Final Project Proposal

Smith-Waterman for Circular RNA Aligner

賴志杰、麥智詠、王晨峰

2022/06/22

https://github.com/CHIHCHIEH-LAI/HLS

Outline

- Smith-Waterman
- a first implementation
- systolic array architecture
- input compression
- shift register
- dual physical ports
- find max value index
- scalable design
- Future Work
    - Fully scalable design

Smith-Waterman Algorithm

設要比對的兩序列為$A = a_1 a_2 \ldots a_n$和$B = b_1 b_2 \ldots b_m$，其中n和m分別為序列A和B的長度。

1. 確定置換矩陣和空位罰分方法
   - $s(a, b)$ - 組成序列的元素之間的相似性得分
   - W_k - 長度為k的空位罰分
2. 建立得分矩陣H並初始化其首行和首列。該矩陣的大小為$n+1$行$m+1$列（注意計數從0開始）。

   $$H_{k0} = H_{0l} = 0, (0 \le k \le n, 0 \le l \le m)$$

3. 從左到右，從上到下進行打分，填充得分矩陣H剩餘部分

   $$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ \max_{k \ge 1}\{H_{i-k,j} - W_k\}, \\ \max_{l \ge 1}\{H_{i,j-l} - W_l\}, \\ 0 \end{cases} \qquad (1 \le i \le n, 1 \le j \le m)$$

   其中，

   $H_{i-1,j-1} + s(a_i, b_j)$表示將$a_i$和$b_j$比對的相似性得分，

   $H_{i-k,j} - W_k$表示a_i位於一段長度為k的刪除的末端的得分，

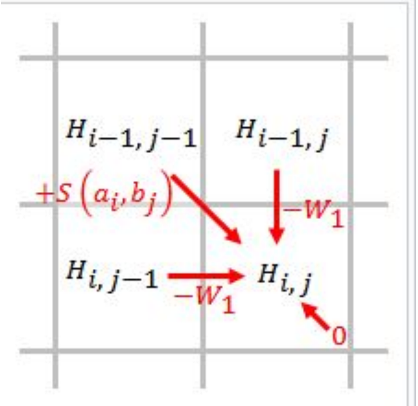
   $H_{i,j-l} - W_l$表示b_j位於一段長度為l的刪除的末端的得分，

   0表示a_i和b_j到此為止無相似性。

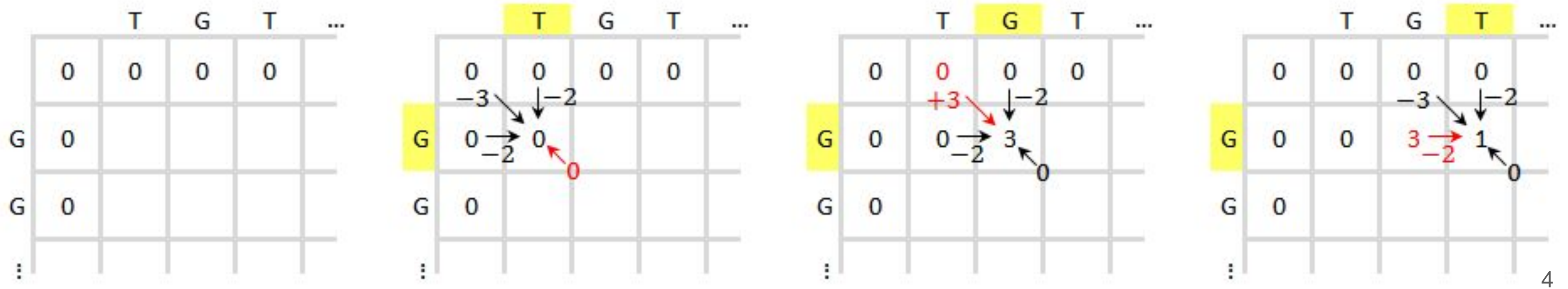
4. 回溯。從矩陣H中得分最高的元素開始根據得分的來源回溯至上一位置，如此反覆直至遇到得分為0的元素。

