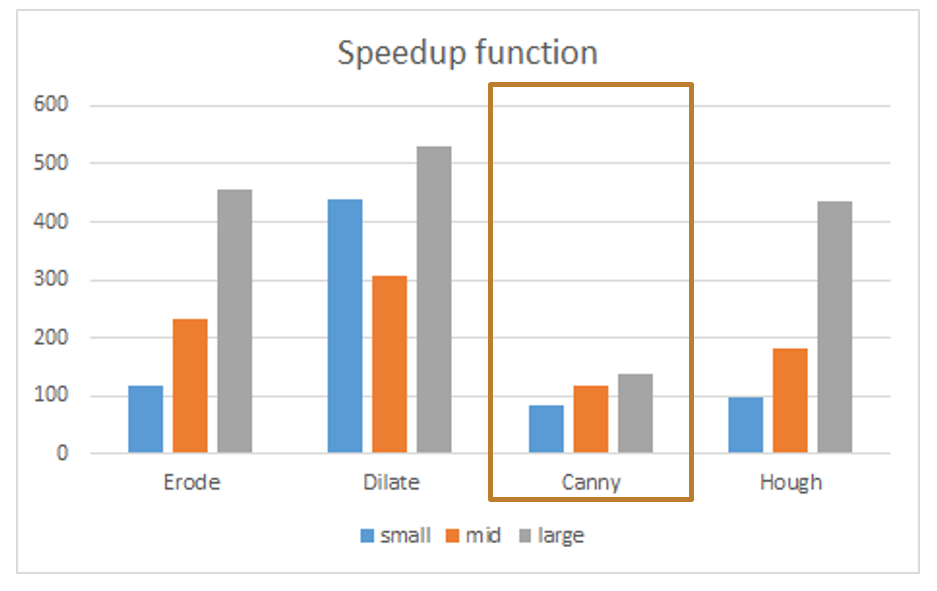


Figure 5: 運用CUDA達到的對於侵蝕、膨脹、Canny邊緣偵測器與Hough轉換的加速成果

在此專題的研究過程中，我們理解到這些使用到的演算法都是相似的，這些演算法都會對整張圖片做一次掃描，而每次掃描都會掃描一個小矩陣，跟摺積十分相似，故可以想像成這個動作是一個摺積，所以Canny邊緣偵測器與Hough 轉換的效果比較差，如前述所提，Hough 轉換是針對一個點畫一個圈，所以若輸入的圖片其尺寸非常大，針對一個點所畫的一個圈便非常不容易重複，故如Fig. 5所示，輸入越大的圖，Hough 轉換的效果越好，其成長的曲線看起來相當漂亮。

最後，我們要解釋的是Fig. 5與Fig. 4兩圖的資料為何會產生落差，可以看到如Fig. 5所示，侵蝕、膨脹與Hough 轉換在輸入大尺寸圖片時均可加速四至五百倍，即使是Canny邊緣偵測器在輸入大尺寸圖片時，也能加速一百二十倍左右，但我們最終實作的成果只有加速整體程式約二十倍，為了解釋這個非常不合理的狀況，便需要了解整個程式中哪個環節佔用最多時間來運算，如Fig. 6所示，佔用最多時間的是overhead的部分，包含memory allocation跟memory copy的部分。

