Final Project Report

Topic: Parallel Low-Poly Style Image Processing

Member:
徐浩宇 106061203
Key words:
Image processing, computer graphics

1. Introduction

Low-poly style image 是使用有限數量的 polygons 去表達一張圖片，通常在早期電腦運算資源不夠強大時， 會在電腦遊戲中使用此圖像表示方法。同時，low-poly style image 也可以有藝術方面的應用，透過轉換圖片 達到一種極簡化的抽象藝術風格。在這次 Final Project 中，我實作了整個 low-poly style image processing 的流程，接著使用 CUDA 去做平行化加速，同時去分析平行化後各個部分效能的提升。

Original Image

Low-Poly Image

2. Pipeline

根據 *Low-poly style image and video processing* [1]，我們可以把整個流程分為 4 個 stages:

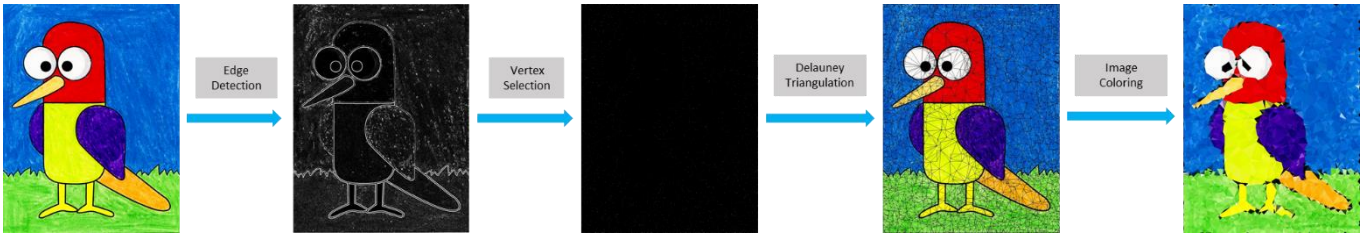
1. Edge Detection: 將圖片做 sobel 濾波後，偵測出圖片的邊緣特徵。

2. Vertex Selection: 利用前面的結果，隨機在圖片內 sample 出數個節點。

3. Delauney Triangulation: 將 sample 出來的節點做三角化。

4. Image Coloring: 將圖片以三角形為基礎做上色。

詳細流程示意圖可以在下一頁看到，關於每一個 stage 的做法，則會在後面分別作詳細說明。



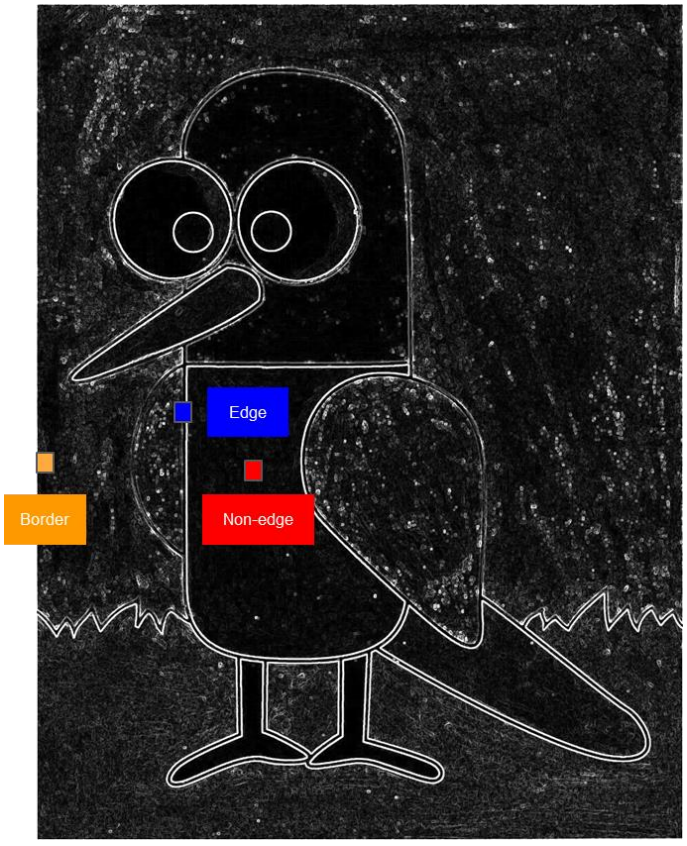
(1) Edge Detection:

我使用了兩個方向的簡易 sobel filter (大小為 3 x 3)去做濾波，並將兩個 filter 的結果取絕對值相加當作 edge 偵測的強度。

$$filter_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad filter_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

(2) Vertex Selection:

為了維持 Low-poly image 的品質，保有 geometric structure 的完整性，我會提高圖片邊緣 (edge) 被 sample 到的機率，利用邊緣閥值 (threshold) 判斷該 pixel 是否落在邊緣特徵上。同時，圖片邊界 (border) 也會有較高的機率被 sample 到，防止破圖的產生。



Vertex selection 各參數的設定:

<1> edge threshold: 20

<2> edge probability: 0.005

<3> non-edge probability: 0.0001

<4> border probability: 0.1

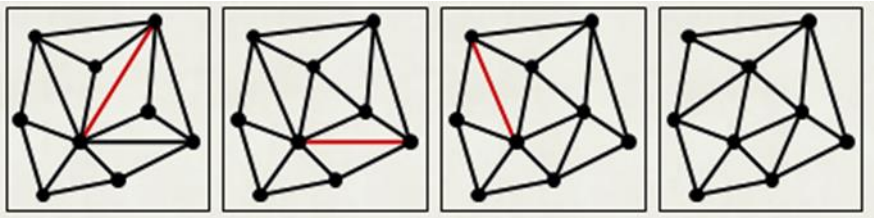
(3) Delauney Triangulation:

我採用 delauney triangulation 的主要原因是它的一個性質《Maxmin Angle Triangulation》，簡言之它會讓建構出的三角形，其最小角度最大化。如此一來，它比較不會產生細長狹窄的三角形，對於圖片上色的效果會比較好。

一般常見建構 delauney triangles 的方法有兩種:

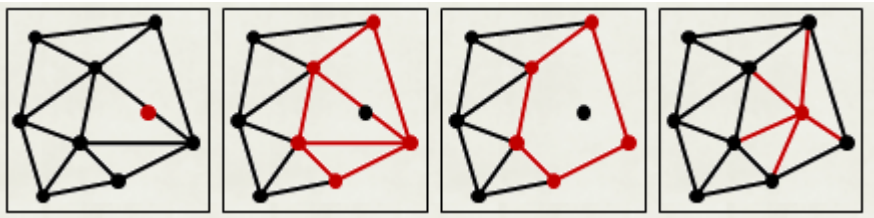
<1> Lawson's Flip Algorithm:

先隨意生成一種 triangulation (ex: scan method)，接著再去檢查每一條邊是否符合 delauney 性質，如果不符合便將其翻轉掉，直到所有邊都滿足 delauney 特性為止。時間複雜度為$O(N^2)$。



<2> Bowyer-Watson Algorithm:

每一次加入一個點，並檢查是否符合 delauney 性質，如果不符合就會重新連結當前點周圍的多邊形頂點，直至所有點都加入為止。時間複雜度為$O(N^2)$。



然而，這兩種演算法比較不具有 parallel 的性質，例如:

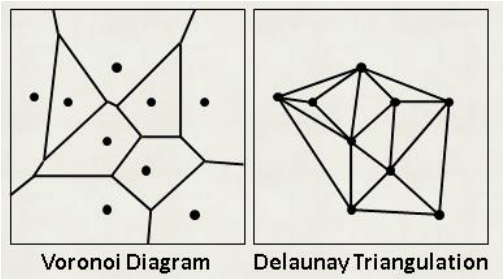
<1>在更新一個邊時，可能會連帶另外數條邊必須因此更新。

<2>第 N 點的加入會跟前面 N-1 個點的結果有關。

因此，我們會用到 Delaunay triangulation 另一種有趣的性質。

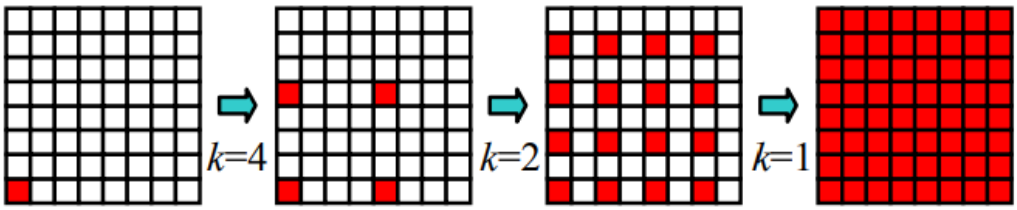
The Delaunay triangulation of a discrete point set P in general position corresponds to the dual graph of the Voronoi diagram for P.

上面那句話說明了 Delauney triangulation 會和 Voronoi diagram 具有對偶圖的關係，也就是說我們可以先求出 Voronoi diagram，再透過轉換得到相對應的 Delauney triangulation。



Voronoi diagram 類似 nearest neighbor，一個 vertex A 會透過計算與其相鄰 vertices 之間的中垂線，去圍出一塊區域，區域內的所有點都以該 vertex A 作為最近的節點。

一般生成 Voronoi diagram 會使用 jump flooding algorithm[2]，主要概念為: 每個節點會往八個方向做擴散(flooding)，持續擴散直到結果穩定。實作上每次 step 擴散大小都會除以 2，直到等於 1 便會結束，示意圖如下。