Smith-Waterman Score Matrix

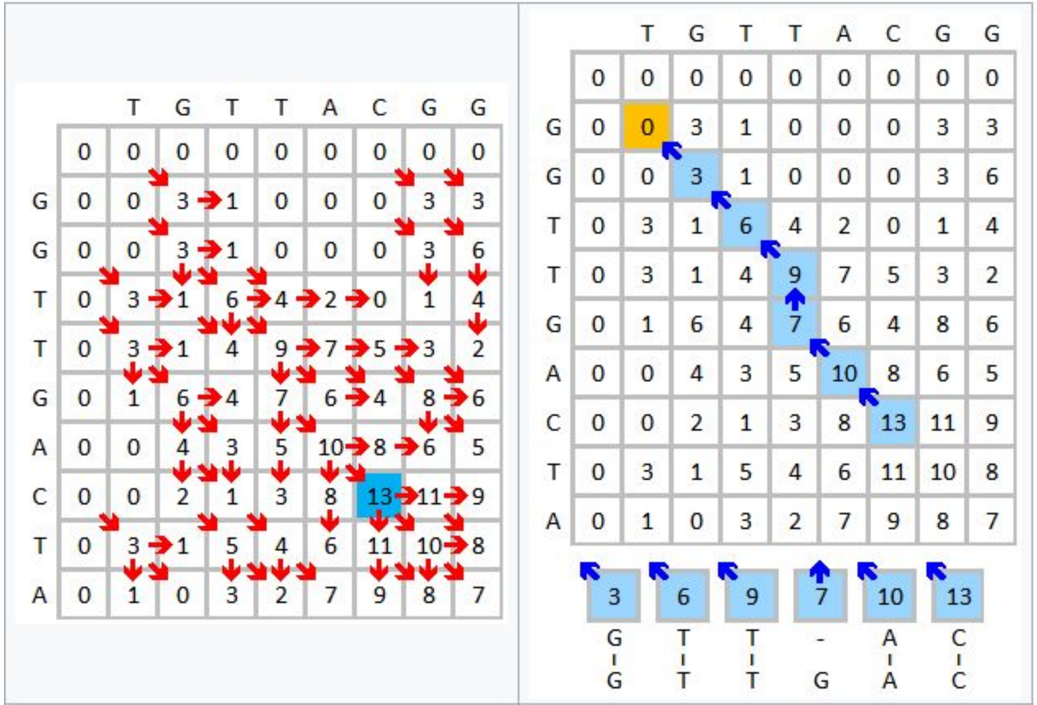
得分矩陣的作用是對兩序列中兩兩位置進行打分以逐步記錄最佳比對。



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



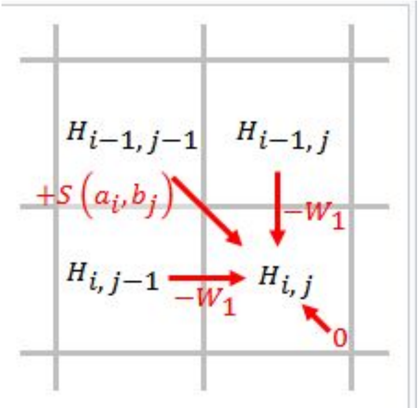
Smith-Waterman Traceback



完成後的得分矩陣及最高分（藍色）　回溯過程以及比對結果

constant

```
// scores used for Smith Waterman similarity computation
static const short GAP_i = -1;
static const short GAP_d = -1;
static const short MATCH = 2;
static const short MISS_MATCH = -1;
```



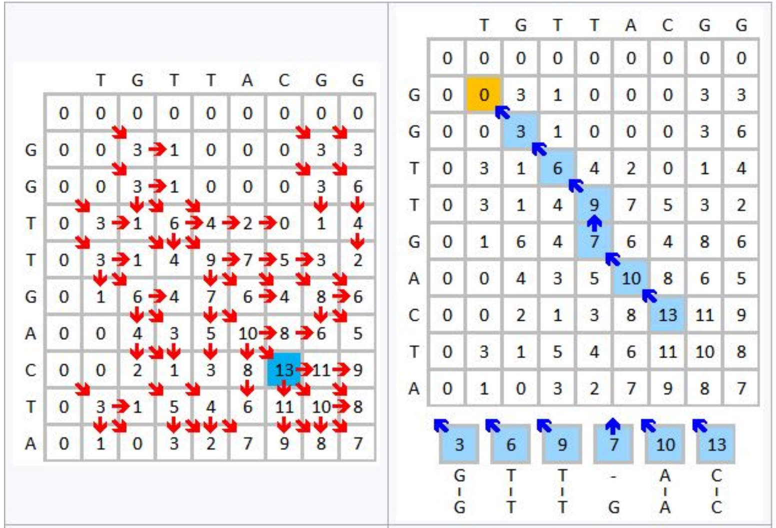
$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - W_1, \\ H_{i,j-1} - W_1, \\ 0 \end{cases} \qquad s(a_i, b_j) = \begin{cases} 1, & a_i = b_j \\ -1, & a_i \neq b_j \end{cases}$$

