GPU 上的平行距離轉換演算法

Efficient Parallel Distance Transform in GPU

朱炫吉 徐熊健
Xuan-Ji Zhu **Shyong Jian Shyu**

銘傳大學資訊工程學系
Department of Computer Science and Information Engineering,
Ming Chuan University
E-mail: sjshyu@mail.mcu.edu.tw, georgetc0831@gmail.com

摘要

本論文提出一個適用於繪圖處理器（graphic processing unit，簡稱 GPU）上，解決二元影像中距離轉換 (distance transformation) 問題的平行演算法；此演算法注著重於計算歐式距離轉換，對 $h \times w$ 的二元影像 I 計算各背景像素點至所有物件點 n 的最短歐式距離，並分為二個階段，第一階段利用 Hillis 和 Steele [5] 提出的前綴平行掃描演算法的性質對影像中的每一橫條做平行雙向掃描，使橫條上的每背景像素點都記錄著在同一橫條上離自己最近物件點的距離；第二階段為對每一直條向上掃描將結果存於 U 中，再向下掃描將結果存於 D 中；接著對 U 向下尋找最近的物件點的歐式距離，對 D 向上尋找最近的物件點的歐式距離，並比較 U 和 D 的大小取其最小歐式距離。本論文會將所提出的方法與 CPU 窮舉法、GPU 窮舉法和 CPU 雙向掃描演算法做比較，並分析在 CUDA 的架構中 block 數和 thread 數的分配以達到較高的效率。

關鍵詞：CUDA、平行演算法、影像處理、歐式距離轉換

Abstract

An efficient parallel algorithm designed for graphic processing unit (GPU) is proposed in this paper to resolve the distance transform problem in a binary image. Given an $h \times w$ binary image I consisting of n objects, the shortest Euclidian distance from each pixel to all these n objects are computed. Our parallel shortest distance scan (PSDS) algorithm extends the ideas of the conventional parallel prefix scan algorithm. It scans rows in parallel from left to right and from right to left to compute the shortest distance from each pixel to the nearest object in the same row. Then, it scans columns in parallel bottom-up and top-down to compute the perpendicular distance from each pixel to the object in the same column. Then, it searches columns in parallel bottom-up and top-down to determine the shortest distance from each pixel to the nearest object in the whole image. The performances of the PSDS algorithm running in GPU is compared against that running in CPU, an enumerative algorithm running in GPU and CPU. The performances of PSDS in GPU using CUDA via different combinations of numbers of threads and blocks are also discussed.

Keywords: CUDA, parallel algorithm, Euclidean distance transform, image processing

一、 前言

1964 年由 Seymour Cray 公司推出了 CDC600 後電腦硬體就進入了平行化高速計算時代，在這發展期間經歷了超級電腦、叢集式電腦的演變，發展至今繪圖處理器 (GPU) 的出現更加大了平行計算的運用空間。

CUDA (compute unified device architecture) 由 NVIDIA 提出可在繪圖處理器進行平行運算的整合技術，讓程式設計者可以利用 CUDA 的 C 語言擴充直接使用 C 語言撰寫程式，並利用 CPU 和 GPU 的「協同處理」將資料及程式流程工作分配到多個執行緒 (threads) 上，由繪圖處理器中的計算核心 (cores) 去執行。目前市面上的中階繪圖處理器在價格上並不昂貴，同時也能提供不錯的運算效能，也因此降低了程式設計者要做平行計算時的花費；目前繪圖處理器的普及，使的越來越多的研究人員討論且投入相關問題。

距離轉換問題在影像處理 (image processing) 領域中，扮演著前導基礎運算的角色；以距離轉換所衍生出的應用領域非常廣泛，包括：清潔機器人的全域覆蓋路徑規劃 (complete coverage path planning) [1]；或光跡追

蹤 (ray-tracing) [2] 等應用。

本論文的後續架構：第二節將介紹影像處理中距離轉換的定義、應用與文獻中提到的相關演算法；第三節中為本研究所提出的平行距離轉換演算法；第四節則呈現該演算法在繪圖處理器上的實驗效能；最後在第五節中做一個結論。

二、 問題定義與文獻探討

本節中將介紹關於距離轉換的定義（二.1 節）與相關的文獻探討（二.2 節）。

二.1. 距離轉換的定義

令 I 為一張二元影像其邊長為 $h \times w$ （內部像素質為黑或白，黑為 1 白為 0），其中黑點代表物件點而白點代表背景像素點，黑點構成的集合為 O 而白點構成的集合為 B ；由上所敘述定義可得到 $O \cup B = I$ 且 $O \cap B = \varnothing$。距離轉換是計算各像素點至所有物件點的最短距離值；令 x、y 為 I 中的點其 $x \in B$ 且 $y \in O$ ，並將 $dt(x)$ 定義為對點 x 做完距離轉換的結果且 $\delta(x, y)$ 定義為 x 與 y 之間的距離，由此我們可以得到：

$$dt(x) = \min\{\delta(x, y) \mid y \in O\}；$$

計算距離轉換的方式有分兩類，第一類歐式距離 (Euclidean distance)，歐式距離是兩點間實際的距離，若 $x \in B$ 且 $y \in O$，$x = (a, b)$ 與 $y = (c, d)$ 則歐式距離為：

$$E(x, y) = \sqrt{(a-c)^2 + (b-d)^2}；$$

歐式距離轉換是計算各背景像素點至所有物件點的最短歐式距離，其公式為：

$$dt(x) = \min\{E(x, y) \mid y \in O\}；$$

因此較為複雜，典型作法為窮舉法 (enumeration)，每一個背景像素點都要對所有的物件點做距離計算，並取最小者做為背景點的值。

第二類為 chamfer 距離，chamfer 距離可當成影像 I 中 x 到 y 所需的移動步數，令 r 為水平和垂直方向移動的步數（每移動到相鄰的點一次，及計一次），令 s 為斜角方向移動的步數（左上, 右上, 左下, 右下），在令 a 為水平垂直方向每次移動的權重值，b 為斜角方每次移動的權重值的權重值；而 ρ 為 x 到 y 的某條移動路徑，此路徑長度 $l_{a,b}(\rho)$ 為：

