-Parallel Programming Final Project-

Smith Waterman

組別: 第三組
組員: 彭嘉洛、李浩榮

報告大綱

1. 演算法介紹 Algorithm Introduction
2. 演算法 Pseudo Code and Tracing
3. 實驗方法 CPU and GPU
4. 實驗結果 Experiment Results
5. 結論 Conclusions

一、演算法介紹

Algorithm Introduction

演算法介紹

動機

通過比對一個基因和蛋白質序列從而希望能夠找出兩者是否存在同源性。通過了解不同生物體中的關聯，而進一步理解生命和進化過程。

簡介

- 史密斯－沃特曼演算法（Smith-Waterman algorithm）是一種進行局部序列比對（相對於全局比對）的算法。
- 該算法的目的是找出兩個序列中具有高相似度的片段。
- 最常被利用於的例子為找出兩個核苷酸序列或蛋白質序列之間的相似區域。
- 是尼德曼－翁施演(Needleman-Wunsch)算法的一個變體，兩者皆為動態規劃的演算法。

二、演算法

Pseudo Code and Tracing

演算法具體實作流程

1. **Initialization 初始化**
    - **對第一個 row 和第一個 col 的值初始化成 0**
2. **Matrix Filling 執行填充陣列**
    - **根據penalty_gap, match, mismatch 的分數依序填充陣列**
3. **Traceback 回溯**
    - **通過根據第二階段得到的分數來源的dir_matrix，反推路線, 找出兩個序列相同對應的片段**

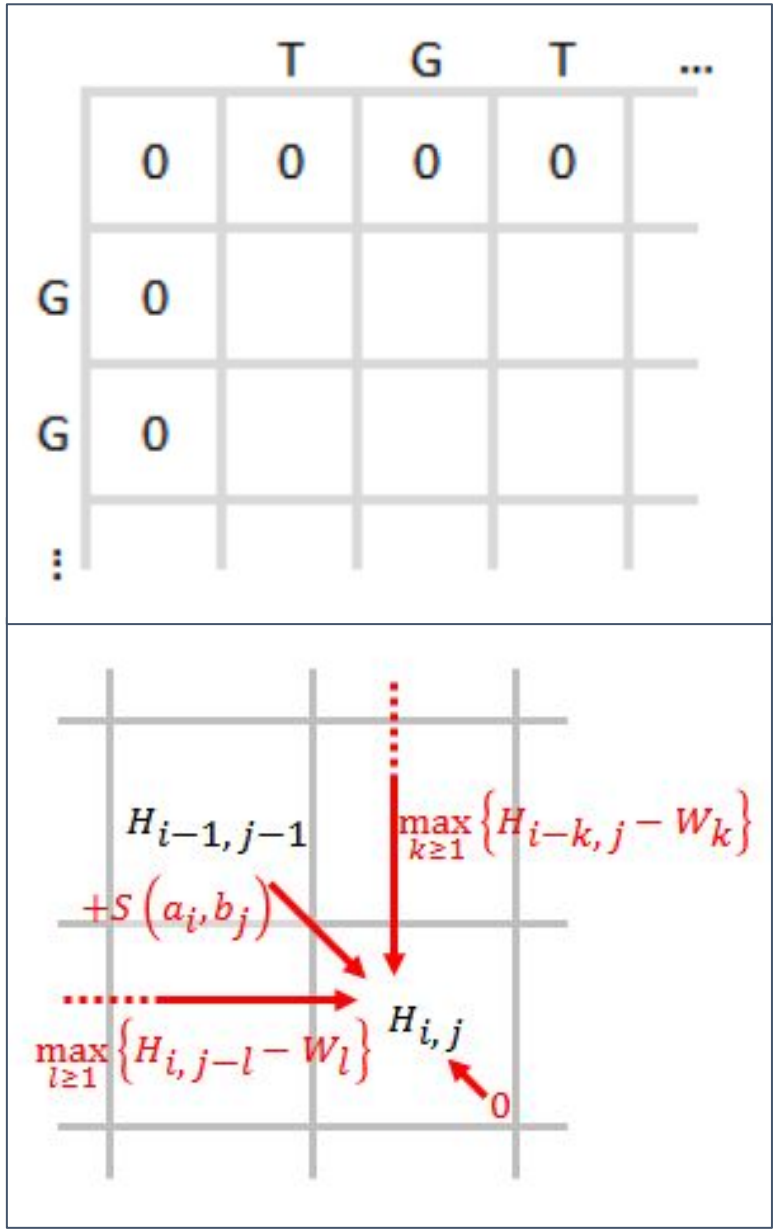
演算法 Pseudo Code

序列一： $A = a_1 a_2 \dots a_n$

序列二： $B = b_1 b_2 \dots b_m$

$$s(a_i, b_j) = \begin{cases} 1, & a_i = b_j \\ -1, & a_i \neq b_j \end{cases}$$

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - W_1, \\ H_{i,j-1} - W_1, \\ 0 \end{cases}$$