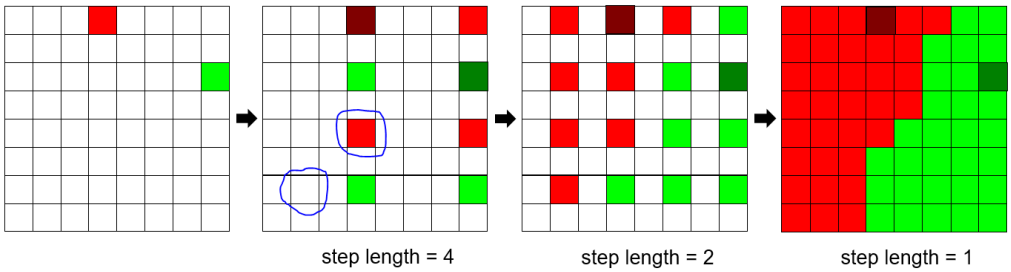


假設在 step_size=M 情況，我們會 iterate 所有的 pixels。其中，一個 target pixel 會分別觀察在 8 個方向上距離該 target pixel 為 M 的 neighbor pixel，各屬於(最接近於)哪一個 vertex。

(1) 如果 neighbor pixel 不屬於任何一個 vertex 則不做任何事

(2) 如果 neighbor pixel 目前屬於 vertex v1，則在以下兩個情況下會更新 target pixel 所屬的 vertex 為 v1:

- target pixel 目前不屬於任何一個 vertex
- target pixel 目前屬於 vertex v0 且 distance(target pixel, v1) < distance(target pixel, v0)的條件成立，則代表 v1 的距離更接近 target pixel



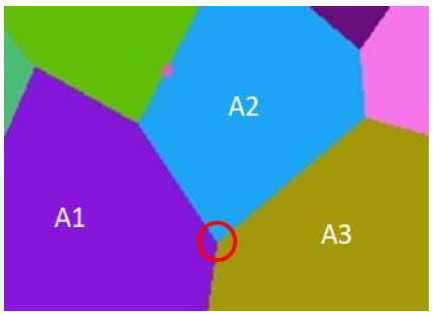
~jump flooding 詳細的 pseudo code 可以在後面看到~

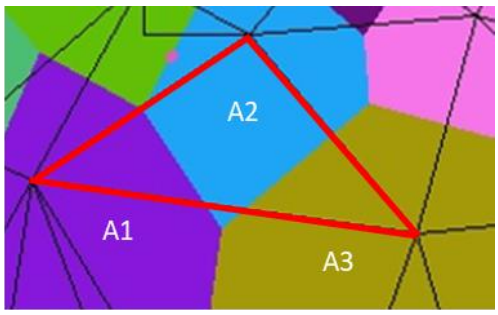
```
Procedure jump_flooding
Begin
  // flooding with decreasing step size
  For step=min(H, W)/2 to 1
    // iterating all pixels
    For each pixel p
      // iterating all 8 directions
      For each direction dir
        ni := neighbor pixel (p + dir * step)
        // get the nearest vertex (owner) of p
        If p not owned or distance(p, ni's owner) < distance(p, p's owner)
          Update p's owner with ni's owner
        End if
      End for
    End for
    step /= 2    // update step size
  End for
End
```

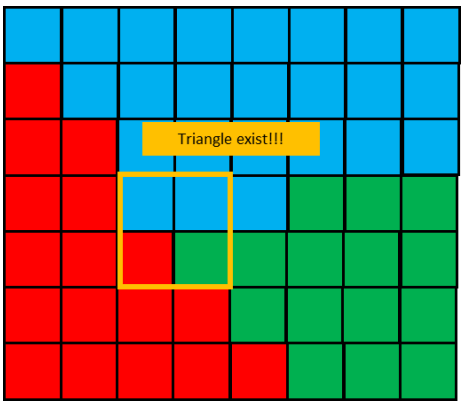
Not faster, but can be parallelize

$$Time\ Complexity = log_2N * N^2 * 8 = O(N^2 log_2N)$$

計算完 Voronoi diagram 後，只要找到 3 個區域的交界處，就可以把 3 個區域各自代表的 vertex (即 owner) 相連組成三角形；假如有 4 個區域的交界，則可以分別組成兩個三角形。基本上就是觀察一個點周圍右、右下、下位置的代表 vertex，加上自己的代表 vertex，並觀察這四個 vertices 當中 distinct vertices 數量即為交界於該點的區域個數。