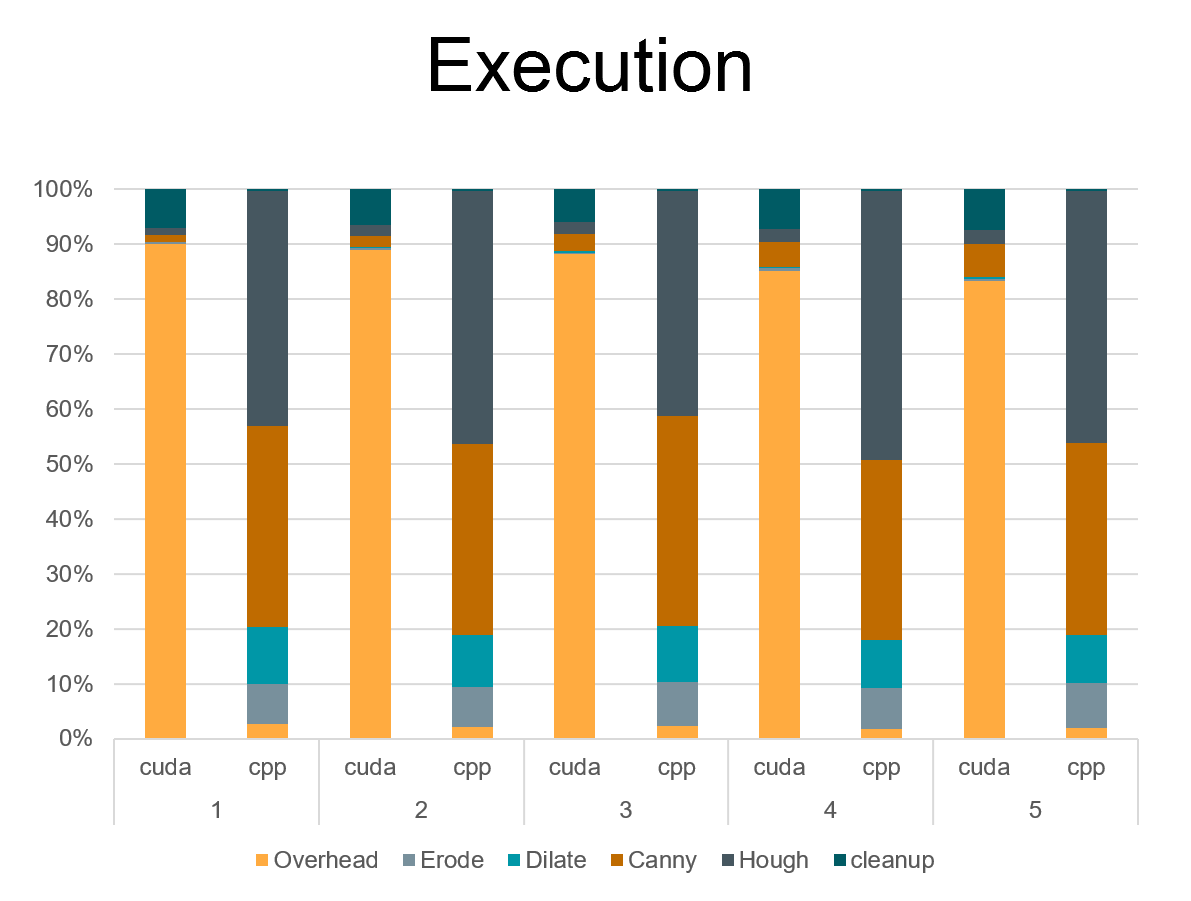


Figure 6: 各個部份的工作所消耗的時間

所以，CUDA的overhead部分包含將圖片重新載入、在CUDA程式中malloc一個記憶體空間與將圖片傳至device端上的耗費，在使用NVVP觀測整個圖後，如Fig. 7所示，可以觀察到在圖中間的部分，其前面有一malloc非常長，整整大於其他那些核心運行的部分非常多。