演算法 Pseudo Code

		A	T	G	C	T
	0	0	0	0	0	0
A	0					
G	0					
C	0					
T	0					

演算法 Pseudo Code

		A	T	G	C	T
	0	0	0	0	0	0
A	0	1				
G	0					
C	0					
T	0					

演算法 Pseudo Code

		A	T	G	C	T
	0	0	0	0	0	0
A	0	1	0			
G	0					
C	0					
T	0					

演算法 Pseudo Code

		A	T	G	C	T
	0	0	0	0	0	0
A	0	1	0	0	0	0
G	0	0	0	1	0	0
C	0	0	0	0	2	0
T	0	0	0	0	0	3

演算法 Pseudo Code

		A	T	G	C	T
	0	0	0	0	0	0
A	0	1	0	0	0	0
G	0	0	0	1	0	0
C	0	0	0	0	2	0
T	0	0	0	0	0	3

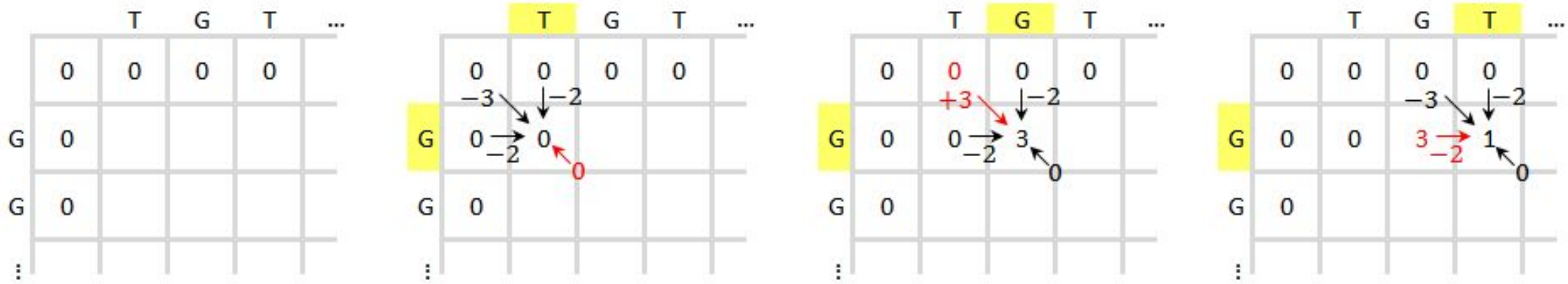
GCT

演算法 Pseudo Code

GCT

Locally Aligned Sequences

演算法 Pseudo Code



演算法 Pseudo Code

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

3	6	9	7	10	13
G	T	T	-	A	C
G	T	T	G	A	C

初始化得分矩阵

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0								
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

置换矩阵： $S(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

空位罚分： $W_k = kW_1$
$W_1 = 2$

三、實驗方法

CPU and GPU

實驗具體流程

1. 使用random隨機產生由A, T, G, C 所組成的兩個序列, 長度可自行設計。
2. 在CPU上allocate src_cpu, dir_cpu, 用於儲存陣列內對應的score和update direction。
3. 呼叫fill_cpu 或 fill_gpu 。
4. fill_gpu會讓GPU 做 cudaMalloc 和 cudaMemcpy 把CPU的兩個陣列儲存score 和 update direction 於 GPU上。
5. 展開至少diagonal length長度次數的for迴圈, 同步化更新每個反對角線上的值。
6. 用cudaMemcpy把memory從DevicetoHost做搬運, 在CPU上根據dir陣列做traceback, 顯示比對結果。

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0							
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3						
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1					
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0							
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3						
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1					
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

CPU 算法

用兩層for迴圈去走遍整個陣列。

具體流程是:

1)從左上角開始, 沿著row方向iter。

2)會先完成每個格子的正上面的值。

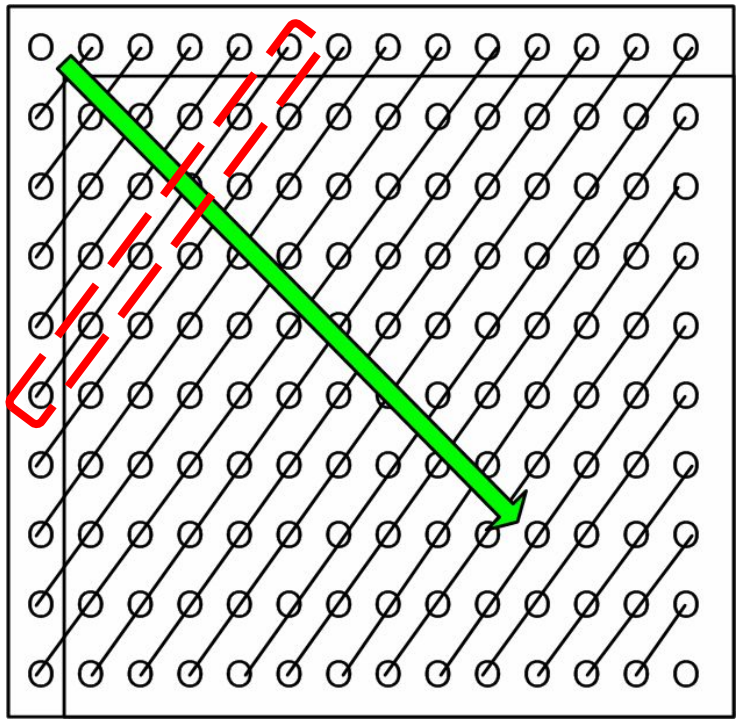
3)接著完成每個格子的左邊的值。

4)才算每個格子應選取的最大值。

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	4	8	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

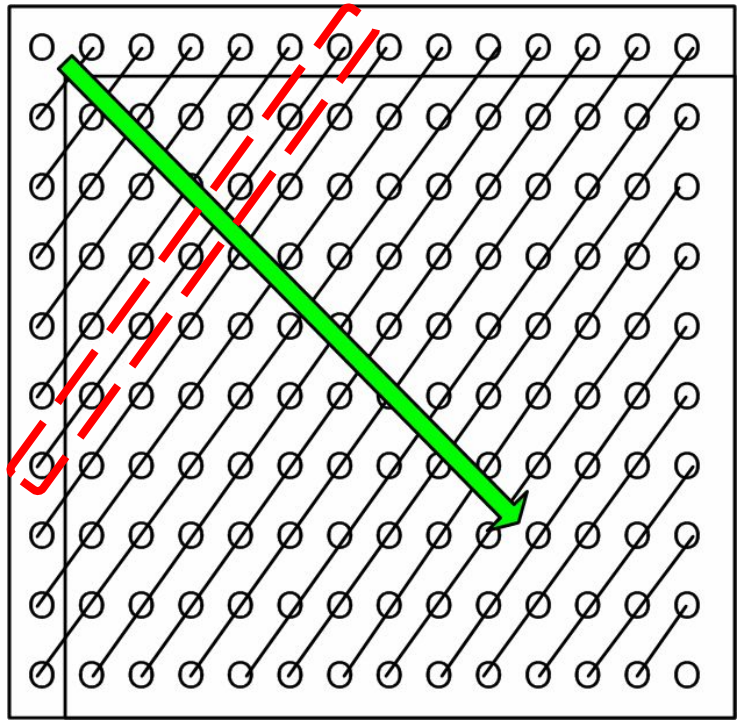
GPU 算法

- 每一個iteration有i個threads同時計算反對角線上的格子的值。
- traceback 就根據先前計算時記錄下來的assign direction 反推且印出對應的序列即可