空位罰分: GAP_i = GAP_d = -w1 = -1

元素之間的相似得分: Match = +S(ai, bj) = 2 or MISSMATCH = -S(ai, bj) = -1

constant

```
// directions codes
static const int CENTER = 0;
static const int NORTH = 1;
static const int NORTH_WEST = 2;
static const int WEST = 3;
```

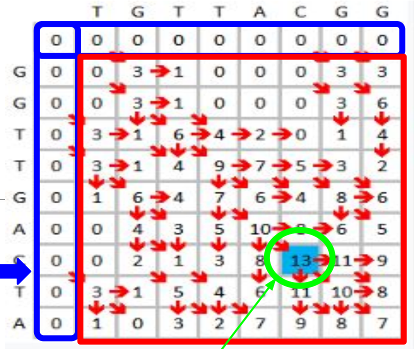


完成後的得分矩陣及最高分（藍色） | 回溯過程以及比對結果

用來記錄得分來源，並儲存結果於direction_matrix

A first implementation

```c
void compute_matrices(
    char *string1, char *string2,
    int *max_index, int *similarity_matrix, short *direction_matrix)
{
#pragma HLS INTERFACE m_axi port=string1 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=string2 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=max_index offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=similarity_matrix offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=direction_matrix offset=slave bundle=gmem

#pragma HLS INTERFACE s_axilite port=string1 bundle=control
#pragma HLS INTERFACE s_axilite port=string2 bundle=control
#pragma HLS INTERFACE s_axilite port=max_index bundle=control
#pragma HLS INTERFACE s_axilite port=similarity_matrix bundle=control
#pragma HLS INTERFACE s_axilite port=direction_matrix bundle=control

#pragma HLS INTERFACE s_axilite port=return bundle=control

    //here the real computation starts...
    int index = 0;
    int i = 0;
    int j = 0;
    short dir = CENTER;
    short match = 0;
    int val = 0;
    int north = 0;
    int west = 0;
    int northwest = 0;
    int max_value = 0;
    int test_val = 0;

    max_index[0] = 0;

    for(index = N; index < MATRIX_SIZE; index++) {
        dir = CENTER;
        val = 0;

        i = index % N; // column index
        j = index / N; // row index

        if(i == 0) {
            // first column
            west = 0;
            northwest = 0;
        } else {

            // all columns but first
            north = similarity_matrix[index - N];
            match = ( string1[i] == string2[j] ) ? MATCH : MISS_MATCH;

            test_val = northwest + match;
            if(test_val > val){
                val = test_val;
                dir = NORTH_WEST;
            }

            test_val = north + GAP_d;
            if(test_val > val){
                val = test_val;
                dir = NORTH;
            }

            test_val = west + GAP_i;
            if(test_val > val){
                val = test_val;
                dir = WEST;
            }

            similarity_matrix[index] = val;
            direction_matrix[index] = dir;
            west = val;
            northwest = north;
            if(val > max_value) {
                max_index[0] = index;
                max_value = val;
            }
        }
    }
}
```



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

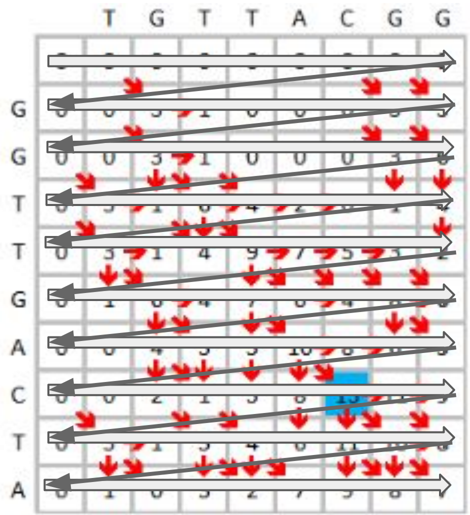
使用直觀演算法不做優化。

A first implementation

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices	II Violation	584786	1.949E6		584787		no	2	~0	0	0	2228	~0	3748	~0	0.00
VITIS_LOOP_53_1	II Violation	584714	1.949E6	148	145	4032	yes									