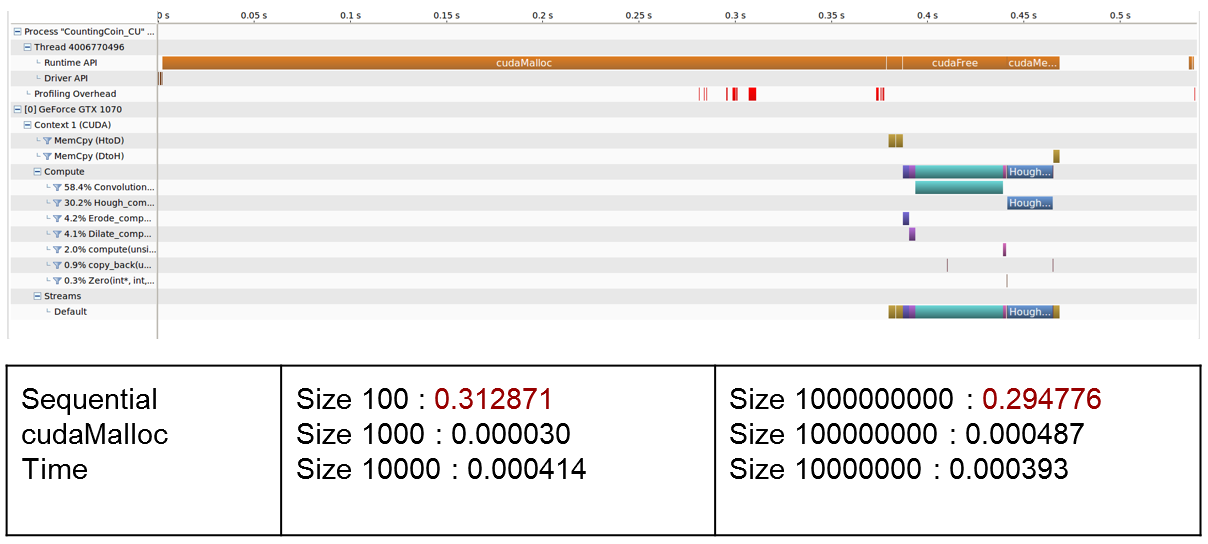


Figure 7: 初始化所耗費的時間

所以，透過一些小型實驗，連續malloc不同大小的空間，可以觀測到只有第一次的malloc會花到比其他次malloc多1000倍的時間，而其他次malloc所使用的時間都相差不遠，無論使用者所malloc的空間有多少，都得到這個相同的結果，經過文獻回顧，在CUDA的第一個API call(Application Programming Interface call)會負責初始化整個執行時期的環境時，故會隱含一個overhead，但因為此專題的主程式平行完後，其本身運行速度非常快，也即是程式運算完所耗費的時間很少，所以CUDA的第一個API call所負責初始化整個執行時期的環境所造成的overhead，便造成一個非常大的overhead，佔整個程式的overhead中非常大的比例，在證實此專題中這個實驗的部分，是將前述的步驟重複執行，如Fig. 8所示，顯而易見地，可以觀察到只有第一次的overhead較大，而在第二次與第三次執行同個程式時，只要耗費與第一次相比少非常多的時間就能完成。

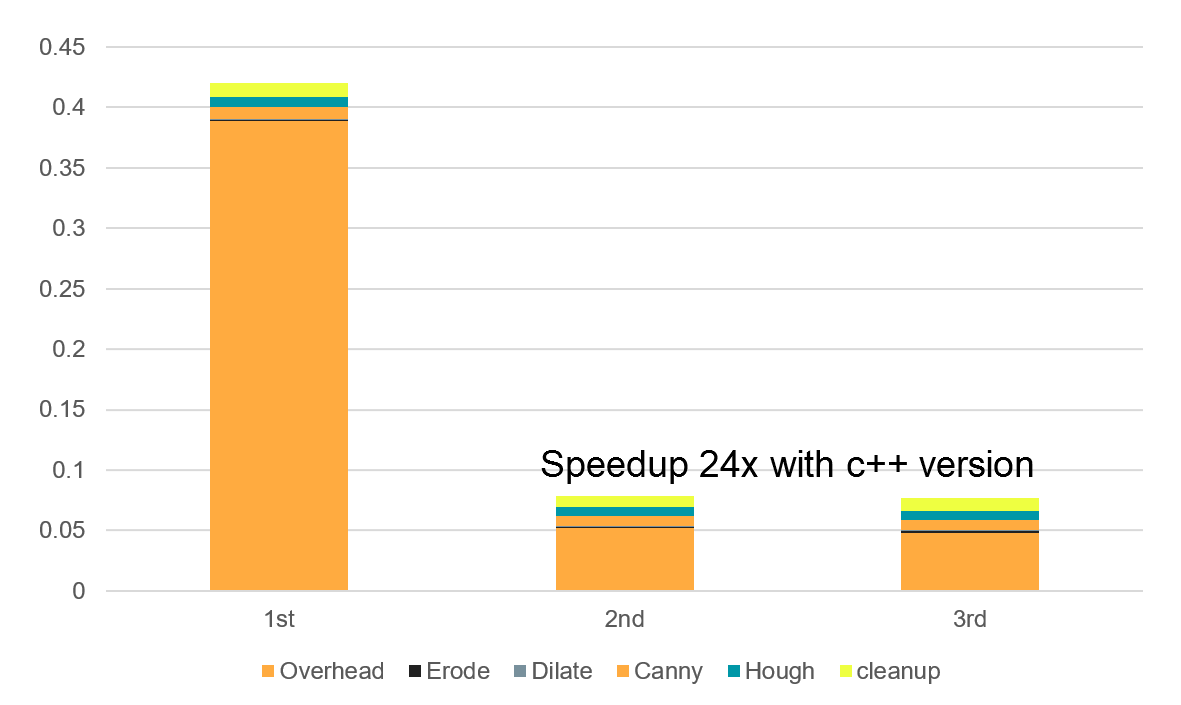


Figure 8: 為觀察overhead的成因所重複執行同一個程式得到的成果展示

在使用這筆測資運行第一次測試的結果與Cpp的版本比較下，能加速四倍，但如果同樣使用這筆測資運行第二次測試的結果跟Cpp的版本比較下，能加速到24倍，為了進一步優化程式，程式裡的各函式裡仍包含著一些隱藏的且暫時的被malloc出來的空間。