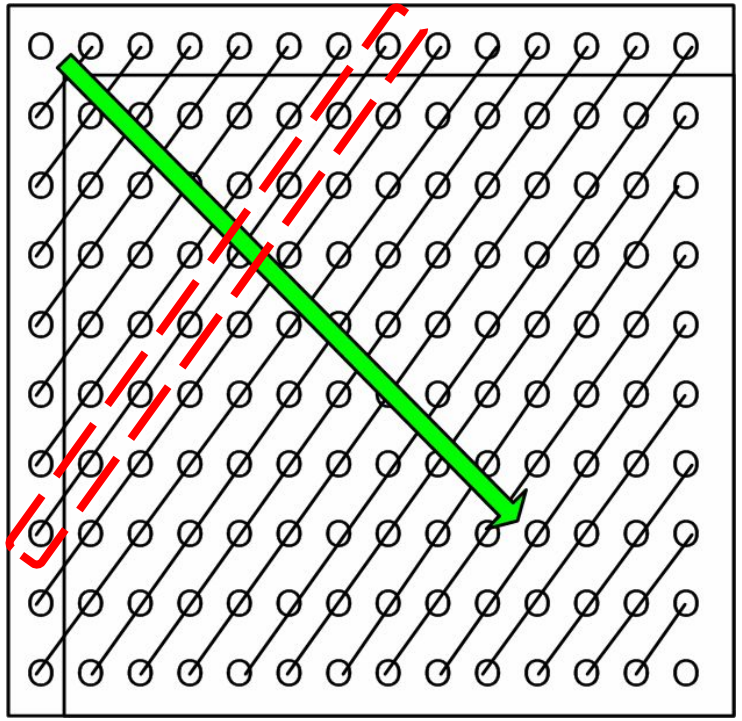
GPU 算法

- 每一個iteration有i個threads同時計算反對角線上的格子的值。
- traceback 就根據先前計算時記錄下來的assign direction 反推且印出對應的序列即可



GPU 算法

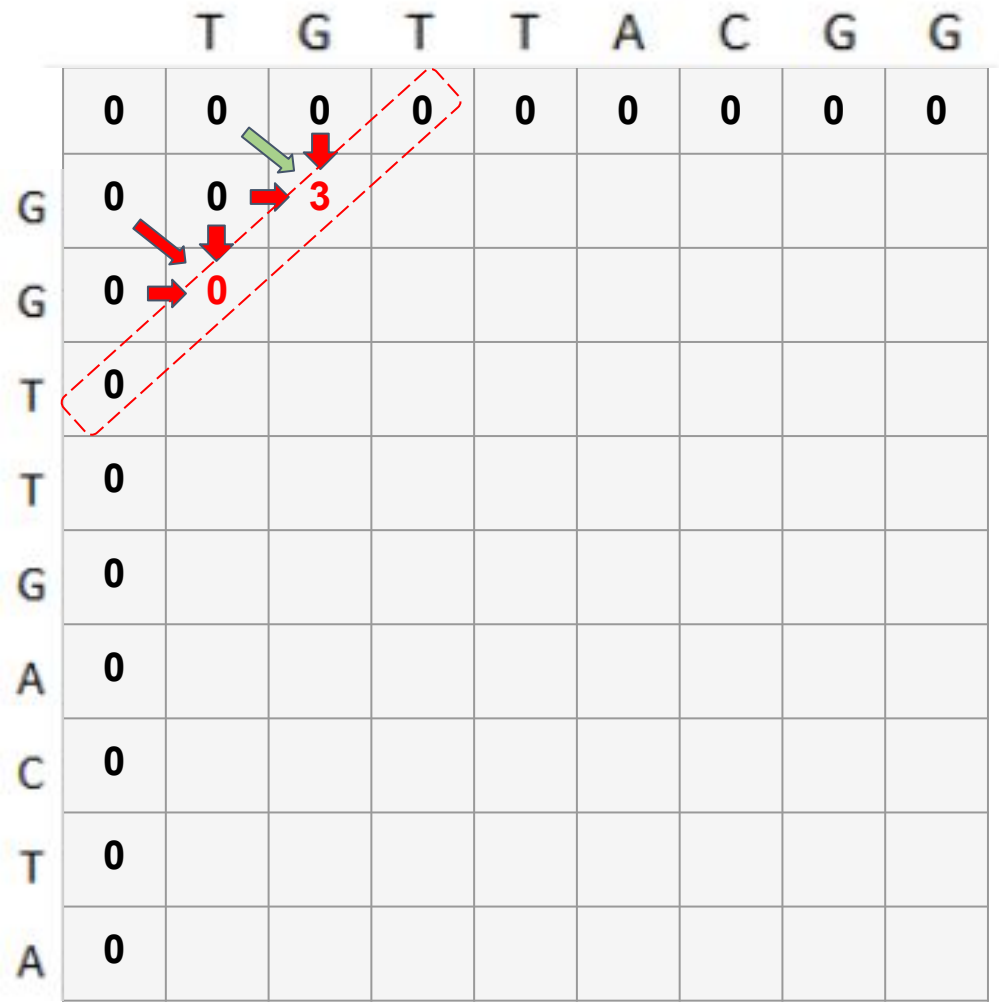
- 每一個iteration有i個threads同時計算反對角線上的格子的值。
- traceback 就根據先前計算時記錄下來的assign direction 反推且印出對應的序列即可



