110062118 賴姿妘

HW1: Adaptive Filtering

一、 優化方法說明

1. 改善 adaptive filter 算法

- **快速計算 subarray 和**

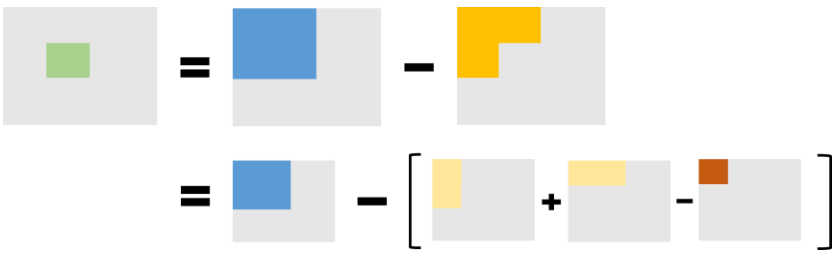
```
for (int x = 0; x < height; x++) {
    for (int y = 0; y < width; y++) {
        int kernelSize = kernelSizes[x][y];
        int kernelRadius = kernelSize / 2;
        double sum = 0.0;
        double filteredPixel = 0.0;

        for (int i = -kernelRadius; i <= kernelRadius; i++) {
            for (int j = -kernelRadius; j <= kernelRadius; j++) {
                int pixelX = std::min(std::max(x + i, 0), height - 1);
                int pixelY = std::min(std::max(y + j, 0), width - 1);
                filteredPixel += input[pixelX][pixelY];
                sum += 1.0;
            }
        }

        output[x][y] = static_cast<int>(filteredPixel / sum);
```

圖一、原本 adaptive filter 算法

首先，我先檢視在原本序列進行的版本中，是否還有可以改善的成分。在原本的 adaptive filter 中，如圖一所示會不停計算以(x,y)為中心四周 pixel 的 R/G/B 值和，但是這種算法十分浪費。舉例來說，若有相鄰兩格要計算 filter 中 pixel 的 sum，將近 5/4 的 pixel 和已經被計算過了，整體的複雜度如下列式子，其中 m 為高度、n 為寬度、k 為 filter 大小，通常為 5：

$$O(m * n * k * k) \tag{1}$$

若能將計算 subarray 和的複雜度變成 O(1)，則至少能快 25 倍。而方法是使用 prefix sum。在 prefix sum 矩陣中，A[x2][y2]相當於將原矩陣 x=0 至 x=x2 和 y=0 至 y=y2 的元素相加。因此，若獲得 prefix sum 則能根據圖二方法，在 O(1) 時間得到 subarray 的和。



圖二、sum of subarray 計算示意圖

- **得到 Prefix sum array**

我自訂了一個函式將原本的 RGB array 轉化成 prefix sum array。算法如圖三。

```
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        prefixSum[i][j] = inputImage[i][j].r;
        if (i > 0) prefixSum[i][j] += prefixSum[i - 1][j];
        if (j > 0) prefixSum[i][j] += prefixSum[i][j - 1];
        if (i > 0 && j > 0) prefixSum[i][j] -= prefixSum[i - 1][j - 1];
    }
}
```

圖三、產生 Prefix sum array 算法

- **補充說明**

實作上，會遇到 filter 超出圖片邊界的問題，原本的演算法會將超出的部分用邊框的數值替代，因此，若遇到這種情況，使用 Prefix sum array 便會很麻煩。所以，在程式設計中，若發現 filter 超出圖片邊界就會使用原始方法計算總和。

2. 平行化

這份程式碼共有三個階段：讀入圖片、處理、寫出圖片。寫入寫出看起來不太有平行化的地方，而且又是呼叫 library，所以我不敢亂動。另外，查找 PNG 讀寫檔的原理後，發現 PNG 牽涉到序列壓縮和寫入，感覺也不是可以平行化的地方。

這份程式碼最能優化的部分是陣列運算的地方，且各元素運算彼此獨立且相同，不須在意同步和負載平衡的問題。以下列出所有進行平行化的部分，基本概念就是將 for loop 拆開計算。

Code	Function

`#pragma omp parallel for`
`for(int y = 0; y < height; y++) {`
`  row_pointers[y] = (png_byte*)malloc(png_get_rowbytes(png,info));`
`}`
	read_png_file

<table>
<tr>
<td>

```
#pragma omp parallel for
for (int y = 0; y < height; y++) {
    png_bytep row = row_pointers[y];
    for (int x = 0; x < width; x++) {
        png_bytep px = &(row[x * 4]);
        image[y][x].r = px[0];
        image[y][x].g = px[1];
        image[y][x].b = px[2];
    }
    free(row_pointers[y]);
}
```