在最初版本中，每次只計算陣列中的一個值，共需要64x64個cycles，效率極低。

每個cycle中，讀取資料都要透過m-axi向memory讀取，因此需要每次cycle需要大量時間在資料傳輸。

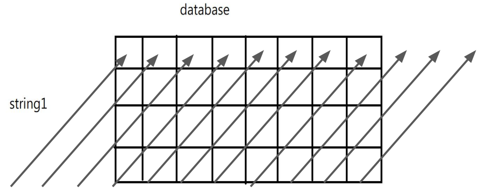


systolic array architecture

```
    char string1[N];
#pragma HLS ARRAY_PARTITION variable=string1 complete dim=1
    char string2[DATABASE_SIZE];
#pragma HLS ARRAY_PARTITION variable=string2 complete dim=1

//  short direction_matrix[DIRECTION_MATRIX_SIZE];
//#pragma HLS ARRAY_PARTITION variable=direction_matrix complete dim=1

    memcpy(string1, string1_g, N*sizeof(char));
    memcpy(string2, string2_g, DATABASE_SIZE * sizeof(char));

    int north[N+1];
#pragma HLS ARRAY_PARTITION variable=north complete dim=1
    int west[N+1];
#pragma HLS ARRAY_PARTITION variable=west complete dim=1
    int northwest[N+1];
#pragma HLS ARRAY_PARTITION variable=northwest complete dim=1

/*
    short directionDiagonal[N];
#pragma HLS ARRAY_PARTITION variable=directionDiagonal complete dim=1
*/

    ap_uint<512> compressed_diag[1];

    init_dep_for:for(int i = 0; i <= N; i++){
        north[i] = 0;
        west[i] = 0;
        northwest[i] = 0;
    }

    int directions_index = 0;

    num_diag_for: for(int num_diagonals = 0; num_diagonals < N + M - 1; num_diagonals++){
#pragma HLS inline region recursive
#pragma HLS PIPELINE

        calculate_diagonal(num_diagonals, string1, string2, northwest, north, west, directions_index, compressed_diag);
        store_diagonal(directions_index,  direction_matrix_g, compressed_diag);
        directions_index ++;
    }
```

先將由AXI-master傳輸的query和database儲存於FPGA上的string1與string2。

方向只有3種可能值，因此只需2bit即可表達。



由於反對角線方向的運算彼此不相關，因此可以同時進行。

systolic array architecture

計算三種可能的value

kernel寫2 bit值可用range
當Left_variable

```
void calculate_diagonal(int num_diagonals, char string1[N], char string2[DATABASE_SIZE],

    int databaseLocalIndex = num_diagonals;
    int from, to;
    from = N * 2 - 2;
    to = N * 2 - 1;

    calculate_diagonal_for: for(int index = N - 1; index >= 0; index --){
        int val = 0;

        unsigned int q = string1[index];
        unsigned int db = string2[databaseLocalIndex];

//      if(num_diagonals < N - 1 && databaseLocalIndex < N - 1 - num_diagonals) db = 9;

        const short match = (q == db) ? MATCH : MISS_MATCH;
        const short val1 = northwest[index] + match;
        const short val2 = north[index] + GAP_d;
        const short val3 = west[index] + GAP_i;
```

why loop starts from N-1?

取三種value最大者

```
        if(val1 > val && val1 >= val2 && val1 >= val3){
            //val1
            northwest[index + 1] = north[index];
            north[index] = val1;
            west[index + 1] = val1;
            compressed_diag[0].range(to,from) = NORTH_WEST;
//          directionDiagonal[index] = NORTH_WEST;
        } else if (val2 > val && val2 >= val3) {
            //val2
            northwest[index + 1] = north[index];
            north[index] = val2;
            west[index + 1] = val2;
            compressed_diag[0].range(to,from) = NORTH;
//          directionDiagonal[index] = NORTH;
        }else if (val3 > val){
            //val3
            northwest[index + 1] = north[index];
            north[index] = val3;
            west[index + 1] = val3;
            compressed_diag[0].range(to,from) = WEST;
//          directionDiagonal[index] = WEST;
        }else{
            //val
            northwest[index + 1] = north[index];
            north[index] = val;
            west[index + 1] = val;
            compressed_diag[0].range(to,from) = CENTER;
//          directionDiagonal[index] = CENTER;
        }
        databaseLocalIndex ++;
        from -= 2;
        to -= 2;
    }

}
```

systolic array architecture

```
void store_diagonal(int directions_index, ap_uint<512> *direction_matrix_g, ap_uint<512> compressed_diag[1]) {

     memcpy(direction_matrix_g + directions_index, compressed_diag, sizeof(ap_uint<512>));

}
```

```
short get(char data[], int key) {
    const int position = (key % 4) * 2;
    key /= 4;
    char mask = 0;
    mask &= 00000000;

    mask |= 3 << position;

    char fin_mask =0;
    fin_mask&= 00000000;

    fin_mask |= 3 << 0;
    return (((data[key] & mask) >> ( position)) & fin_mask);
}
```