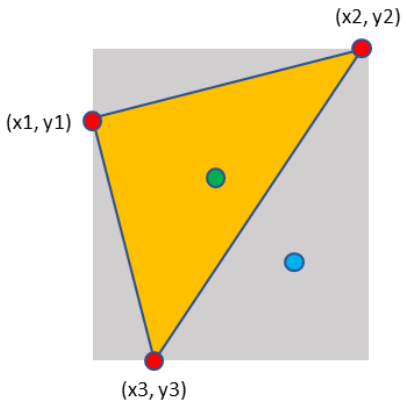




~Delaunay Triangulation 的步驟完成~

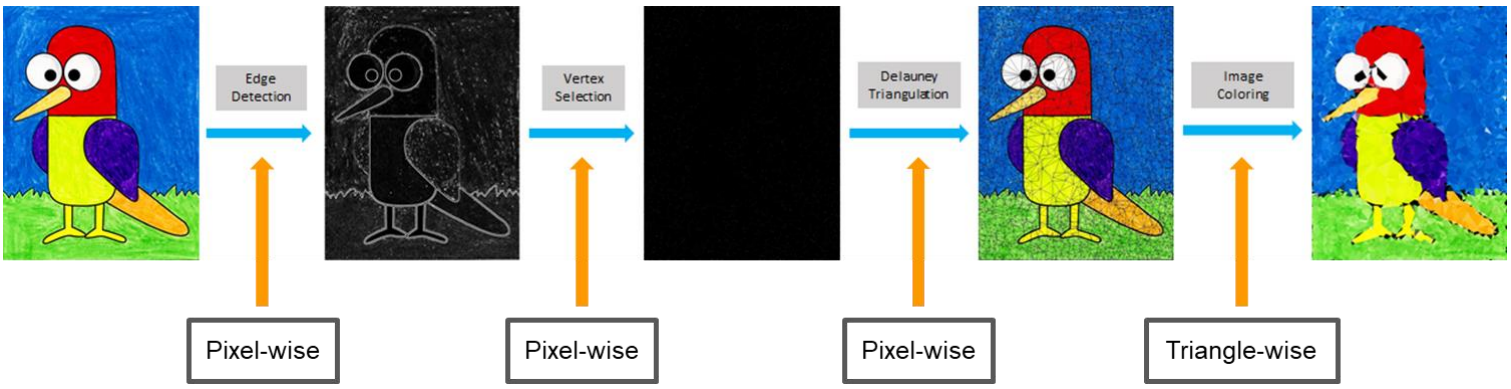
(4) Image Coloring:

我會對每一個三角形進行上色，首先計算出包圍三角形的長方形區域，接著依序嘗試長方形區域內的每個 pixel。若 pixel A (green dot) 位於三角形內部，則以三角形中心點的顏色作為該 pixel A 的顏色；反之，若 pixel A (blue dot) 位於三角形外則不做事。我參考 Stackoverflow 上面的作法[3]，可以很簡單地算出一個 2D 點是否落在 2D 三角形內部。



3. Parallel Strategy

實作完 sequential 版本後，可以針對(1)~(4)每個部份下去平行化。



(1) Edge Detection:

可以做 pixel-wise 的平行化，每一個 CUDA thread 負責計算一個 pixel 經過 filter 後的 gradient 數值加總。

(2) Vertex Selection:

由於需要專門使用在 GPU 的亂數方法，這邊會使用到 **CURAND**。每一個點都需要獨自做亂數，所以總共會需要 H*W 個 curandState。

首先，我平行初始化每一個 pixel 的亂數種子，得到相對應的 curandState。接著，每一個 thread 會利用自己的 curandState 去隨機 uniform 亂數，並根據亂數數值及其負責 pixel 的種類，去決定負責的 pixel 是否會作為節點。

(3) Delauney Triangulation:

- Voronoi diagram:

  在使用 for loop 去做每一種 step size 時，都會對全部的 pixels 做更新。因此，我可以簡單地做 pixel-wise 的平行化，讓每一個 thread 負責一個 pixel，透過觀察 8 個方向上 neighbor pixels 的資訊，去更新該 thread 所負責 pixel 的資訊。

  GPU 無法確定每一個 pixel 處理的順序，有可能正在看的 neighbor pixel 資訊剛更新，卻因為執行順序先後導致其他 pixel 沒辦法看到最新

的資訊，使得 flooding 結果有 noise pattern 產生。然而，我覺得這種問題影響不大，得到 voronoi diagram 的近似解是可以接受的，畢竟這些 flooding 的小誤差不會對最終圖案的品質有太大影響。

- Triangulation:

  我選擇使用 pixel-wise 平行化處理，每一個 thread 會負責檢查一個 pixel 是否在區域的交界處。如果是在交界處，就生成三角形。

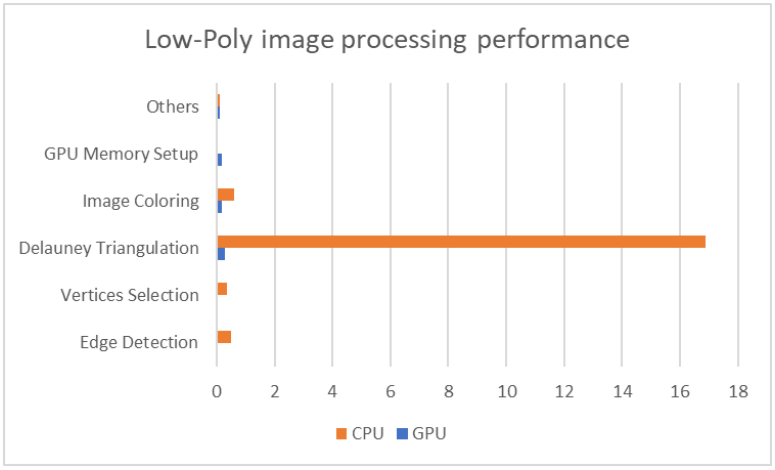
  因為 GPU 必須預先分配記憶體，所以實作上我先平行化地判斷每一個 pixel 含有的三角形數量 (0, 1 or 2)。接著，再計算 prefix sum 去得到全部三角形的數量，順便利用累積加總的特性，得到每個交界處形成的三角形的 index。最後可以利用這些資訊，去分配相應大小的記憶體並執行 triangulation。