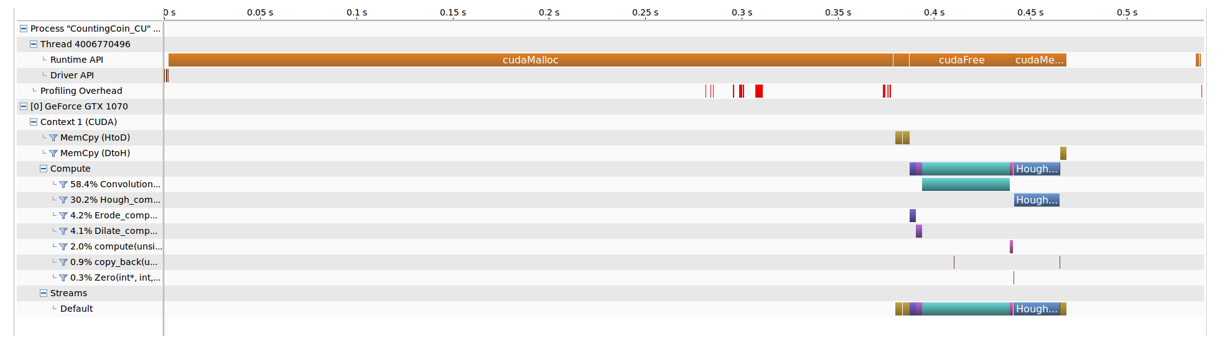


Figure 9: 對Pre-lloc進行最佳化所得到的結果

所以，希望將這些malloc的成本加到程式一開始執行的地方，再將這些malloc出來的空間傳給核心，進一步節省初始化所耗費的大量時間，如Fig.9所示，此為修改過後的成果。

再來，看到Fug. 10所示的這幾個函式，每個函式在做完pre-malloc之後，每個函式都有一部份的效能改進，而關於更進一步的優化，在malloc不同大小的空間的時候，全部都可以只耗費一樣的時間，故採用的方法是一開始就malloc一塊較大的空間，接著，再把這塊較大的空間切成我們所需要的大小，接下來是希望能組合不同核心，將這些函式通通合在一起，因為此專題的程式是一個核心一個核心l執行的，希望能將不同的核心合在一起，運用此法減少launch overhead，但最後發現嘗試的結果並不讓人非常滿意，由於演算法的內容都是執行摺積的動作，都有資料相依性的侷限，會影響到資料的執行緒，所以，在先執行的步驟其程式運算完之前，下一個步驟所執行的程式拿不到正確的資料。