Host 解讀2 bit 編碼的資料需透過shift與mask運算。

systolic array architecture

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		504	1.680E3		505		no	60	2	0	0	16389	~0	200829	23	0.00
Loop 1		167	557.000	73	1	95	yes									
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		128	427.000	3	1	127	yes									

優化內容:

- 先將由AXI-master傳輸的query和database儲存於FPGA上的string1與string2。
- 反對角線方向的運算 unroll後可以同時執行，最多可以有效計算 N個值。
- 需要回傳的directions經過 2 bit 壓縮編碼，可減少數據傳輸量。

input compression

```
void set_char_main(unsigned int  * array, int index, unsigned char val){
    switch(val){
        case 'A' : array[index / 16] |= (0 << ((index % 16) * 2));
        break;
        case 'C' : array[index / 16] |= (1 << ((index % 16) * 2));
        break;
        case 'G' : array[index / 16] |= (3 << ((index % 16) * 2));
        break;
        case 'T' : array[index / 16] |= (2 << ((index % 16) * 2));
        break;
    }

}
```

DNA有ATCG 4種鹼基對，RNA有AUCG 4種鹼基對。
無論分析DNA或RNA皆可只用 2 bit 表示一種鹼基對。

```
    ap_uint<512> string1[N / NUM_ELEM  + 1];
//#pragma HLS ARRAY_PARTITION variable=string1 complete dim=1
    ap_uint<512> string2[(DATABASE_SIZE)/ NUM_ELEM + 1];
//#pragma HLS ARRAY_PARTITION variable=string2 complete dim=1

//  short direction_matrix[DIRECTION_MATRIX_SIZE];
//#pragma HLS ARRAY_PARTITION variable=direction_matrix complete dim=1

    memcpy(string1, string1_g, (N/NUM_ELEM + 1) * 64);
    memcpy(string2, string2_g, ((DATABASE_SIZE) / NUM_ELEM + 1) * 64);
```

可節省空間與傳輸量

```
        const short q = string1[index/NUM_ELEM].range((index%NUM_ELEM) * 2 + 1, (index%NUM_ELEM) * 2);
        short db = string2[databaseLocalIndex/NUM_ELEM].range((databaseLocalIndex%NUM_ELEM) * 2 + 1, (databaseLocalIndex%NUM_ELEM) * 2);
```

在硬體中可以直接獲得指定位置的 bit value，
因此不需要額外時間做解碼。

input compression

優化前 (systolic array):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		504	1.680E3		505		no	60	2	0	0	16389	~0	200829	23	0.00
Loop 1		167	557.000	73	1	95	yes									
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		128	427.000	3	1	127	yes									

優化後 (systolic array + input compression):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		337	1.123E3		338		no	60	2	0	0	12307	~0	459216	52	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

優化內容:

將string1 與 string2 壓縮編碼，節省空間與傳輸量。

string1與string2壓縮後，長度僅為1個ap_uint<512>，因此Loop1(memcpy)縮短為2 cycle。

shift register

修改前:

```
    num_diag_for: for(int num_diagonals = 0; num_diagonals < N + M - 1; num_diagonals++){
#pragma HLS inline region recursive
#pragma HLS PIPELINE

        calculate_diagonal(num_diagonals, string1, string2, northwest, north, west, directions_index, compressed_diag);
        store_diagonal(directions_index,  direction_matrix_g, compressed_diag);
        directions_index ++;
    }
```

```
    int databaseLocalIndex = num_diagonals;
    int from, to;
    from = N * 2 - 2;
    to = N * 2 - 1;

    calculate_diagonal_for: for(int index = N - 1; index >= 0; index --){
        int val = 0;
        const short q = string1[index/NUM_ELEM].range((index%NUM_ELEM) * 2 + 1, (index%NUM_ELEM) * 2);
        short db = string2[databaseLocalIndex/NUM_ELEM].range((databaseLocalIndex%NUM_ELEM) * 2 + 1, (databaseLocalIndex%NUM_ELEM) * 2);
```

```
        databaseLocalIndex ++;
        from -= 2;
        to -= 2;
    }
```