$$l_{a,b}(\rho) = ra + sb$$

而 x 到 y 的 chamfer a-b 距離轉換，則為 x 到 y 中所有路徑中的最短的一條，其公式為：

$$\delta(x, y) = \min_{\rho}\{l_{a,b}(\rho)\}；$$

在 chamfer a-b（縮寫：$C_{a\text{-}b}$）轉換中 a, b 大都為正整數，目前(a, b)常用的值為 $C_{1\text{-}\infty} = (1, \infty)$ 稱為街區距離 (city-block, CB)、$C_{1\text{-}1} = (1, 1)$ 為棋盤格距離 (chessboard)、$C_{2\text{-}3} = (2, 3)$、$C_{3\text{-}4} = (3, 4)$、$C_{5\text{-}7} = (5, 7)$等。相較於街區距離和棋盤格距離，$C_{2\text{-}3} = (2, 3)$、$C_{3\text{-}4} = (3, 4)$ 和 $C_{5\text{-}7} = (5, 7)$ 這三種參數的值更佳的逼近歐式距離，舉例來說在歐式距離中斜角方向的鄰近點其距離值為 $\sqrt{2}$=1.414 比值為 1:1.414，而 $C_{2\text{-}3}$、$C_{3\text{-}4}$、$C_{5\text{-}7}$ 這三種比值分別為 1:1.5、1:1.333、1:1.4，因此在影像上執行距離轉換後的結果會在越後者越接近。

本論文著重於歐式距離轉換 (Euclidean distance transform) —計算各像素點至所有物件點的最短歐式距離值—平行演算法的設計與實作。

二.2. 距離轉換演算法之相關文獻

在 chamfer 距離轉換演算法中，由 Rosenfeld 與 Pfaltz [3] 提出的雙向掃描演算法，有極佳的時間複雜度 $O(n^2)$。Rosenfeld 與 Pfaltz 於 1968 年提出的雙向掃描演算法；雙向掃描演算法分別是前向(forward) 和後向(backward) 掃描。執行演算法前，會先將圖片中的所有物件點的距離值設為 0，而所有背景像素點的距離值設為 ∞。前向掃描是從影像頂端由左至右一列一列的由上至下循序掃描（由左上至右下）。在掃描的過程中，對每一個點 $I(i,j)$ 給予暫時的距離值 $d_1(I(i,j))$，其計算方式為：

$$d_1(I(i,j)) = \min\begin{Bmatrix} d_2(I(i,j)) \\ d_2(I(i-1,j-1)) + a \\ d_2(I(i-1,j)) + b \\ d_2(I(i-1,j+1)) + a \\ d_2(I(i,j-1)) + b \end{Bmatrix}；$$

亦即 $d_1(I(i, j))$ 為其上自己與相鄰參考點 $(I(i-1, j-1), I(i-1, j), I(i-1, j+1), I(i, j-1))$ 加移動長度權重值（a 和 b 即為 chamfer a-b 距離的參數）後的最小值。

後向掃描與前向掃描類似，是從影像底端由右至左一列一列的由下至上循序掃描（由右下至左上）；掃描過程中亦會對每一個點 $I(i,j)$ 求其暫時距離值 $d_2(I(i,j))$，其計算公式為：

$$d_2(I(i,j)) = \min\begin{Bmatrix} d_2(I(i,j)) \\ d_2(I(i+1,j+1)) + a \\ d_2(I(i+1,j)) + b \\ d_2(I(i+1,j-1)) + a \\ d_2(I(i,j+1)) + b \end{Bmatrix}；$$

亦即 $d_2(I(i, j))$ 為其自己與相鄰參考點 $(I(i+1, j+1)、I(i+1, j)、I(i+1, j-1)、I(i, j+1))$ 加上移動長度權重值後的最小值。圖 1 表現雙向掃描中，計算前向掃描 $d_1(I(i, j))$ 和後向掃描

$d_2(I(i,j))$ 時分別參考的相鄰點。

$I(i\text{-}1, j\text{-}1)$	$I(i\text{-}1, j)$	$I(i\text{-}1, j\text{+}1)$
$I(i, j\text{-}1)$	$I(i, j)$	

(a) 前向掃描

	$I(i, j)$	$I(i, j\text{+}1)$
$I(i\text{+}1, j\text{-}1)$	$I(i\text{+}1, j)$	$I(i\text{+}1, j\text{+}1)$

(b) 後向掃描

圖 1 雙向掃描所參考的相鄰點

2006 年由 Shyu 、 Choa 和 Chia [4] 將雙向掃描演算法平行化並實作於電腦叢集上，利用個人電腦當作處理單元，並使用網路作為訊息交換的媒介，另外並設置一個共享儲存媒體來存放原始檔案與計算結果。如圖 2。

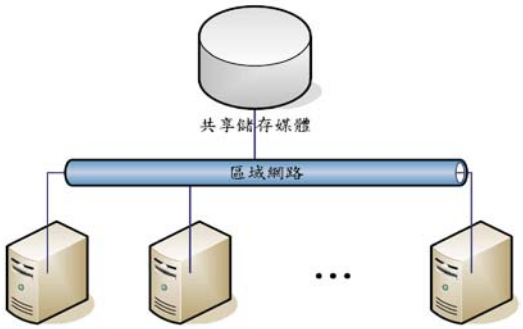
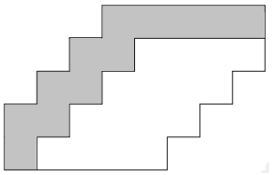


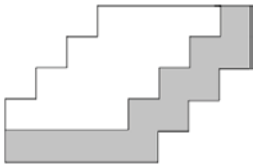
圖 2 個人電腦叢集系統架構

將雙向掃描平行化時需要將影像切割為長條狀的子影像，並由各台電腦去計算各自負責的數條子影像；由於雙向掃描在計算時需要計算到相鄰的四個參考點，如圖 3 (a) 與 (b) 所示，因此資料據有繼承關係，必需將計算完的訊息傳輸給下一台電腦。