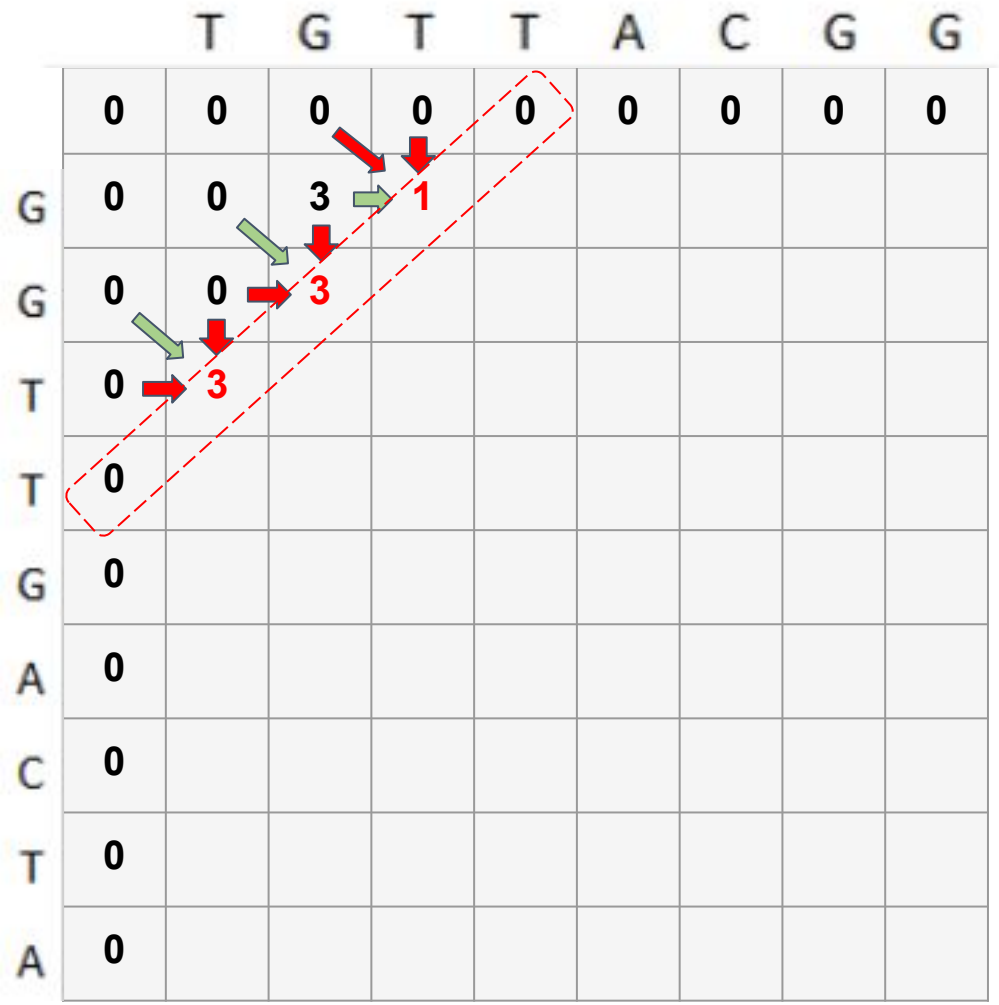
GPU 算法

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0							
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

GPU 算法



GPU 算法



GPU 算法

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	4	8	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

GPU 算法

- 每一個dimBlock內有同時有diag_len個threads一起同步計算

```
// loop over diagonals of the matrix
for (int i = 1; i <= ((A_LEN + 1) + (B_LEN + 1) - 1); i++) {
  int col_idx = max(0, (i - (B_LEN + 1)));
  int diag_len = min(i, ((A_LEN + 1) - col_idx));

  // launch the kernel: one block by length of diagonal
  int blks = 1;
  dim3 dimBlock(diag_len / blks);
  dim3 dimGrid(blks);
  fill_gpu<<<dimGrid, dimBlock>>>(d_h, d_d, d_seqA, d_seqB, i,
                                  d_max_id_val);
  cudaDeviceSynchronize();
}
```