原本string2的起始index隨著num_diagnals增加而增加，因此db存取的index會隨時間變化。

shift register

修改後:

```
    num_diag_for: for(int num_diagonals = 0; num_diagonals < N + M - 1; num_diagonals++){
#pragma HLS inline region recursive
#pragma HLS PIPELINE

        calculate_diagonal(num_diagonals, string1, string2, northwest, north, west, directions_index, compressed_diag, shift_db, direction_matrix_g);

        update_database(string2, shift_db, num_diagonals);
//      store_diagonal(directions_index,  direction_matrix_g, compressed_diag);
//      directions_index ++;
    }
```

```
    int databaseLocalIndex = 0;
    int from, to;
    from = N * 2 - 2;
    to = N * 2 - 1;

    calculate_diagonal_for: for(int index = N - 1; index >= 0; index --){
        int val = 0;
        const short q = string1[index/NUM_ELEM].range((index%NUM_ELEM) * 2 + 1, (index%NUM_ELEM) * 2);
        unsigned int db = shift_db[databaseLocalIndex/NUM_ELEM].range((databaseLocalIndex % NUM_ELEM) * 2 + 1, (databaseLocalIndex % NUM_ELEM) * 2);
```

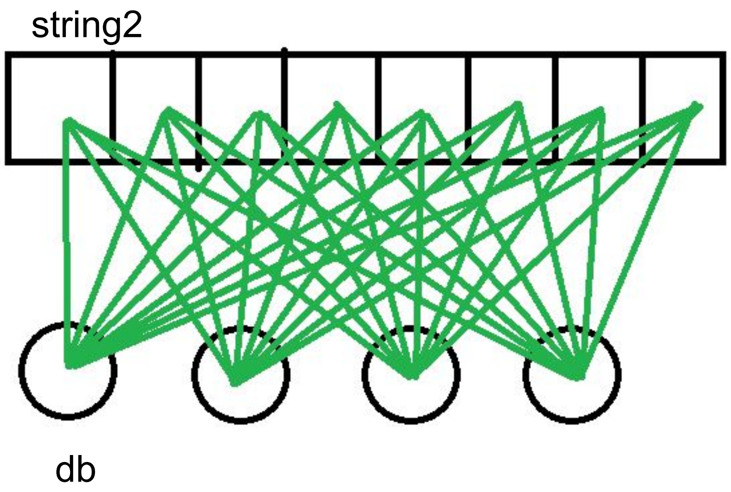
string2透過shift_db做媒介，起始index固定在0。
存取的index固定，不隨時間變化。

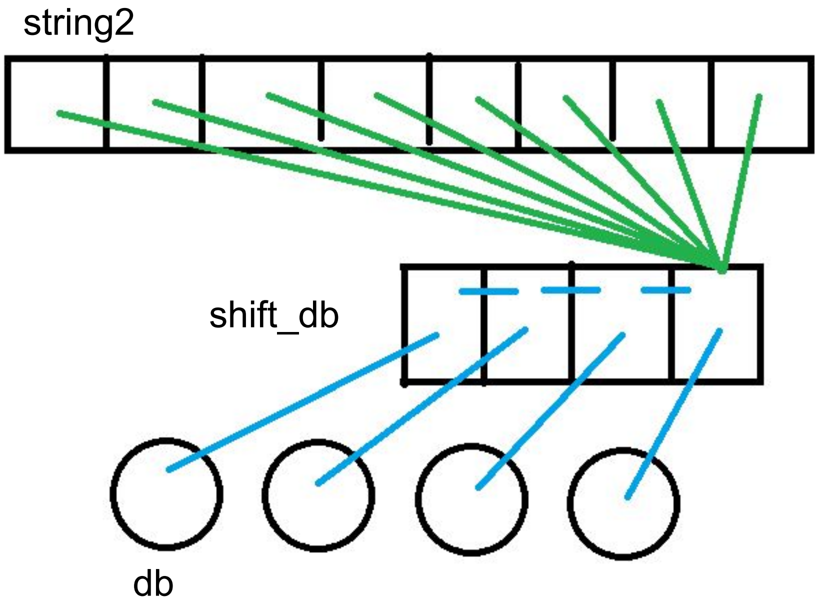
```
void update_database(ap_uint<512> *database, ap_uint<512> *shift_db, int num_diagonals){
    int startingIndex = N + num_diagonals;
    update_database:for(int i = 1; i < N; i++){
#pragma HLS PIPELINE
        shift_db[(i-1)/NUM_ELEM].range(((i-1)%NUM_ELEM)*2+1, ((i-1)%NUM_ELEM)*2) = shift_db[i/NUM_ELEM].range((i%NUM_ELEM)*2+1, (i%NUM_ELEM)*2);
        //set_char(shift_db, i-1, get_char(databaseLocal, i));
        //databaseLocal[i-1] = databaseLocal[i];
    }
    shift_db[(N-1)/NUM_ELEM].range(((N-1)%NUM_ELEM) * 2 + 1,((N-1)%NUM_ELEM) * 2) = database[startingIndex/NUM_ELEM].range((startingIndex%NUM_ELEM) * 2 +1, (startingIndex%NUM_ELEM) *2);
    //set_2bit(shift_db, N-1, get_2bit(database, startingIndex));
    //set_char(shift_db, N-1, get_char(database, startingIndex));
    //databaseLocal[N-1] = database[startingIndex];
}
```

shift_db作為一個滑動的窗口。

shift register

data傳輸網路圖





可看到數據之間的連線數大幅減少！

shift register

優化前 (systolic array + input compression):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		337	1.123E3		338		no	60	2	0	0	12307	~0	459216	52	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