傳輸的總資料量與長條狀的子影像內的切割區塊大小成反比；切割區塊越大，需要傳送的資料量會較少；切割區塊越小，需要傳送的資料量就會越大；此演算法必須在切割區塊和傳送的資料量之間取得一個平衡點，使得傳輸時間與運算時間在平行運算時的總和得以最小。



(a) 前向掃瞄



(b) 後向掃瞄

圖 3 類平行四邊形的分隔區與其前置相鄰參考點

圖 3 白色部分為分隔區的形狀，稱為類平行四邊形；灰色部份為額外需要的相鄰參考點，稱前置相鄰參考點。而區塊計算完後與下個分隔區相鄰的邊界點則成為下個分隔區所需的前置相鄰參考點。

假設由 3 台電腦所組成的電腦叢集，分別為 P_1、P_2 和 P_3，α 與 β 分別表示類平行四邊形分隔區的長與寬，因此會將一張 $h \times w$ 圖片切割為 $(h/\alpha)(w/\beta + 1)$ 個分隔區，如圖 4 中所表示，為方便說明將每個區塊以不同的數字表示；以前向掃描為例，P_1 計算分隔區 1（類三角形）和分隔區 2（類平行四邊形）時，其他處理器是處於等待狀態的（尚未取得前置相鄰參考點的資訊）；當 P_1 計算分隔區 3 時，P_2 才開始計算分隔區 6；而 P_1 計算分隔區 4 時，P_2 可計算分隔區 7；當 P_1 計算分隔區 5 時，P_2 可計算分隔區 8，而 P_3 則可計算分隔區 11；依此類推。

在十六台個人電腦構成的電腦叢集中執行此平行距離轉換演算法，就 3840×3840 的二元影像而言，該平行演算法相對循序演算法之加速比為 8.18；若考慮資料讀入的前置時間更可高達 13.44。

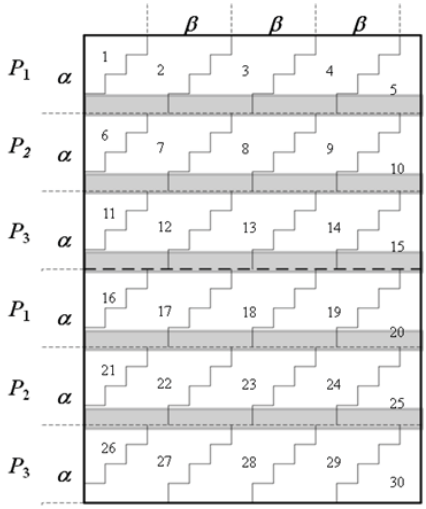


圖 4 執行平行雙向掃描時電腦執行的區塊和切割的區塊大小

三、 本研究之平行距離轉換演算法

本論文所提出的平行距離轉換演算法基本上以計算實際的歐式距離為主，此演算法適用於繪圖處理器的平行演算法。三.1 節介紹繪圖處理器硬體架構；三.2 節介紹 CUDA 的軟體架構；三.3 節介紹 SIMT 架構；三.4 節介紹適用於繪圖處理器的窮舉法 (enumeration) 距離轉換；三.5 節介紹由本論文提出的平行距離轉換演算法。

三.1. 繪圖處理器硬體架構

本論文所使用的繪圖處理器為 NVDIA 推出的 GTX560ti 其架構如圖 5，內部有 8 個 SM (streaming multiprocessor)，每個 SM 內有 48 個 core、16 個存取單元 (load / store units) 和 4 個 SFUs (special function units)，共有 384 個 core 、128 個存取單元和 32 個 SFUs。

在 GTX560ti 內部含有 1024 MB 的 global memory 可讓 8 個 SM 共同存取，而每個 SM 內部包含 48 KB 的 share memory 和 32768 × 32 bit 暫存器供給 SM 本身存取。

三.2. CUDA 的軟體架構

在 CUDA 的架構下，一個程式分 host 跟 device 兩個部分；host 是指在 CPU 上執行的部分，而 device 是指在 GPU 上執行的部分，我們將會運行在 GPU 上的 CUDA 函式稱之為 kernel，也就是在一個完整的 CUDA 程式中，資料在 GPU 端平行處理的部分。

CUDA 的平行方式是將計算工作轉為大量的執行緒 (thread)，讓繪圖處理器動態的調度執行；CUDA 的軟體架構中最小單位是執行緒，而數個執行緒組成一個執行緒區塊 (block)，數個執行緒區塊組成一個 grid（如圖 6）；一個 block 最多可由 1024 個 thread 組成，一個 grid 最多可由 65535 個 block 組成；在同個 block 內部的 thread 可以被同步執行，但 block 之間是無法溝通，且執行順序是無法掌握的。