(4) Image Coloring:

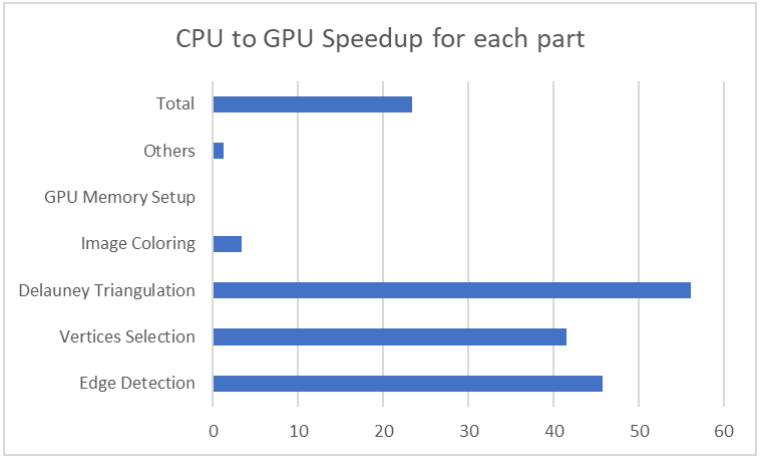
這邊和前面相比有些不同，沒有採用 pixel-wise 平行化，而是以 triangle-wise 的方式平行化，即每一個 thread 會負責一個三角形的上色動作。

4. Performance & Optimization

為了統一時間測量的標準，這邊全部使用第一頁的 burger.jpg 去做測試，圖片的長寬為 3247x4330，總共會產生 24836 個三角形。



單位: 秒



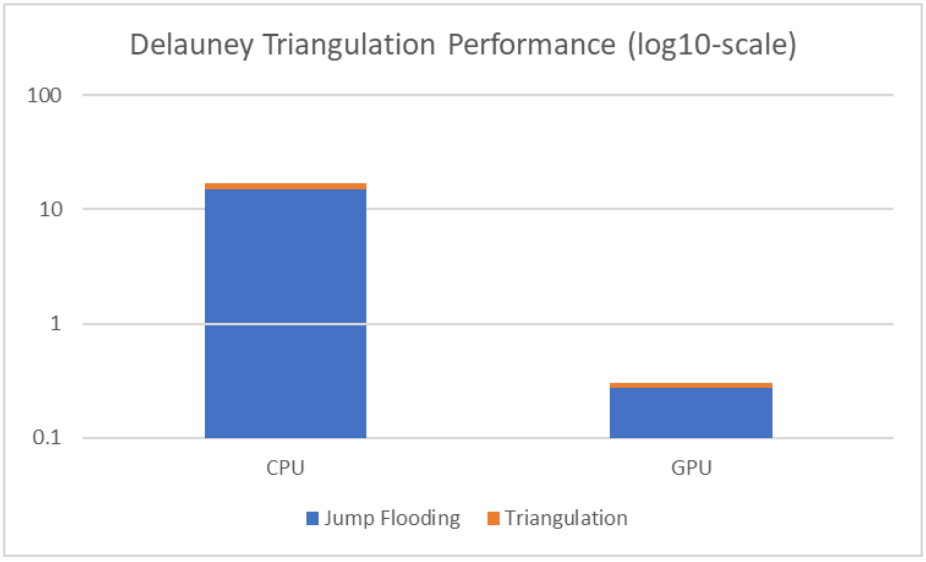
單位: 倍數

這邊可以發現 CPU 會花費大量的時間在 Delauney triangulation 上面，而使用 GPU 去做可以大大降低運行的時間。至於 speedup 部分，可以觀察到只要有使用 pixel-wise 平行化的部分 (i.e.前三步驟)，在高畫質圖片的 testcase 下，都會有超過 **40 倍**的 speedup；然而，使用 triangle-wise 平行化的 image coloring 則沒有這麼大的 speedup。