</td>
<td>read_png_file</td>
</tr>
<tr>
<td>

```
#pragma omp parallel for collapse(2)
for (int x = 0; x < height; x++) {
    for (int y = 0; y < width; y++) {
        int kernelSize = kernelSizes[x][y];
        int kernelRadius = kernelSize >> 1;
        double sum = 0.0;
        double filteredPixel = 0.0;
        double bordercheck;

        sum =  kernelSize == 10 ? 121  :  25;
        int x1 = std::max(x - kernelRadius, 0);
        int y1 = std::max(y - kernelRadius, 0);
        int x2 = std::min(x + kernelRadius, height - 1);
        int y2 = std::min(y + kernelRadius, width - 1);

        bordercheck  =  (x2 - x1 + 1) * (y2 - y1 + 1);

        if(bordercheck!=sum){

            for (int i = -kernelRadius; i <= kernelRadius; i++) {
                for (int j = -kernelRadius; j <= kernelRadius; j++) {
                    int pixelX = std::min(std::max(x + i, 0), height - 1);
                    int pixelY = std::min(std::max(y + j, 0), width - 1);
                    filteredPixel += computeSum(prefixSum, pixelX, pixelY,pixelX, pixelY);
                }
            }
        }
        else{
            filteredPixel = computeSum(prefixSum, x1, y1,x2,y2);
        }
        output[x][y] = static_cast<int>(filteredPixel / sum);
    }
}
}
```

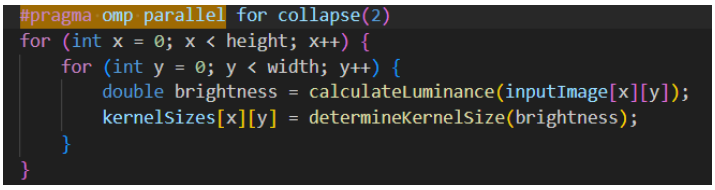
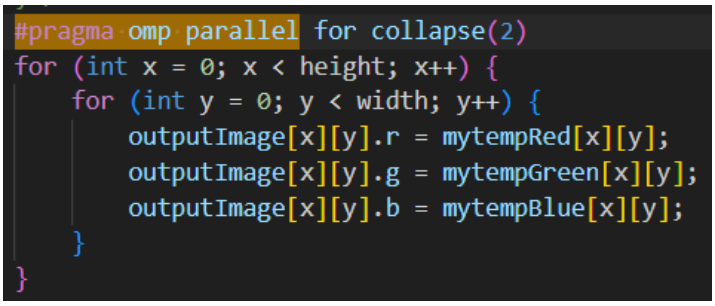
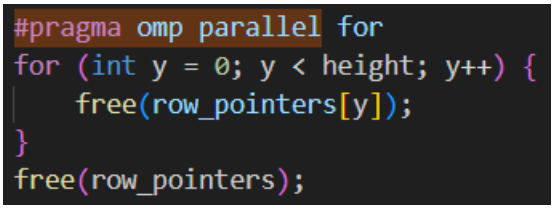
</td>
<td>myApplyFilterToChannel

分別計算 filteredPixels，使用 collapse 將雙層迴圈攤開，能以元素為單位進行分工。</td>
</tr>
<tr>
<td>

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        calculatePrefixSum(inputImage, redPrefix,0,height,width);
    }
    #pragma omp section
    {
        calculatePrefixSum(inputImage, greenPrefix,1,height,width);
    }
    #pragma omp section
    {
        calculatePrefixSum(inputImage, bluePrefix,2,height,width);
    }

}
```

</td>
<td>adaptiveFilterRGB

使用 3 個 thread 計算 prefix sum，由於計算 prefix sum 元素之間不不獨立，因此不進行矩陣運算相關優化。</td>
</tr>
</table>

	adaptiveFilterRGB 分別計算 KernelSize，使用 collapse 將雙層迴圈攤開，能以元素為單位進行分工。
	adaptiveFilterRGB 將處理好的圖片資料搬到 output image。使用 collapse 將雙層迴圈攤開，能以元素為單位進行分工。
	write_png_file
	write_png_file

3. 輸出設定

進行完上述優化後，我將各階段耗時印出，如圖四。可以發現 bottle neck 在寫出的部分。