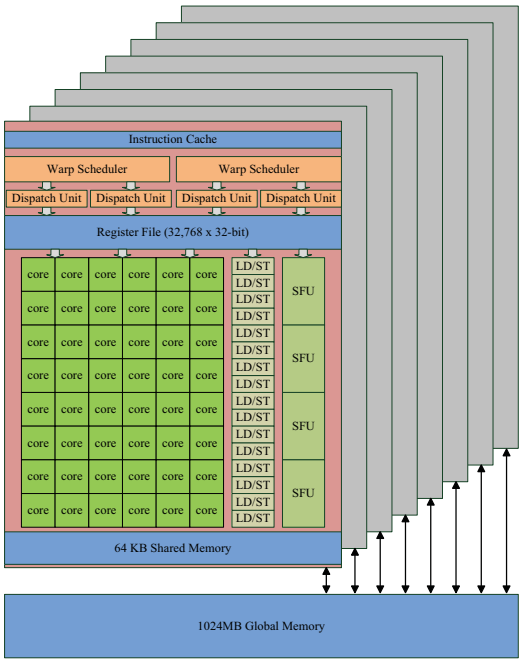


圖 5 GTX560ti 硬體架構

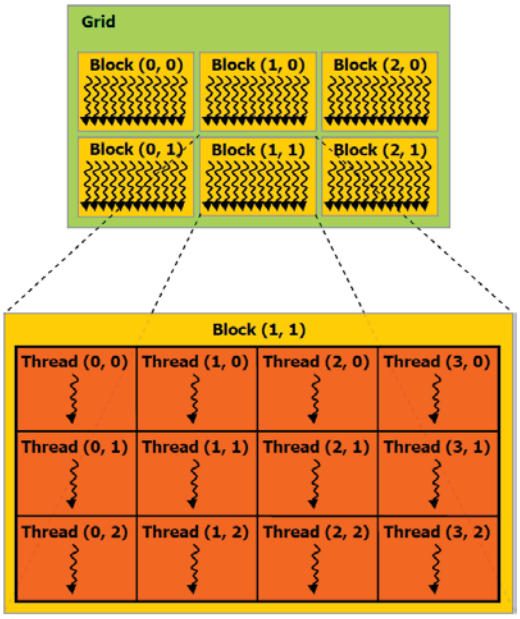


圖 6 CUDA軟體架構

繪圖處理器內部的記憶體是屬於多層級的架構，因此 CUDA 內的記憶體模型也被如此定義；圖 7 為 CUDA 記憶體模型架構，grid 內有三種記憶體分別是 global、constant、texture memory，這些記憶體都可以跟 host 端互傳資料；thread 能對 global memory 做存取另外兩種只能讀取；每個 block 都各自有一塊獨立 shared memory 的可供 thread 做存取，在block 內每個 thread 都分配到一些 registers 跟 local memory。

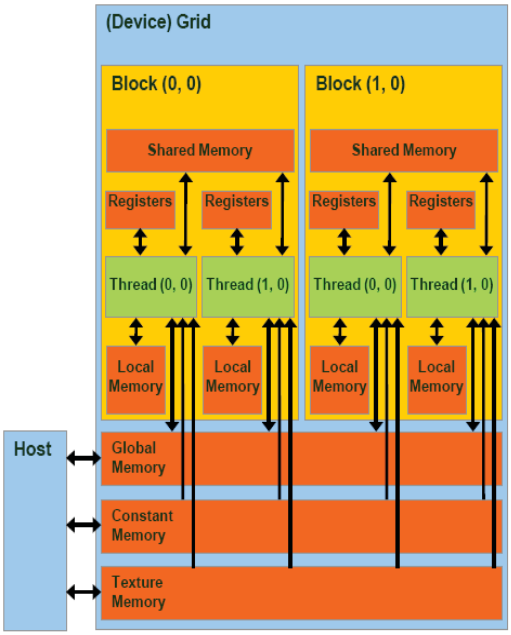


圖 7 CUDA記憶體架構

三.3. SIMT 架構

單一指令多執行緒（single-instruction, multiple-thread, 簡稱 SIMT），SM 在管理、執行和排程 thread 是以一組 32 個連續的 thread 為單位，稱之為 warp，在運算時同一個 block 內的 thread 會被分配到一個同一個 SM 上，而 SM 會同步的執行一個 warp 內的一個指令，因此 warp 是最小的執行單位；依照繪圖處理器的不同一個 SM 一次能執行的 warp 數也會不相同，以 GTX560ti 為例，一個 SM 內含有 2 個 warp scheduler，因此在執行時可以一次處理兩個 warp。

三.4. 適用於繪圖處理器的窮舉法距離轉換

窮舉法為典型的演算法，計算各背景像素點至所有的物件點的歐式距離，並取其最小值。演算法流程簡述如下：

GPU 窮舉距離轉換演算法

輸入：一張 $h \times w$ 的二元影像 I

輸出：該二元影像 I 相對應之歐氏距離地圖 (distance map)

1. host 端會先搜尋 I 中的所有物件點的座標並存於一個陣列 L 中；
2. 將二元影像 I 和陣列 L 傳入 device 中且將陣列 L 綁定 texture memory。
3. 對二元影像 I 上的每點去計算 L 內所有點的歐式距離，並取最小值，產生一張距離地圖如下：

3.1 for ($x = Bid$；$x < h$；x += block 總數)

3.2 for ($y = Tid$；$y < w$；y += 一個 block 的 thread 數)

3.3 for ($p = 0$ to 表單 L 長度)

$$I(x, y) = dt(x, y)$$

4. 距離轉換的結果傳回 Host 端。

3. 內的演算法利用 block 的編號 (*Bid*) 綁定影像中的橫條，使的同個 block 內的 thread 可被綁定在同一橫條上，且利用 thread 的編號 (*Tid*) 去綁定 在同一橫條中，thread 負責處理的陣列單元； $dt(x, y)$ 定義為計算點 (x, y) 對 L 內記入的所有物件點的歐式距離並取最小值，公式如下：

$$dt(x, y) = \min\{ E((x, y), (L_x[p], L_y[p])) | (L_x[p], L_y[p]) \in L \}。$$

此演算法因為是計算歐式距離，所以精確度高，但由於是以物件點為主的計算方式，因此物件點的多寡會直接的影響到計算時間。

三.5. 本論文提出的平行距離轉換演算法

本論文所提出的演算法，是以 Hillis 和 Steele [5] 提出的前綴平行掃描演算法為基礎，先對每條橫條計算一維距離轉換，再做縱向的一維距離轉換，並利用啟發式搜尋法的概念，將已經求得的物件點距離當成限制範圍，排除範圍外的物件點，透過不斷更新的限制範圍求得最近物件點距離，其演算法流程簡述如下：