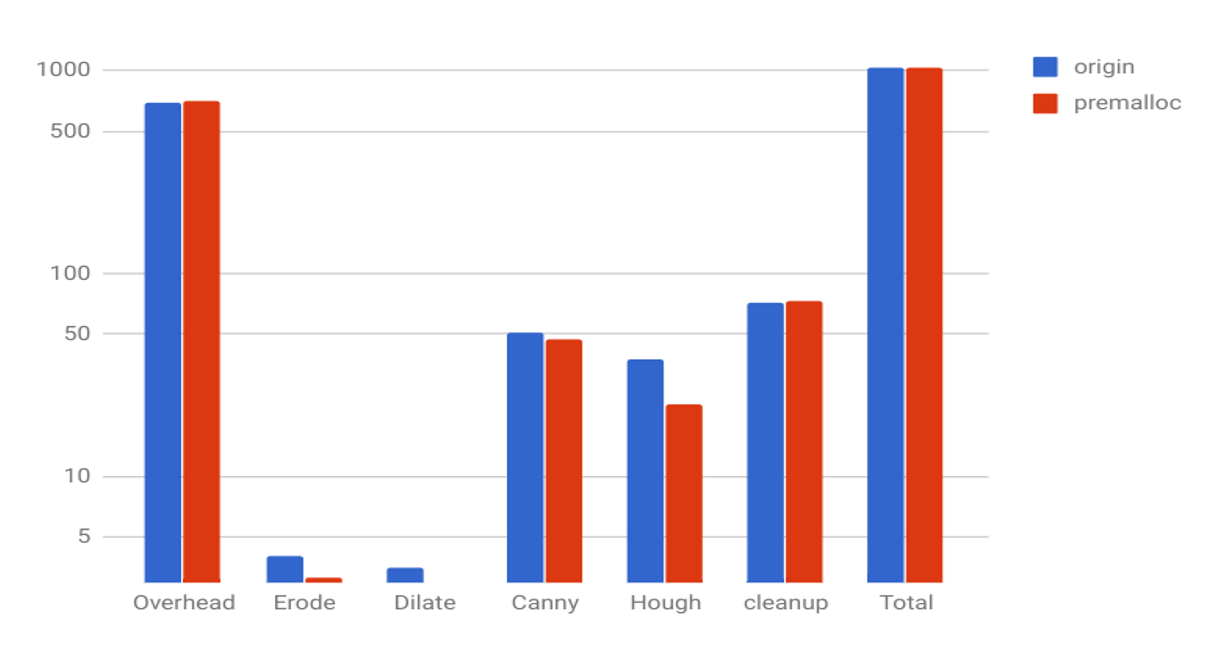


Figure 10: 在經過pre-malloc後主要被使用的

所有函式的效能比較

除了研究CUDA與其相關內容，此專題亦探討GPUCC在此專題題目上所能呈現的效能增進[12]，GPUCC，為Google於2016年在CGO所發表一名為gpucc: An Open-Source GPGPU Compiler，論文內容是GPUCC的最佳化(optimization)的細節與實驗相關數據，論文中所提及的開源編譯器Clang現已可編譯CUDA，此專題亦將相同的程式交給Clang編譯，以便得出結果與NVCC進行比較。

NVCC為Nvidia的閉源編譯器，Clang為GPUCC的開源編譯器，以編譯時間的結果來說，Clang耗費的時間為NVCC的一半，每次NVCC編譯CUDA檔案都耗時甚鉅，若使用Clang則編譯時間快速許多。