```
[u2380045@lgn303 hw1]$ srun -A ACD113026 -c8  ./hw1 cases/input_4.png output.png
read_png_file time: 0.254866 seconds
adaptiveFilterRGB time: 0.290820 seconds
write_png_file time: 4.223158 seconds
```

圖四、各階段耗時

因此我進行兩個設定，使輸出過程更快。

✓ png_set_compression_level(png,0);

這個指令代表不進行壓縮。

✓ png_set_filter(png, PNG_FILTER_TYPE_BASE, PNG_FILTER_NONE);

這個指令代表不進行濾波。

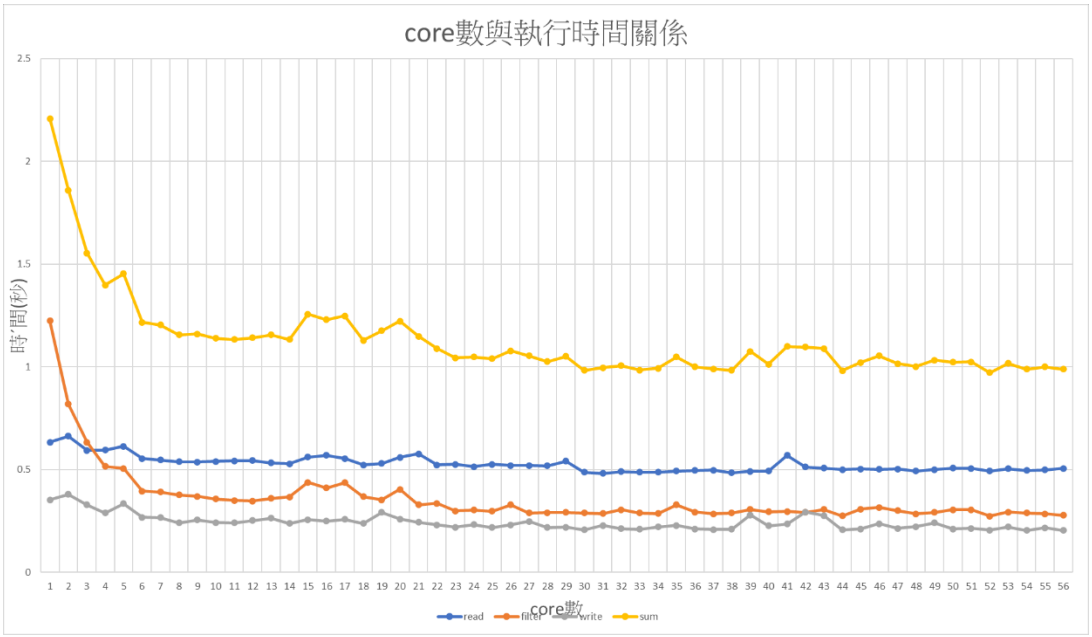
二、 效能分析

1. 核心數與效能分析

我以 testcase 3 做測試，繪製出圖五結果。可以看見，執行時間與 core 數並非線性關係，而是隨著 core 數增加，執行時間會收斂到一定值。這是因為根據 Amdahl's Law：

$$Speedup = 1/[(1-S)/P + S] \quad (2)$$

其中(1-S)是可平行化的部分，P 為核心數，即使 P 增加，(1-S)項趨近於零，仍會因為有不可平行化的部分，使速度無法增加。



圖五、執行時間與核心數關係，黃色為總時間、藍色是讀花費的時間、橘色是去造階段花費時間、灰色是寫出時間

2. **Ablation Study**

以下我以 testcase 3 做測試，計算各更動對加速的貢獻。由案例 2,4 可以看見優化演算法在平行化的情形還是有用的。

Index	Modified Algorithm	Parallelization	Modified Writing setting	Time(s)
1				7.34
2		★		4.14
3	★			3.78
4	★	★		3.18
5	★	★	★	1.62

表一、不同條件下對執行時間的影響

三、 Pthread 與 OpenMP 比較

- Pthread

  優點：

  (1) Pthread 可以更精準控制每條 thread 的創建、同步和終止

  (2) 指令相較底層，容易除錯

  缺點

  (1) 程式碼不具彈性，無法依不同機器進行適當的 thread 分配

  (2) 需要較多程式碼以達成平行化，容易寫錯

- OpenMP

  優點：

  (1) 程式碼簡單，容易使用

  (2) 可根據核心數靈活擴充 thread 數

  缺點

  (1) 平行化實作依靠編譯器執行，效能受編譯器影響。

  (2) Thread 控制細節度不如 Pthread

四、 困難

我想這份作業相對簡單，製作時並沒有遇到太大的困難。剛看到作業時，我第一個想法是把能平行化的迴圈全部平行化。之後，才著手進行演算法的改造。我一開始透過肉眼檢查圖片覺得很正常，使用 judge 跑，有幾筆是 accepted 有幾筆是 wrong answer，一直思考到底是哪裡出問題。後來我先把平行化拿掉，把原本的處理方法輸出跟我的方法比較來 debug。

將演算法改造和平行化後，我看到記分板第一名誇張的速度，便仔細思考還有哪裡可以優化，便把各步驟的花費時間印出，發現輸出圖片佔大多數時間，所以想著手改進輸出階段。然而，該部分使用函式庫撰寫，我也不太改更動，查詢 PNG 的壓縮方式，也覺得不太能平行化。之後發現可以更改壓縮程度，進行更動後，速度快上許多，後來又發現 PNG 編碼還有濾波器可以進行設定，又讓速度更快了。