輸入：一張 $h \times w$ 的二元影像 I。
輸出：該二元影像 I 相對應之歐氏距離地圖。
步驟一：host 端會先將二元影像 I 所有物件點的設為 0、背景像素點設為 ∞；
步驟二：將影像 I 傳入 device 中;
步驟三：對影像 I 做向左前綴平行掃描、向右前綴平行掃描；
步驟四：對影像 I 做向上前綴平行掃描，將結果存於陣列 U 中，接著對影像 I 做向下前綴平行掃描將結果存於陣列 D 中；
步驟五：對陣列 U 向下尋找最接近的物件點並計算其距離存於 U 內，接著對陣列 D 向上尋找最接近的物件點並計算其距離；
步驟六：比較 U 和 D 並取其最短距離存回 I 中；
步驟七：將以成為歐氏距離地圖的 I 回傳給 Host，並將 I 輸出；

三.5.1. 傳遞距離的平行掃描演算法

一個簡單的前綴平行掃描演算法是由 Hillis 和 Steele [5] 在 1986 年提出，其演算法執行步驟如下：

前綴平行掃描演算法

輸入：長度為 k 的一維整數陣列 a 和運算子 $\oplus$

輸出：長度為 k 的一維陣列 $b = \text{scan}(a, \oplus)$

1. for $d = 1$ to $\log_2 n$ do
1.1 forall k in parallel do
if $k_i \geq 2^d$ then $b_i = a_{i-2^{d-1}} \oplus a_i$；
2. output b； // $b = \text{scan}(a, \oplus)$

輸入一個一維陣列 a 和運算子 $\oplus$，然後輸出一維陣列 $b = \text{scan}(a, \oplus)$，$b_i = a_0 \oplus a_1 \oplus a_2 \ldots \oplus a_i$；$k$ 為 core 的數量，而 k_i 負責計算 a_i，所有的 k 在執行 if $k_i \geq 2^d$ then $b_i = a_{i-2^{d-1}} \oplus a_i$ 時是同時的；在圖 8 是個陣列長度為 8 的例子；此演算法在一維陣列 a 的維度等於核心數時，其時間複雜度為 $O(\log_2 n)$。

在 2005 年時由 Horn [6] 將 Hillis 和 Steele 提出的平行掃描演算法實做於繪圖處理器上，此平行掃描演算法是設定在 core 的數量等於資料量且可平行執行，但在 CUDA 的架構下是以一個 warp 為執行單位，所以 block 內的所有的 thread 並不是平行執行，因此要使用到 CUDA 提供的同步指令 __syncthreads，當 SM 執行到這個指令時就會等待，直到 block 內的所有的 thread 都執行到這個指令後，才接下去執行；其演算法如下：

Horn 提出的演算法

輸入：長度為 k 的一維陣列 *share* 和運算子 $\oplus$

輸出：長度為 k 的一維陣列 *share* = scan(*a*, $\oplus$)

```
__device__ void GPU_scan(float *share)
1.    {for (d = 1 to log2n do)
1.1    {if Tid ≥ 2^d then share[out][ Tid ] =
          share [in][ Tid − 2^(d-1)] ⊕
                              share [in][ Tid ];
        else
          share[out][Tid] = share [in][ Tid ];
1.2    __syncthreads();
1.3    swap(in, out);
      }
 2.   output share;  // share = scan(a, ⊕)}
```

Horn 提出的演算法利用 *Tid* 綁定所執行的陣列單元，且需用到兩倍 thread 數的 share memory 陣列 *share*[2][*n*]；而此演算法只適用於一個 block 內部的 thread。

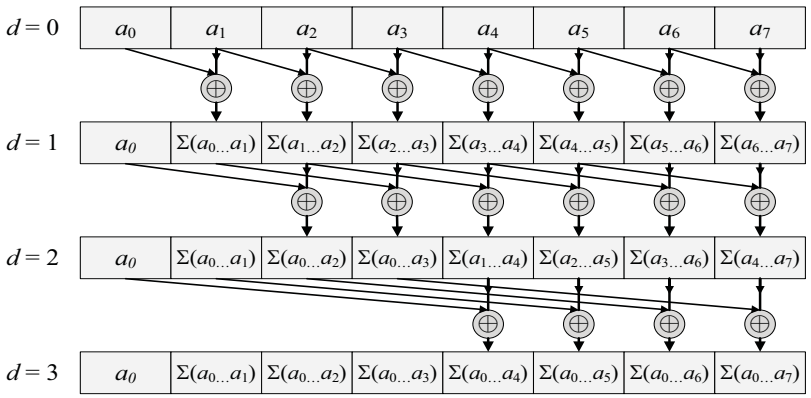


圖 8 陣列大小為 8 的前綴平行掃描演算法

本篇論文修改了 Horn 提出的演算法，並利用在本論文演算法的步驟三中去傳遞物件點的距離，使直條上的每個單位元素都記錄著，在自己所屬的橫條上離自己最近的物件點距離，下方的演算法為將 Horn 提出的演算法改為可傳遞物件點的距離演算法：

本論文演算法步驟三的掃瞄部分

輸入：一維陣列 *share* 其陣列單位為物件點或背景像素點

輸出：一維陣列 *share* 記錄從左方或右方來的最近物件點距離

```
__device__ void scan_DT(float *share)
1 {for (d = 1 to log2n do)
1.1     {if Tid ≥ 2^d hare[out][Tid] =
            min(share[in][Tid − 2^(d-1)] + 2^(d-1),
                        share [in][ Tid ]);
         else
            share[out][Tid] = share [in][ Tid ];
1.2      _syncthreads();
1.3      wap(in,out);
1.4      output share [out][Tid]; // share 記錄從左方或右方來的最近物件點距離
1.5       __syncthreads();
}
   }
```

本論文演算法步驟三的掃瞄部分其目的為傳遞物件點的距離，而下方的演算法為呼叫步驟三的掃瞄部分去執行向右及向左傳遞；下方的演算法中利用 *Bid* 決定各個 block 所負責計算的橫條，利用 *Tid* 決定橫條內各個 thread 所負責計算的陣列單位，接著將陣列 *I* 的資料複製到 share memory 中；在對 share 執行物件點距離傳遞前，必須先利用 scan_fix() 修正 share[*out*][0]，等到 block 內所有 thread 都執行完這步後才開始執行 scan_DT。（圖 9 為向右傳遞範例）

