JPEG 解碼器專案報告

一、專案概述

本專案實作了一個 Baseline JPEG 解碼器，能夠將 `.jpg` 檔案解碼並輸出為 `.bmp` 格式的點陣圖。整體流程遵循 JPEG 標準，依序解析各段（Segment），最終透過反量化、反 Zigzag、IDCT 還原影像像素。

二、解碼流程

1. 讀取 JPEG 標記（Marker）

JPEG 檔案由多個段落組成，每個段落以 `0xFF` 開頭，後接一個標記碼。主要段落包括：

標記碼	名稱	功能
`0xD8`	SOI	Start of Image，檔案起始
`0xE0`	APP0	應用程式資訊（如 JFIF）
`0xDB`	DQT	定義量化表
`0xC0`	SOF	定義影像尺寸與採樣因子
`0xC4`	DHT	定義霍夫曼表
`0xDA`	SOS	開始掃描，進入熵編碼資料區
`0xD9`	EOI	End of Image，檔案結束

程式透過 `readStream()` 函式依序讀取各標記，並呼叫對應的解析函式。

2. 量化表（DQT）

`readDQT()` 讀取 8×8 量化矩陣，用於後續反量化。每個係數會乘上對應的量化值還原原始 DCT 係數。

3. 影像資訊（SOF）

`readSOF()` 讀取影像高度、寬度，以及各顏色分量（Y、Cb、Cr）的採樣因子與對應量化表 ID。

4. 霍夫曼表（DHT）

`readDHT()` 讀取 DC 與 AC 的霍夫曼表，建立查表結構供熵解碼使用。

5. 熵解碼（SOS + Data）

進入 SOS 段後，`readData()` 開始逐 MCU（Minimum Coded Unit）解碼：

1. **讀取 DC 係數**：透過霍夫曼解碼取得差分值，累加得到實際 DC。
2. **讀取 AC 係數**：透過 Run-Length 編碼解析 (zeros, value) 對，填入 64 個係數。
3. **反量化**：將係數乘上量化表值。
4. **反 Zigzag**：將一維順序還原成 8×8 矩陣。
5. **IDCT**：透過二維逆離散餘弦變換還原空間域像素值。
6. **YCbCr → RGB**：將色彩空間轉換為 RGB，寫入 BMP 檔案。

三、效能優化

優化前問題

原始版本使用 `std::map<std::pair<unsigned char, unsigned int>, unsigned char>` 儲存霍夫曼表，查表時需要：

1. 建立 `std::pair` 物件
2. 進行 `std::map::find()` 查詢（O(log n) 複雜度）
3. 若未找到則繼續累積位元重試

這導致每次霍夫曼匹配都有較高的函式呼叫與記憶體配置開銷。

優化後改進

1. 霍夫曼表結構改寫

將 `std::map` 替換為自訂的 `HuffTableEntry` 結構：

```
struct HuffTableEntry {
    unsigned int startCode[17];   // 每個長度的起始碼
    unsigned int endCode[17];     // 每個長度的結束碼
    unsigned int count[17];       // 每個長度有幾個符號
    std::vector<unsigned char> symbols[17]; // 各長度的符號陣列
};
```

優點：

- 以陣列索引取代 map 查詢，存取複雜度從 O(log n) 降至 O(1)
- 預先計算 `startCode` 與 `endCode`，可快速判斷當前碼是否落在該長度範圍內
- 使用 `vector::assign()` 預配置空間，避免 `push_back` 的重複配置

2. 霍夫曼匹配邏輯優化

原始 `matchHuff()` 每次都要建立 `std::pair` 並呼叫 `map::find()`：

```
// 舊版
if (huffTable[ACorDC][number].find(std::make_pair(count, len)) != ..
```

```
    codeLen = huffTable[ACorDC][number][std::make_pair(count, len)];
    return codeLen;
}
```

新版改用範圍檢查：

```
// 新版
if (codeVal >= table.startCode[length] && codeVal <= table.endCode[l
    unsigned int offset = codeVal - table.startCode[length];
    return table.symbols[length][offset];
}
```

優點：

- 純整數比較，無物件建立開銷
- 提前排除不可能的長度（`count[length] == 0` 或範圍外），減少無效迭代

3. 量化表讀取修正

修正了一個筆誤：

```
// 舊版（錯誤）
t == t << 8;

// 新版（正確）
t = t << 8;
```

這個 bug 會導致高精度量化表讀取錯誤。

4. 餘弦快取（原有優化）

IDCT 使用預計算的 `cos_cache[200]` 陣列，避免每次都呼叫 `cos()` 函式：

```
void init_cos_cache() {
    for (int i = 0; i < 200; i++) {
        cos_cache[i] = cos(i * M_PI / 16.0);
    }
}
```

並採用分離式二維 IDCT（先對列做一維 IDCT，再對行做一維 IDCT），將複雜度從 $O(n^4)$ 降至 $O(n^3)$。

四、效能測試結果

使用 `time` 指令測量：

```
0.37user 0.01system 0:00.38elapsed 98%CPU
```

- **user time（0.37s）**：CPU 在使用者態執行的時間，主要是演算法計算
- **system time（0.01s）**：系統呼叫時間，佔比極低（約 2.6%）
- **CPU 使用率 98%**：幾乎沒有 I/O 等待

這表示目前的瓶頸在於計算本身（IDCT、色彩轉換等），而非檔案 I/O。

與舊版比較

版本	執行時間
舊版（map-based）	~725 ms
新版（array-based）	~400 ms
ffmpeg（參考）	~30 ms

優化後速度提升約 **45%**。

五、程式碼架構

```
main.cpp
├── readStream()           // 主迴圈，依標記分派
│   ├── readAPP()          // 解析 APP0/JFIF
│   ├── readDQT()          // 解析量化表
│   ├── readSOF()          // 解析影像資訊
│   ├── readDHT()          // 解析霍夫曼表
│   └── readSOS() + readData()  // 熵解碼主體
│       ├── readMCU()      // 讀取一個 MCU
│       │   ├── readDC()   // 讀取 DC 係數
│       │   └── readAC()   // 讀取 AC 係數
│       ├── MCU::decode()  // 反量化 + 反 Zigzag + IDCT
│       └── MCU::toRGB()   // 色彩轉換
└── main()                 // 進入點
```

七、結論

本專案成功實作了 Baseline JPEG 解碼器，並透過將霍夫曼表從 `std::map` 改為陣列結構、預計算範圍邊界等優化，將執行時間從約 725 ms 降至約 400 ms，提升約 45%。後續仍有進一步

優化空間，如位元緩衝、快速 IDCT 等，可望進一步縮短與 ffmpeg 的差距。