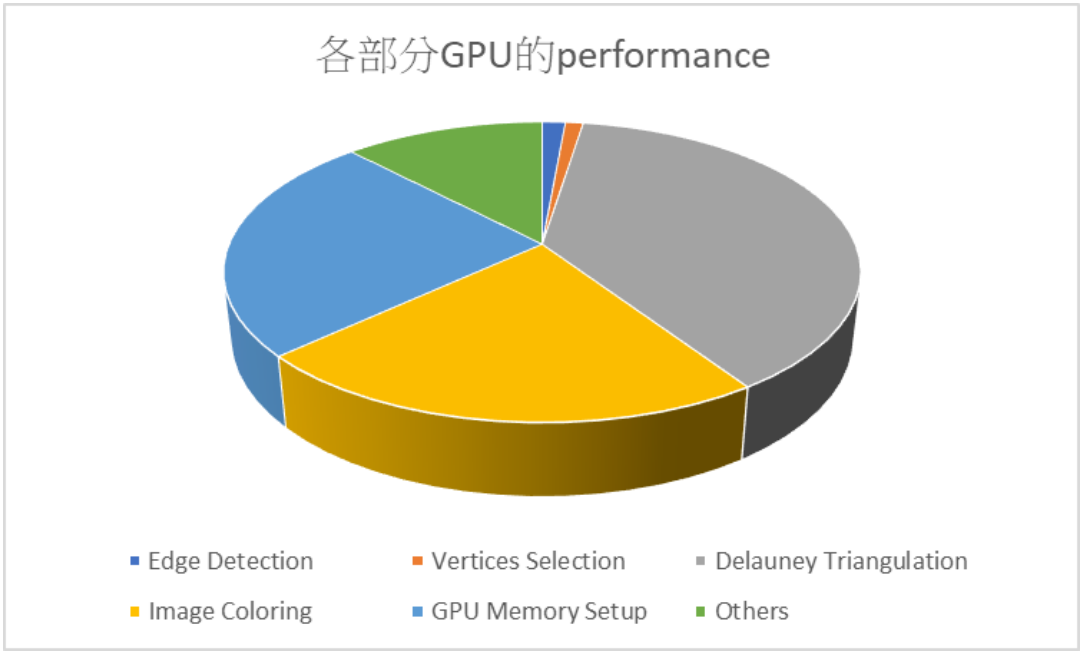
進一步觀察 Delauney triangulation 的部分，可以看出大多數的時間會花費在形成 voronoi diagram 上面，算蠻合理的。假設圖片大小為 N*N，那麼總共會有 log2(N)次的更新，每次更新 N^2 個 pixels，每個 pixels 需要看 8 個方向:

$$Time\ Complexity = log_2 N * N^2 * 8 = O(N^2 log_2 N)$$

多出來的 log2(N)對整體 performance 影響很大。



單位: log 秒



下面說明了我實作中有嘗試過成功的加速方法以及其效果:

- **Random state initialization:**

```
curand_init ( long long seed, long long sequence, long long offset, curandState_t *state)
```

通常在做平行化亂數初始化時，會固定 seed 並輸入不同的 sequence number 以得到亂數值(如下圖)。雖然這樣的方式可以有比較好的 random 性質，即2^{67}次後才會重複 loop，但是在 CURAND 會花費大量的時間。

```
curand_init(SEED, id, 0, &state[id]);
```

```
curand_init( (SEED << 20) + id, 0, 0, &state[id])
```

因此，我修改了初始化的方式，改成使用不同的 seed 加上相同的 sequence number (0)，有效地將時間從 115s 縮減為 0.0052s。通常不同的 seed 之間產生的數值不太會 statistically correlated，而且這次的 task 對於亂數品質的要求不用這麼完美，所以這樣的加速是可行的。

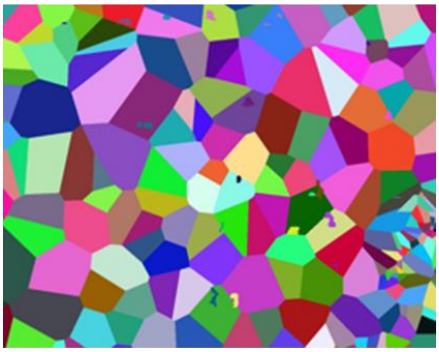
- **Use "Thrust" for prefix sum:**

通常像是 prefix sum 這種 sequential 的任務，在 GPU 的表現會比較差。原本我的方法是先將 GPU 的資料搬到 CPU，在 CPU 端做 prefix sum 後再搬回去 GPU；之後，我改使用 Thrust library 直接在 GPU 做 prefix sum，可以把時間從 0.0412s 降至 0.0008s，省去資料搬運產生的 latency。

- **2D jump flooding:**

原本我使用 1D 的方式去分配 jump flooding 的任務，卻發現產生的 Voronoi diagram 會出現很多 noise。後面我改成用 2D 的方式去分配任務，發現 Voronoi diagram 的品質明顯改善。個人猜測是因為 2D 的分配方式在 step size 小的時候會有 **locality** 特性，能夠更 consistent 地去更新每一個 pixel 的 owner，使得 noise 可以有效地減少。

Voronoi 1D



Voronoi 2D

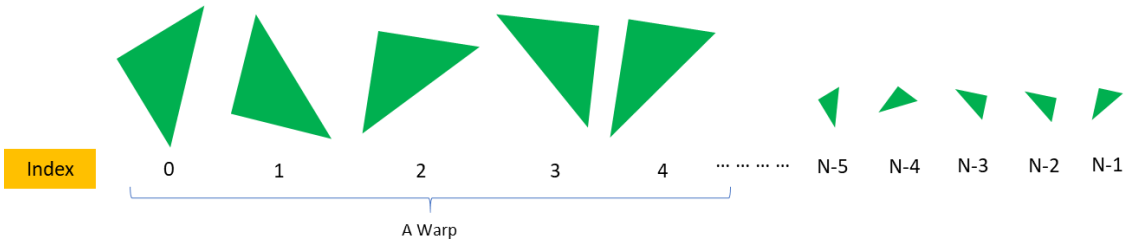
- **Sort-by-key on triangles' area:**

  前面有提到說 Image Coloring 是所有部分 speedup 最少的，我認為主要原因是，即使可以平行化去填色，仍然存在著 load imbalance 的問題。因為每一個三角形的大小不同，所以每一個 block 的運行時間會被裡面最大的三角形 dominate，導致運算效率不夠高。