本論文演算法的步驟三

輸入：一張 $h \times w$ 的二元影像 *I*

輸出：影像 *I* 的每一橫條都為一維的歐氏距離地圖

```
__global__ void GPU_LR_scan(float *I )
1.      {extern __shared__ float share[];
                //向右傳遞
2.       for (i = Bid；i < h；i += block 總數)
2.1.      {for ( j = Tid；j < w； j += 一個
                        block 的 thread 數)
2.1.1.     {share [out][Tid] = I [i][j]；
2.1.2.      if(Tid == 0)
              share [out][Tid] = scan_fix(
                        I [i][j], I [i][j-1])；
2.1.3.      __syncthreads()；
2.1.4.      scan_DT ( I , share)；
            }
          }
3.        __syncthreads()；
//向左傳遞
4.       for (i = Bid；i < h；i += block 總數)
4.1.      {for ( j = Tid + w -1-2* Tid；j > 0；j −=
                一個 block 的 thread 數)
4.1.1.     {share [out][Tid] = I [i][j]；
4.1.2.      if(Tid == 0)
              share [out][Tid] = scan_fix(
                        I [i][j],I [i][j+1])；
4.1.3.      __syncthreads()；
4.1.4.      scan_DT ( I , share)；
            }
          }
5.      output  I；}  //  I 的每一橫條都為一維的歐氏距離地圖
```

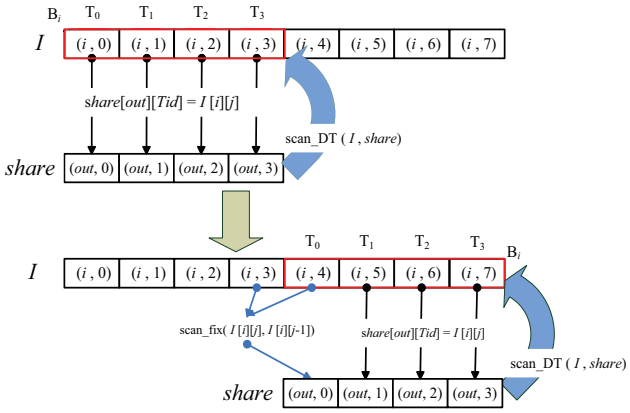


圖 9 向右掃描範例

本論文演算法步驟三的修復部分

輸入：兩個變數 a, b

輸出：a

__device__ void scan_fix(float a , float b)

1. {if($a > b + 1$)

1.1. $a = b + 1$；

2. output a；

}

左右向的物件點距離傳遞結束後，本論文演算法步驟四為向下傳遞建立陣列 D 及向上傳遞建立陣列 U，其演算法如下，概念與左右向傳遞一樣左右向的物件點距離傳遞結束後，本論文演算法步驟四為向下傳遞建立陣列 D 及向上傳遞建立陣列 U，其演算法如下，概念與左右向一樣。

本論文演算法步驟四

輸入：一張 $h \times w$ 的二元影像 I

輸出：陣列 D 記入來自於上方物件點的 y 軸距離、陣列 U 記入來自於下方物件點的 y 軸距離。

__global__ void GPU_UD_scan(float *I)

1. {extern __shared__ float *share* []；
//向下傳遞
2. for ($j = Bid$; $j < h$; j += block 總數)
2.1. { for ($i = Tid$; $i < w$; i += 一個 block 的 thread 數)
2.1.1. {if(I [i][j] < ∞)
share [*out*][*Tid*] = 0；
else
share[*out*][*Tid*] = I [i][j]；
2.1.2. if(*Tid* == 0)
share [out][*Tid*] = scan_fix(
share [out][*Tid*] , D [i - 1][j])；
2.1.3. __syncthreads()；
2.1.4. scan_DT (D , *share*)；
}
}
3. __syncthreads()；
//向上傳遞
4. for ($j = Bid$; $j < h$; j += block 總數)
4.1. {for ($i = Tid + w - 1 - 2*Tid$; $i > 0$; i -= 一個 block 的 thread 數)
4.1.1. {if(I [i][j] < ∞)
share [out][*Tid*] = 0；
else
share [out][*Tid*] = I [i][j]；
4.1.2. if(*Tid* == 0)
share [out][*Tid*] =
scan_fix(*share* [out][*Tid*] ,
U [i + 1][j])；
4.1.3. __syncthreads()；
4.1.4. scan_DT (U , *share*)；
}
}
5. output U 和 D；// D 記錄位於上方物件點的 y 軸距離、U 記錄位於下方物件點的 y 軸距離

本論文演算法步驟四和步驟三的差別在於，每個 block 被設定為計算直行，且同一個 block 內的 thread 會被分配到同一條直行上，各自計算自己負責的陣列元素；在將影像 I 複製到 *share* 時，會將小於 ∞ 的值設為 0 複製到 *share* 中，再執行 scan_DT()。

當建立好影像 I、陣列 D 和陣列 U 後，可以透過這三個陣列的資訊去搜尋每個背景像素點最接近的物件點距離。(圖 10 為當影像 I、陣列 D 和陣列 U 建立好後 8×8 的範例)。