優化後 (systolic array + input compression + shift register):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices	II Violation	25354	8.451E4		25355		no	60	2	0	0	19376	1	211583	24	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		25146	8.381E4	198		127	no									
calculate_diagonal_for		66	220.000	3	1	64	yes									
update_database	II Violation	126	420.000	2	2	63	yes									

可看到總Latency大幅增加！
update_database出現II violation (原先認為此處應該unroll)

shift register

```
void update_database(ap_uint<512> *database, ap_uint<512> *shift_db, int num_diagonals){
        int startingIndex = N + num_diagonals;
        update_database:for(int i = 1; i < N; i++){
#pragma HLS PIPELINE
                shift_db[(i-1)/NUM_ELEM].range(((i-1)%NUM_ELEM)*2+1, ((i-1)%NUM_ELEM)*2) = shift_db[i/NUM_ELEM].range((i%NUM_ELEM)*2+1, (i%NUM_ELEM)*2);
                //set_char(shift_db, i-1, get_char(databaseLocal, i));
                //databaseLocal[i-1] = databaseLocal[i];
        }
        shift_db[(N-1)/NUM_ELEM].range(((N-1)%NUM_ELEM) * 2 + 1,((N-1)%NUM_ELEM) * 2) = database[startingIndex/NUM_ELEM].range((startingIndex%NUM_ELEM) * 2 +1, (startingIndex%
        //set_2bit(shift_db, N-1, get_2bit(database, startingIndex));
        //set_char(shift_db, N-1, get_char(database, startingIndex));
        //databaseLocal[N-1] = database[startingIndex];
}
```

原因:
在update_database裡面出現 #pragma HLS PIPELINE，因此判斷是此pragma誤導Vitis HLS。
因為上層num_diag_for已經切pipeline，因此會盡量讓下層的updata_database平行或unroll，但是update_database下面多了#pragma HLS PIPELINE，使update_database的for loop無法被unroll，造成II vioilation。

解決方法:
我們將其註解調並重新合成。

shift register

優化前 (systolic array + input compression):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		337	1.123E3		338		no	60	2	0	0	12307	~0	459216	52	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

優化後 (systolic array + input compression + shift register):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		337	1.123E3		338		no	60	2	0	0	13706	~0	118187	13	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

可發現雖然Latency與先前完全相同，但因為 shift register很容易於硬體實現，也簡化了資料傳輸方向，因此資源消耗量大幅降低。

shift register

嘗試不同硬體規模:

原配置: M與N分別為128與256, 此大小需要較多資源且合成極慢 (需要數個小時)。

```
#define N 256
#define M 128
#define NUM_ELEM 256
#define DATABASE_SIZE (M + 2 * (N - 1))
#define DIRECTION_MATRIX_SIZE ((N + M - 1) * N)
#define MATRIX_SIZE (N * M)
```

若只修改M與N的定義, 則其電路無法正確運作。
經分析後, 發現code其他部分僅能在N = 256與M = 128的條件下成立。

因此我們將code改寫能夠適應各種N與M的配置。

dual physical ports

```
void compute_matrices( ap_uint<512> *string1_g, ap_uint<512> *string2_g, ap_uint<512> *direction_matrix_g)
{
#pragma HLS INTERFACE m_axi port=string1_g offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=string2_g offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=direction_matrix_g offset=slave bundle=gmem1
```

```
void compute_matrices( ap_uint<512> *string1_g, ap_uint<512> *string2_g, ap_uint<512> *direction_matrix_g)
{
#pragma HLS INTERFACE m_axi port=string1_g offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=string2_g offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=direction_matrix_g offset=slave bundle=gmem2
```

```
    memcpy(string1, string1_g, (N/NUM_ELEM + 1) * 64);
    memcpy(string2, string2_g, ((DATABASE_SIZE) / NUM_ELEM + 1) * 64);
```