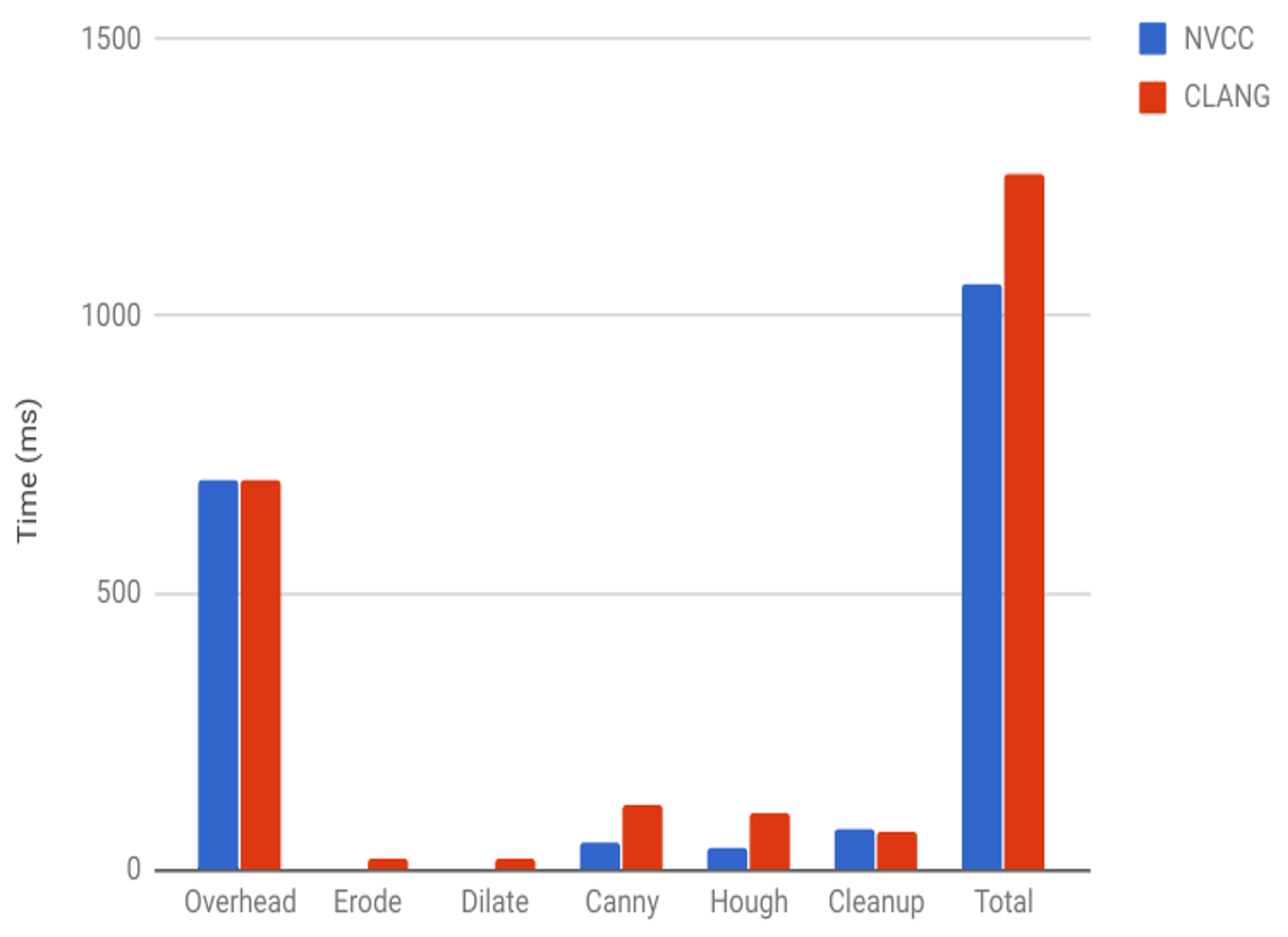


Figure 11: NVCC與Clang的編譯時間比較圖

但是，在程式執行期時間的表現上，Clang的表現卻比較差，但是，Clang與NVCC的差距並不會只有如此的小，實際上，Clang在執行期時間上的表現與NVCC相比，NVCC的表現大勝Clang，如Fig. 12所示，此為每個函式的執行時間，在對Clang來說最好的狀況下，Clang的表現仍比NVCC慢了兩倍之多，所以，在此專題所採用的案例上，Clang的表現相當不好，故前述的數據與成果都是採用NVCC產生的實驗結果。