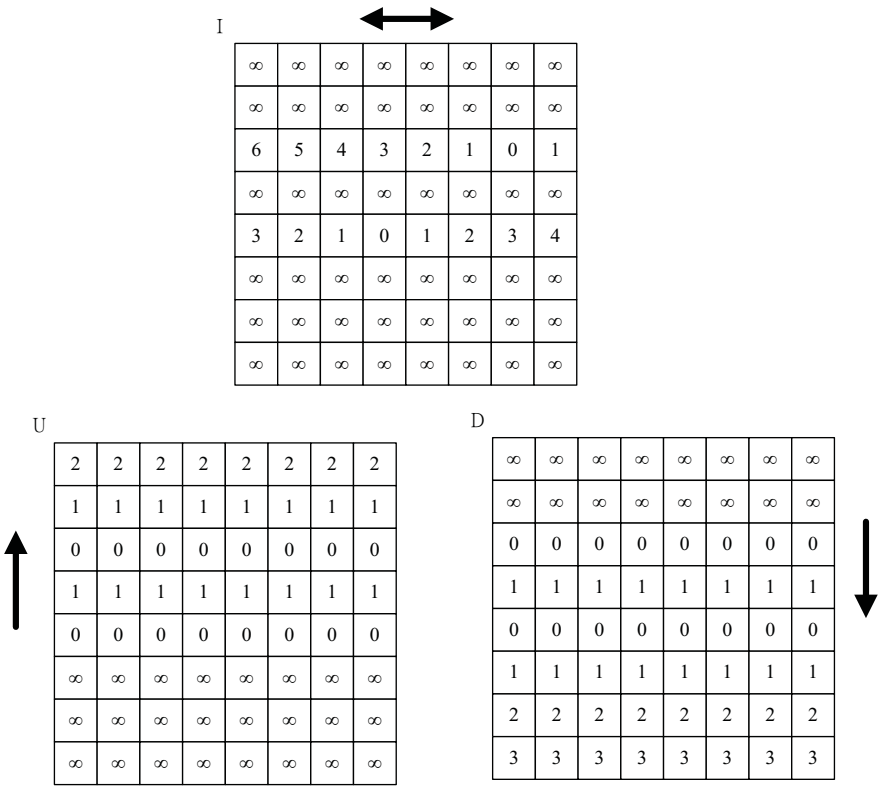


圖 10 為影像 I、陣列 D 和陣列 U 建立好後 8×8 的範例

三.5.1. 搜尋最近的物件點演算法

本論文演算法步驟五向下計算最近點的距離

輸入：$h \times w$ 影像 I 和陣列 U
輸出：陣列 U 其內記錄來自於下方最近的物件點距離

```
__device__ void closest_object (float *I ,
float *U)
1.    { for (y = Bid；y < h；y += block 總數)
1.1.      { for (x = Tid；x < w；x +=  一個
                    block 的 thread 數)
1.          { for(Dy = U [x][y] ;
                Bottom = ∞ - 1；Dy < Bottom；
                  Dy = 1 + U [x+ Dy+1][y])
1.1.1.1.      { NewDist = Dist ( I[x][Dy],Dy)；
1.1.1.2.        if (NewDist < Bottom)
                      Bottom = NewDist；
              }
1.1.2.         __syncthreads()；
1.1.3.         U [x][y] = Bottom；
          }
        }
      }
2.    output  U；}  // U其內記錄來自於下
      方最近的物件點距離
```

上方演算法為向下搜尋最近的物件點距離，當執行完上方演算法時，可得到當前背景點以下最近的物件點距離；演算法利用 D_y 的增加往下找點和 Bottom 的減少來縮短所需向下尋找的範圍，首先先將 D_y 設為 U [x][y]，是因為 U [x][y] 內部記錄的是物件點的縱向距離，此縱向距離為距離背景像素點 (x, y) 下方最近的縱向距離，且透過 I [x][D_y]可得到橫向距離，再利用並利用 Dist() 來計算其距離，公式如下：

$$\text{Dist}(I\,[x][D_y], D_y) = \sqrt{(I[x][D_y])^2 + (D_y)^2}$$

此物件點的距離不見得會是最近的，但可透過計算此點距離存入 *Bottom* 中來排除多餘不需要計算的物件點，當有發現新的物件點距離比較短時，就更新 *Bottom*。（如圖 11）而向上搜尋其概念一樣，利用影像 I 和陣列 D，可得到當前背景像素點以上最近的物件點距離；並比較陣列 U 和陣列 D 內較小者存回影像 I 中，再回傳給 host。

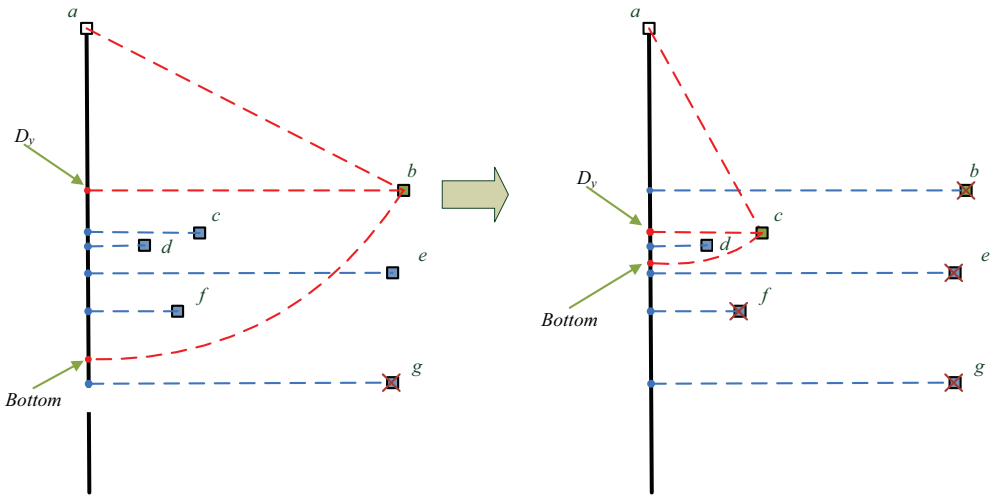


圖 11 向下搜尋物件點

四、 實驗成果

本章會說明本論文所提出之演算法，在不同數目的 block 數和 thread 數的情況下所得之運算結果，分析較佳的組合；並與 CPU 窮舉法、GPU 窮舉法和 CPU 雙向掃描演算法做比較。

四.1. 繪圖處理器的分配

在 GTX560ti 中每個 SM 一次最多可以排程 1536 個 thread 和 8 個 block;因 SM 總數為 8 所以 GTX560ti 一次最多可排程的 block 數為 64，但由於可排程的 thread 數為 1536，所以 thread 和 block 數是必須做分配的。