四、實驗結果

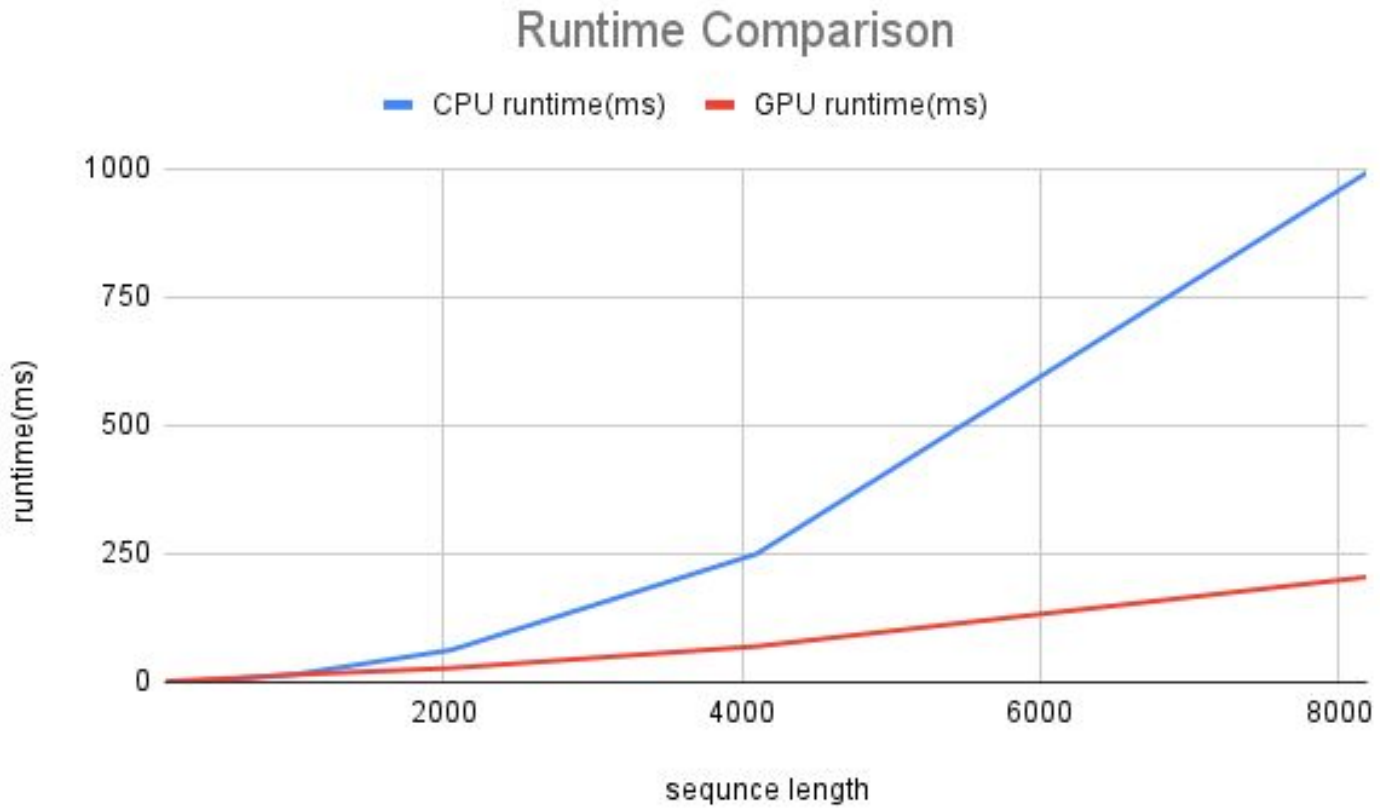
Experiments Results

Environment

- CPU: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
- GPU: GTX 1080TI

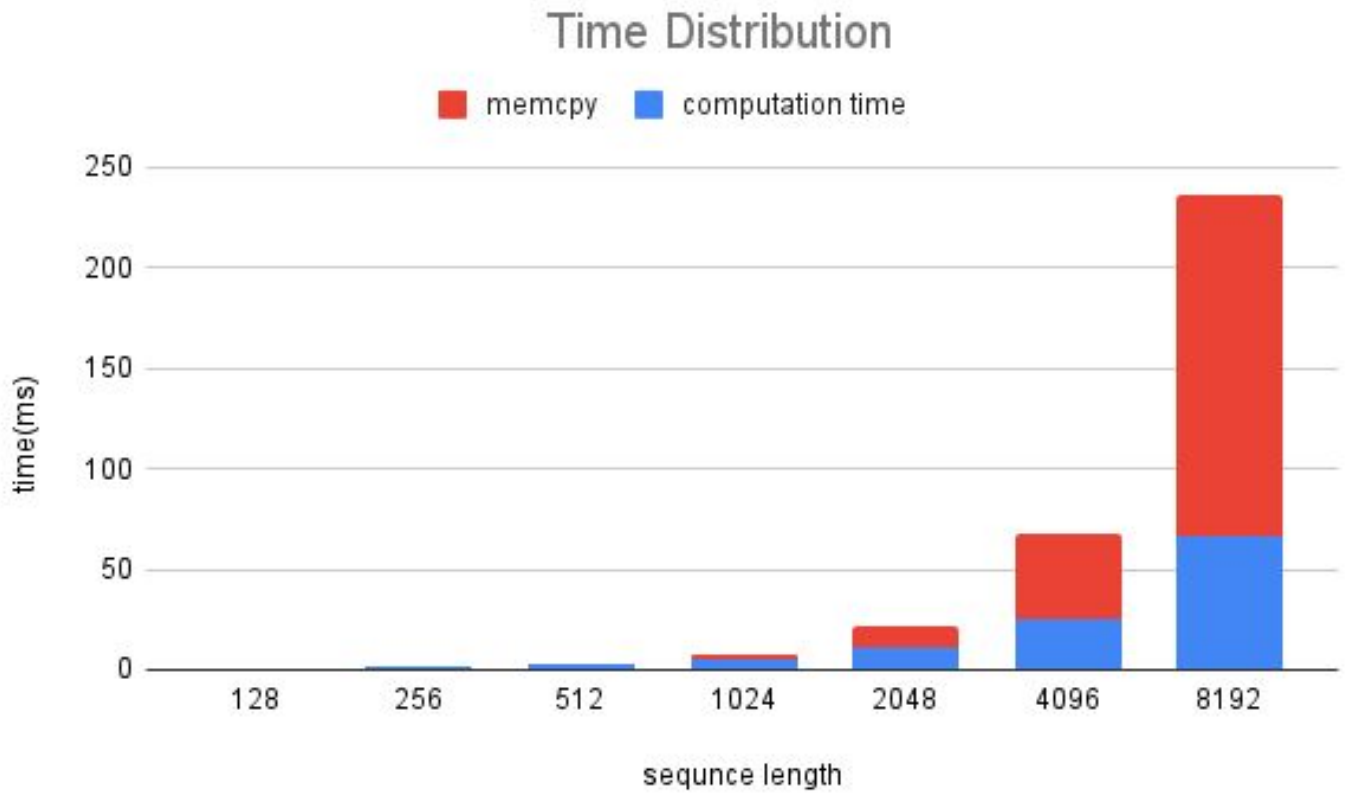
Runtime Comparison

sequnce length	CPU runtime(ms)	GPU runtime(ms)
128	0.702658	1.36467
256	1.05436	3.32832
512	3.95245	6.6968
1024	16.0134	14.8353
2048	62.5533	27.4593
4096	249.464	69.7115
8192	992.547	204.729



Time Distribution

sequnce length	computation time(ms)	memcpy(ms)
128	0.5429	0.03936
256	1.1233	0.16778
512	2.3177	0.64866
1024	4.8795	2.6266
2048	10.699	10.6043
4096	25.727	42.207
8192	66.358	169.38



- 當我們在收集不同 sequence length 的 computation time 時，發現在大於1024長度的情況下， nvprof 中的 fill_gpu 只會維持在被 call **2048** 次，且 computation time 卻會越來越快(但理論上 sequence 長度變長， 計算時間應該也要變長)。
- 我們後來就發現到， 當沿著對角線每次iteration向GPU要的threads數超過1024後，GPU就不執行這項requests的計算。所以從陣列左上角的沿著對角線方向 call 了GPU 1024次後，GPU就不會執行，直到靠近右下角的對角線長度為小於等於 1024的時候才會再執行， 因此主計算程式總共被call2048次。