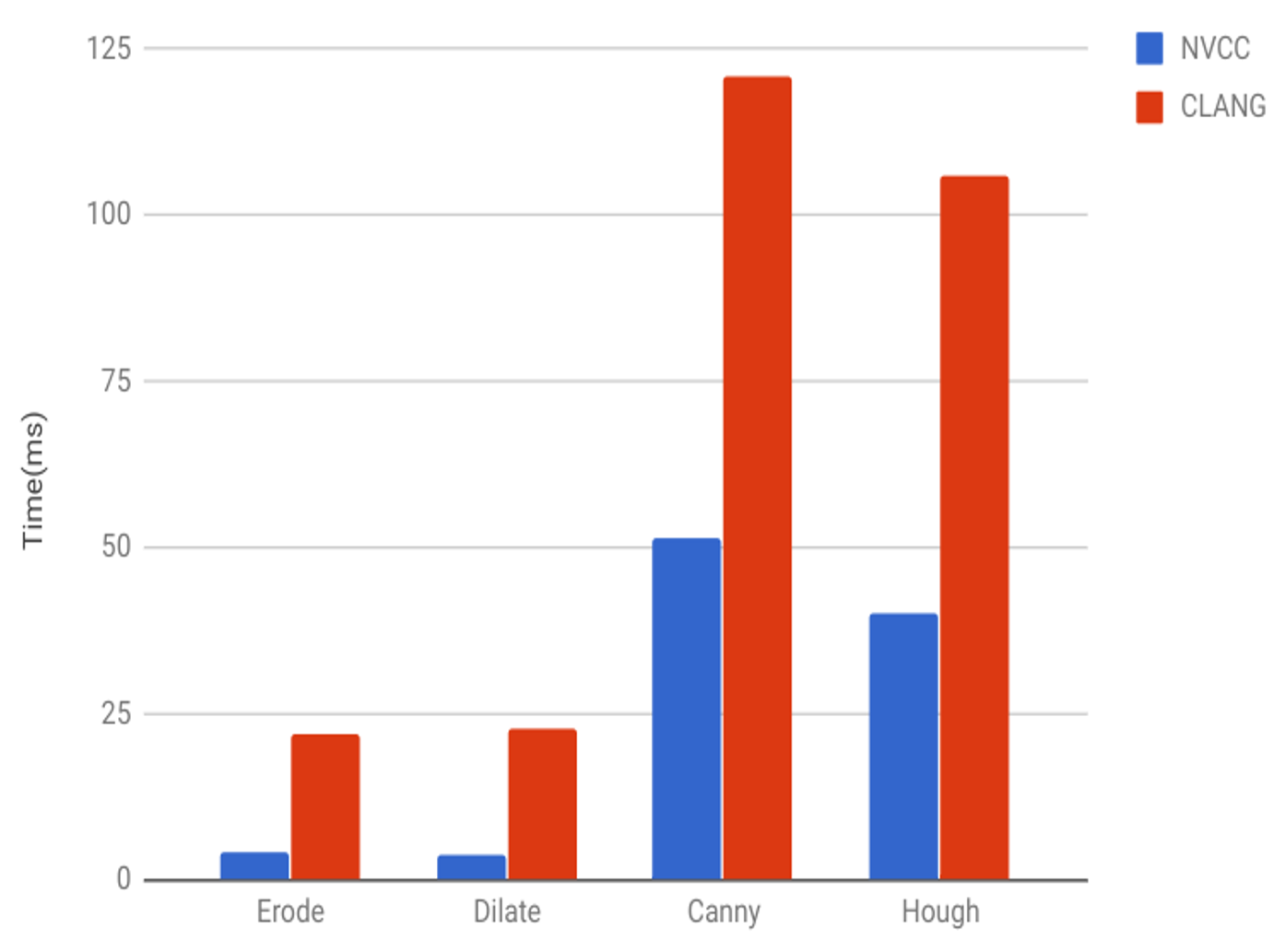


Figure 12: NVCC與Clang的效能表現比較圖

接著，我們更進一步探討GPUCC效能方面的優缺點，雖然在此專題的題目與程式上的表現，NVCC都大勝Clang，但在gpucc: An Open-Source GPGPU Compiler這篇論文中，其數據成果是跟NVCC有並駕齊驅的表現(performance)，在GPUCC的論文中有特別指出，Clang與NVCC在整體來說能並駕齊驅，只限於特定案例，在此專案的案例中Clang的表現均不好，但用於其他案例，Clang可能表現奇佳。所以，未來在其他研究上，除了以NVCC來進行編譯，也建議使用Clang來編譯相同的程式，可能可以得到非常好的編譯時間跟成果，最後，我們要特別強調GPUCC是開源的，可以很清楚地了解GPUCC是如何進行最佳化且較為容易找出裡面有哪些程式上的錯誤。  
  
**5 RELATED WORK**

GPUCC的製作團隊製作GPUCC起因於: 在GPUCC誕生以前，世界是處於一個缺乏適用於CUDA編譯器與高效能編譯器(High Performance Compiler)的環境下，再者，針對binary dependencies、performance tuning、modern language features與bug turnaround times等問題做進一步地改進，接下來介紹上述內容產生的問題。

首先是binary dependencies方面的問題，允許供應商所提供的二進位檔案進入資料中心(data center)將導致大量的二進位檔案依賴關係(binary dependencies)，在更新核心庫(core library)時，所有相關供應商的二進位檔案都必須同時被更新，而當相關供應商歇業時，上述所有相關供應商的二進位檔案都必須同時被更新的這個要求，基本上是不可能達到的。第二個要討論的是performance tuning的問題，編譯器方面其目前的發展，是正在調整成一套通用的標準，如: SPEC，但這些測量基準可能無法精準代表並量測出資料中心的重要工作量，若能做到調節編譯器使其針對特定的工作負荷量產生啟發式的優化，必定能帶來更好的效能表現，尤其在資料中心與超級電腦上進行此項目的優化時，百分比等級的優化足以帶來實際龐大數量的效能改進。

接下來要討論的是Modern language features這個問題，若程式語言像Cpp這個程式語言一樣地發展，其程式語言的發展組織可能支援更多的現代語言標準或不相容的環境，這些不相容的環境都可能與專有編譯器所支援的語言功能子集產生衝突，上述問題使得複雜的source sandboxing有其必要性。最後的問題是bug turnaround times，廠商一定是同時服務多個客戶且其發布過程極其複雜，相應地，在整個工具鏈(toolchain)中找出與修復程式碼的錯誤通常需要幾週至幾個月的時間，使用開源編譯器可以大量縮短錯誤周轉的時間。