四.1.1. Block及Thread的分配

圖 12 為分配一個 block 含有 256 個 thread 且 block 變動的實驗結果；一個 SM 一次只會排程 1536 / 256 = 6 個 block，8 個 SM 共排程 48 個 block，雖然一個 SM 內只有 6 個 block 但由於 SM 可容納的 thread 數 1536 已滿載，因此不需要將 SM 內的 block 數排滿，如圖 12 所示，當 block 數為 48 時每個 SM 都排滿 1536 個 thread 其效果為佳，當 block 為 56 時其執行時間需較長，而當 block 數接近 96 時，其執行時間又逐漸縮短；在thread 數量的選擇，以 1536 的因數為佳，且在 8 個 block 內可排滿 1536 個 thread，例如 192、256、384、512 及 768 個 thread，block 數量的選擇為(1536 / thread數 ×8 (SM))的倍數。

圖 13 為 block 固定、thread 數變動且都滿載的實驗結果；以 3072 個 block 為例當 thread 數增加時其計算時間會降低，且 3072 個 block 為圖 13 中 block 數為最多且效率為最好；由此得知 block 數和 thread 數在滿載的情況下越多越好。

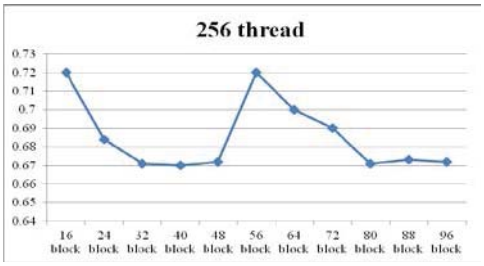


圖 12

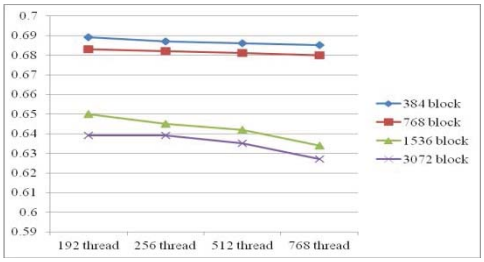


圖 13

物件點數量	10	20	30	40	50	60	70	80	90
本論文演算法	0.43s	0.43s	0.43s	0.43s	0.44s	0.44s	0.44s	0.44s	0.45s
GPU 窮舉演算法	0.12s	0.15s	0.18s	0.21s	0.23s	0.26s	0.28s	0.31s	0.34s
CPU 窮舉演算法	3.62s	7.51s	11.94s	16.39s	21.3s	25.24s	29.94s	34.26s	38.56s
CPU 雙向掃描演算法	0.23s	0.23s	0.23s	0.22s	0.23s	0.23s	0.23s	0.23s	0.23s

表 1 四種距離轉換演算法在點數少時所需時間

物件點數量	100	200	300	400	500	600	700	800	900
本論文的演算法	0.44s	0.46s	0.48s	0.5s	0.52s	0.55s	0.6s	0.64s	0.67s
GPU 窮舉演算法	0.39s	0.63s	0.9s	1.16s	1.43s	1.69s	1.96s	2.23s	2.5s
CPU 窮舉演算法	42.08s	83.34s	124.03s	167.23s	210.46s	252.51s	295.35s	335.66s	378.1s
CPU 雙向掃描演算法	0.23s	0.23s	0.23s	0.22s	0.23s	0.23s	0.23s	0.23s	0.23s

表 2 四種距離轉換演算法在點數較多時所需時間

四.2. 實驗結果

實驗中所使用的資料樣本為 6144 × 6144的二元影像，且 GPU 上的 block 分配為 6144、thread 為 768，在雙向掃描演算法中由於物件點數不會影響其計算效率，因此在物件點較多時其雙向掃描的計算時間會是最優的，但本論文所提出的演算法，在物件點密度極高時，會排除很多不需計算的物件點，因此計算效率是令人滿意的。

表 1 彙整了 GPU 窮舉距離轉換、本論文的距離轉換演算法、CPU 窮舉距離轉換和 CPU 雙向掃描算法，在物件點少時其效率最好的為 GPU 窮舉距離轉換，但由於窮舉法每多一個物件點就要增加 n 平方次的計算量，因此所需的計算時間成長幅度很大；而 CPU 雙向掃描算法對每個點都是比較周圍八個點的大小，所以物件點的多寡不會有影響；而本論文提出的演算法在物件點少時其效率不是太好，但物件點多時，可以排除不需計算的物件點，因此物件點的多寡對其演算法影響較小，如 表 2 顯示物件點超過 200 時其本論文所提出的演算法花費時間比 GPU 窮舉距離轉換少；由 CPU 和 GPU 的窮舉法可發現，GPU 在做數字計算時其效能是大大的優於 CPU，由此可知 GPU 再做數字計算的效能是優於 CPU 的。

五、 結論

本論文發表一個建構於繪圖處理器上，以 Horn 演算法為基礎平行距離轉換演算法，透過上下限制的方式，去排除不需要計算的物件點；雖然其效率不如雙向掃描演算法好，但雙向掃描演算法並不是計算實際的距離，因此在某些需要精確度高的運用方面，本論文所提出的演算法也有其優點。

參考文獻

1. Joon Seop Oh,Yoon Ho Choi,Jin Bae Park,Zheng Y.F. Complete coverage navigation of cleaning robots using triangular-cell-based map. *IEEE Transactions on Industrial Electronics, 2004* .
2. Milos Sramek and Arie Kaufman. Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics, 2000.*
3. Azriel Rosenfeld. and John L Pfaltz. Distance functions on digital pictures. *Pattern Recognition, 1968*.
4. Shyong Jian Shyu, Ting Wei Chou and Tsorng Lin Chia. Distance Transformation in Parallel. *Journal of Informatics & Electronics, 2006.*
5. Hillis, W. Daniel, and Steele Jr., Guy L. Data Parallel Algorithms. *Communications of the ACM*, *1986*.
6. Horn, Daniel. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, M. Pharr, Ed., ch. 36, pp. 573–589. Addison Wesley, *2005.*
7. NVDIA Corporation. NVIDIA Fermi Compute Architecture Whitepaper, *2009*
8. NVDIA Corporation. CUDA C Programming Guide Version 4.2, *2012*.