GPU

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	4	8	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

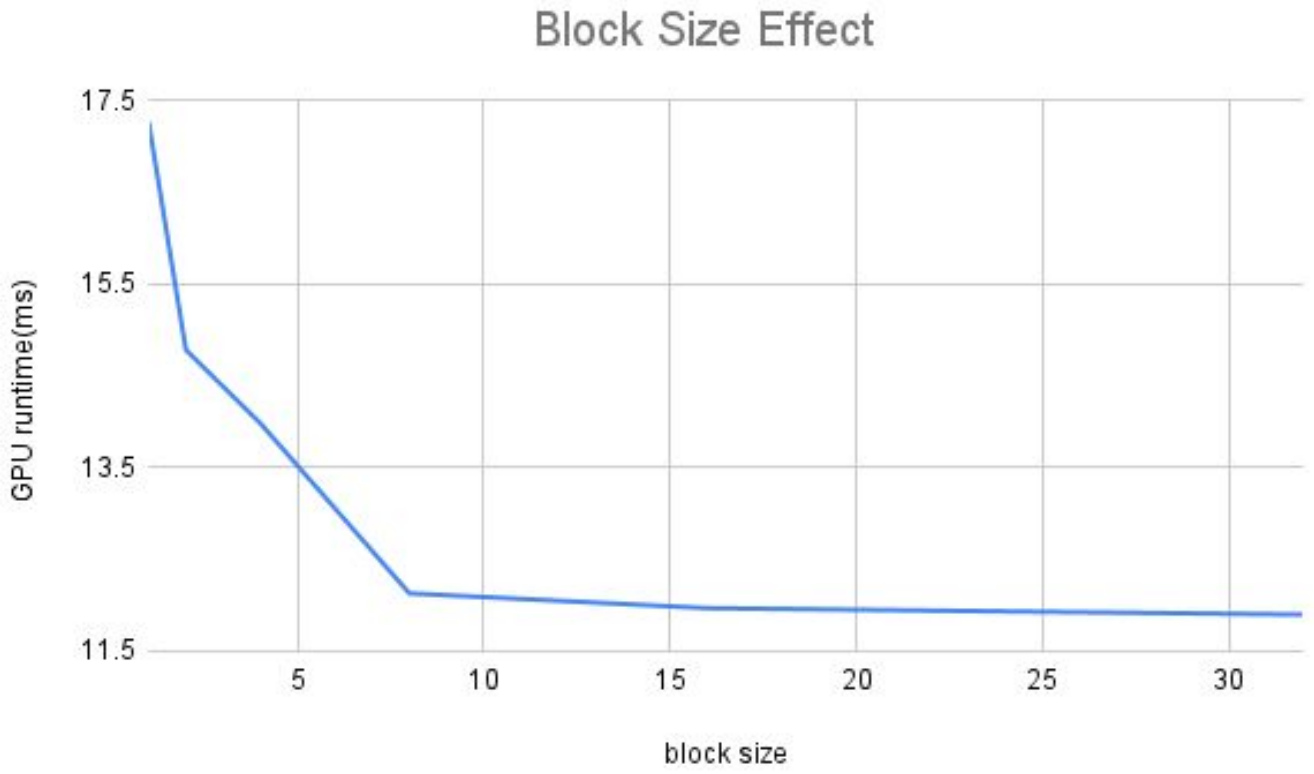
- 最大對角線的長度可能會超過GPU (GTX 1080TI 為例) 一次blocksize內能展開的threads數。
- GPU會直接忽略這部分的requests。

- 解決方法是增加 blocksize (blks)的數量，確保每一次request的threads數都不會超過上限值。

Block Size Effect

- sequence length = 1024

block size	GPU runtime(ms)
1	17.2672
2	14.7847
4	13.9844
8	12.1305
16	11.9692
32	11.8994



五、演示

Demo

Demo

五、結論
Conclusions

What we learn 學到了什麼

- 想要在把既有的演算法實作在GPU CUDA上平行化, 必然會遇到GPU硬體架構上設計的限制。因此更需要非常了解演算法中的過程, 在軟體上做程式的優化才能有效地最大化我們的演算法效益。
- CPU跟GPU的解法思路是截然不同的, 所以如果從GPU的角度去思考問題, 或許能得到一個新的解法, 兼具創新與效能。

-感謝聆聽-