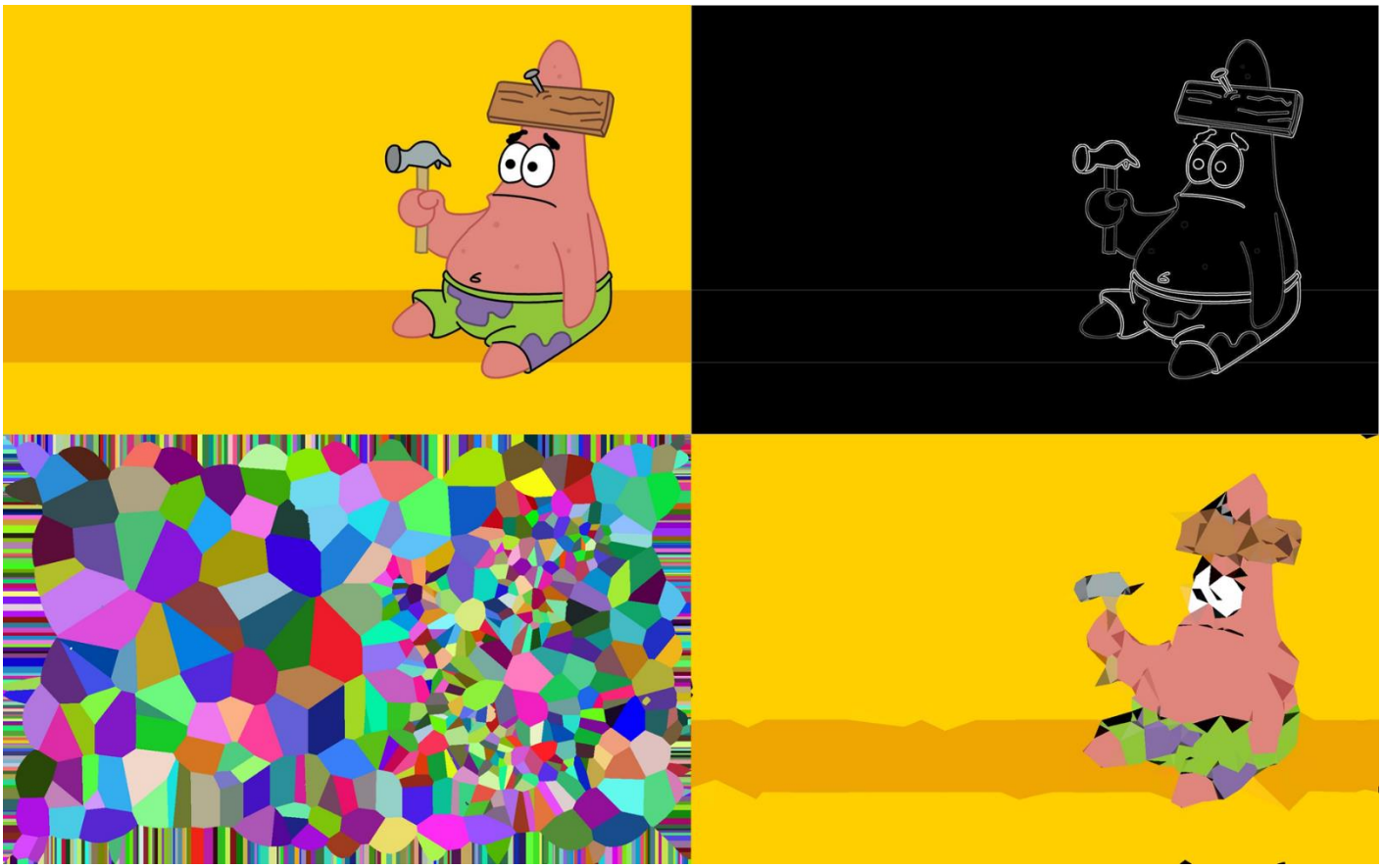
  我把 block 大小從 256 降至 64 後，花費時間從 0.268s 降至 0.251s。