為解決上述問題，GPUCC提出以下解決方案，Straight-line scalar optimizations，他們的重點特點是簡化了執行整數與指標運算的一部份冗於表達式，例如: 在b\*n已被算出的情況下，GPUCC能夠將(b+1)\*n改寫為b\*n+n，上述的優化對經常存取陣列的高效能編譯器所編譯的程式特別有用，例如: 矩陣乘法、內積(dot product)與反向傳播(back propagation)。另一個方法為Loop unrolling and function inlining，由於單指令流多執行緒流與無亂序等之運算過程消耗以及記憶體的calling convention，一般GPU上的jump與函數呼叫的耗費比CPU上來得更多，Loop unrolling and function inlining便減少了GPU上的jump與函數呼叫，同時提高Constant Propagation與Scalar Replacement of Aggregates的機率，其中Scalar Replacement of Aggregates更明顯減少了存取堆疊變數的成本。最後要介紹的加速方法是Inferring memory spaces，GPUCC以定點資料流分析(a fixed-point data-flow analysis)的方式實踐了記憶體儲存空間的推定，其傳遞給各個記憶體儲存空間一個函數，此法首先假設所有的derived pointers透過指標的運算傳遞給各個記憶體儲存空間，接著，同時不斷重複著復原所有因假設失敗而無法被指派給指定的記憶體儲存空間的所有指標們，並給他們新的假設令他們得以被指定給記憶體儲存空間，直到已確定使用的記憶體儲存空間與假設要用的記憶體儲存空間包含了所有指標所要用的記憶體儲存空間為止，此法改善了仰賴自身的指標感應變數們的狀況。

最後，由上述內容可知GPUCC是個功能沒有缺漏、開源且性能良好的CUDA編譯器，其貢獻在於啟用可被重視的GPU編譯器和架構研究，即使在受限制的環境中也能輕鬆部署GPU。

**7 CONCLUSIONS**  
此專題證明了這些獨立的影像處理演算法，可以透過CUDA在效能上得到很優異的提升，在整體時間上，此專題的加速成果沒有到想像中那麼地明顯，起因於此專題採用的方法是載入比較大的圖片，也就是載入的資料量是較為龐大的，即使在程式的編寫上已盡力且沒有任何問題，但沒有足夠的運算量來負擔那些overhead造成的不良影響，另外，此專題沒有在令核心同時執行多個指令上多做著墨，導致整體時間的加速沒有想像中來得明顯的，在研究此專案的過程中學習到很多CUDA與其相關更底層的硬體技術與知識，希望在未來能在以GPGPU加速影像處理演算法上做出更多突破。

此專題之程式碼:

**REFERENCES**

|  |  |
| --- | --- |
| [1] | PARK, In Kyu, et al. Design and performance  evaluation of image processing algorithms on GPUs.  IEEE Transactions on Parallel and Distributed  Systems, 2011, 22.1: 91-104. |
| [2] | THALAU, Peter, et al. The magnetic compass  mechanisms of birds and rodents are based on  different physical principles. Journal of the  royal Society Interface, 2006, 3.9: 583-587. |
| [3] | OTSU, Nobuyuki. A threshold selection method  from gray-level histograms. *IEEE transactions on*  *systems, man, and cybernetics*, 1979, 9.1: 62-66. |
| [4] | R. C. Gonzalez and R. E. Woods, Digital Image  Processing, 3rd Edition Prentice Hall, 2007. |
| [5] | CANNY, John. A computational approach to edge  detection. *IEEE Transactions on pattern analysis*  *and machine intelligence*, 1986, 6: 679-698. |
| [6] | DUDA, Richard O.; HART, Peter E. Use of the Hough  transformation to detect lines and curves in pictures.  Communications of the ACM, 1972, 15.1: 11-15. |
| [7] | BRACEWELL, Ronald Newbold; BRACEWELL,  Ronald N. *The Fourier transform and its*  *applications*. New York: McGraw-Hill, 1986. |
| [8] | BUTENHOF, David R. *Programming with POSIX*  *threads*. Addison-Wesley Professional, 1997. |
| [9] | BARNEY, Blaise, et al. Introduction to parallel  computing. *Lawrence Livermore National*  *Laboratory*, 2010, 6.13: 10. |
| [10] | BRADSKI, Gary; KAEHLER, Adrian. *Learning*  *OpenCV: Computer vision with the OpenCV library*.  "O'Reilly Media, Inc.", 2008. |
| [11] | JONES, Stephen. Introduction to dynamic parallelism.  In: GPU Technology Conference Presentation S. 2012.  p. 2012. |
| [12] | JingyueWu, Artem Belevich, Eli Bendersky, Mark  Heffernan, Chris Leary, Jacques Pienaar, Bjarke  Roune, Rob Springer, Xuetian Weng, and Robert  Hundt. gpucc: an open-source GPGPU compiler. In  Proceedings of the 2016 International Symposium on  Code Generation and Optimization, pages 105–116.  ACM, 2016. |