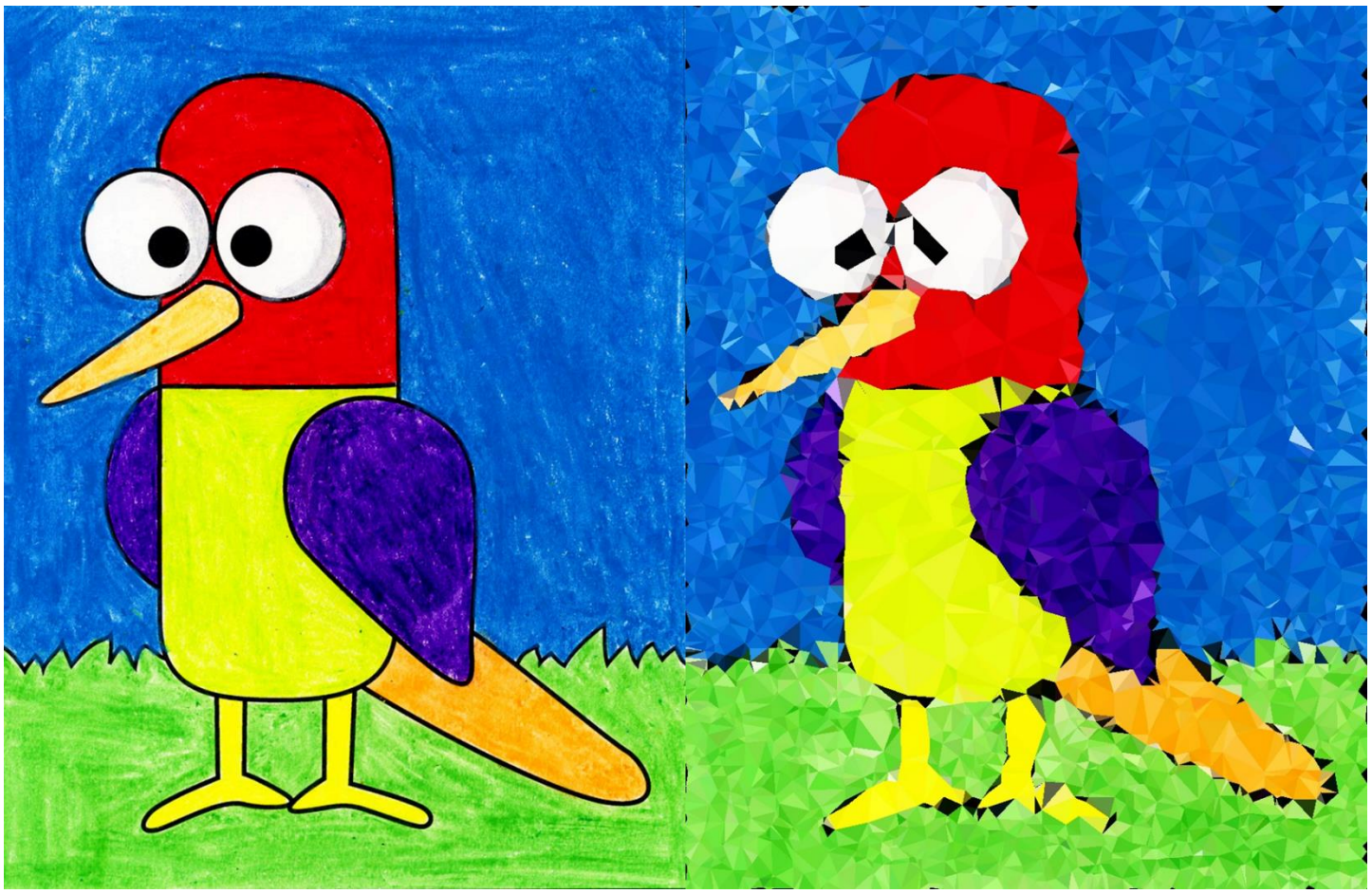
  我認為若要根本解決 load imbalance 的問題，必須讓每一個 block 裡面的三角形大小相近，以達到高效率的運算。最直覺的方法便是依照面積大小 sorting，如此一來，GPU 分配的同一個 block 裡面的三角形大小會更接近。

  我先平行計算出每一個三角形外接 bounding box 的面積，接著利用 Thrust library 裡面的 sort-by-key 函式，讓三角形依照其 bounding box 面積去做排序。經過測試發現，運行時間有效降低至 0.179s！



5. Qualitative Results





6. Remarks

透過這次 final project，我更了解到如何 step-by-step 去平行化處理一個 task。除了實作部分外，我還花了不少時間在 server 上安裝 OpenCV 以及嘗試 separable compilation 上面。另外，我想解釋有些部分使用到 Thrust library 的原因，主要是為了避免自己的 implementation 會有 overhead，而失去原本可能潛在加速的機會 (ex: 預先 sort triangles 那一步)。

7. Reference

[1] Zhang, W., Xiao, S., & Shi, X. (2015). Low-poly style image and video processing. 2015 International Conference on Systems, Signals and Image Processing (IWSSIP)

[2] Guodong Rong, Tiow-Seng Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. Proceedings of the 2006 symposium on Interactive 3D graphics and games (ACM)

[3] Stackoverflow: How to determine if a point is in a 2D triangle?

(https://stackoverflow.com/questions/2049582/how-to-determine-if-a-point-is-in-a-2d-triangle)