原本string1_g與string2_g為同一個bundle，因此資料依序傳輸。
將string1_g跟string2_g設為不同的bundle以實現dual physical ports，同時傳輸兩者資料。

dual physical ports

優化前 (systolic array + input compression + shift register):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		337	1.123E3		338		no	60	2	0	0	13706	~0	118187	13	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

優化後 (systolic array + input compression + shift register + dual phsical ports):

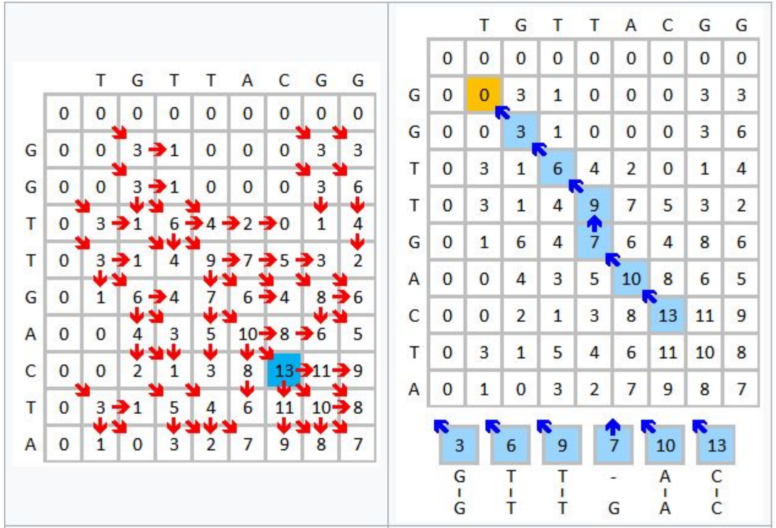
Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		336	1.120E3		337		no	90	3	0	0	15120	~0	119775	13	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

dual ports讓string1_g和string2_g可以同時傳，在本次實驗中 string1_g與string2_g長度皆為1個ap_uint<512>，原先需要2個cycle，優化後僅需要1個cycle。

雖然看似只省去僅僅一個 cycle，但在做scalable design時，string1_g和string2_g如果很大，整體省下的cycles還是很可觀的。

find max value index

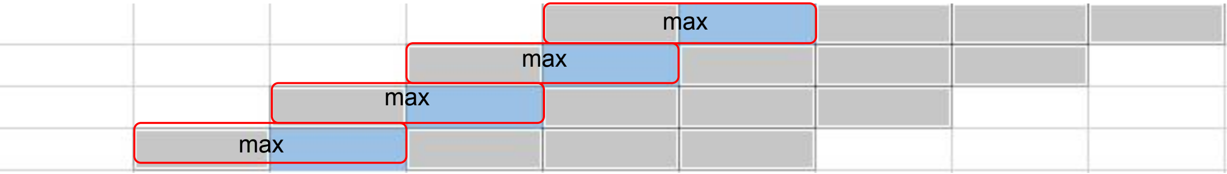
在做smith-waterman時，不僅需要要算direction maxtrix，還要算出哪裡有最大值，這樣才有辦法從最大值的那個點進行backtracking。



完成後的得分矩陣及最高分（藍色）

回溯過程以及比對結果

find max value index




```
    num_diag_for: for(int num_diagonals = 0; num_diagonals < N + host_M - 1; num_diagonals++){
#pragma HLS inline region recursive
#pragma HLS PIPELINE

        calculate_diagonal(num_diagonals, string1, string2, northwest, north, west, directions_index, compressed_diag, shift_db, direction_matrix_g);

        compare(num_diagonals, north, max_index, max_value);

        update_database(string2, shift_db, num_diagonals);
//      store_diagonal(directions_index,  direction_matrix_g, compressed_diag);
//      directions_index ++;
        if(rest_size > 0){
            if(num_diagonals - (N-1) - tail >= NUM_ELEM){
                memcpy(string2 + replace_idx%3, string2_g + replace_idx, 64);
                tail += NUM_ELEM;
                replace_idx++;
                rest_size--;
            }
        }
    }

    find_max(max_index, max_value);
    memcpy(max_value_idx, max_index, sizeof(int));
```

```
void compare(int num_diagonals, int north[N + 1], int max_index[N + 1], int max_value[N + 1]){
    for(int i=0;i<N;i++){
        if(north[i] > max_value[i]){
            max_index[i] = num_diagonals;
            max_value[i] = north[i];
        }
    }
}
```

針對每條對角線，我們會對此對角線與前一條對角線上N個值各別去做同位置的比大小，使得max_index都會存所有目前對角線上N個最大值的index。

find max value index

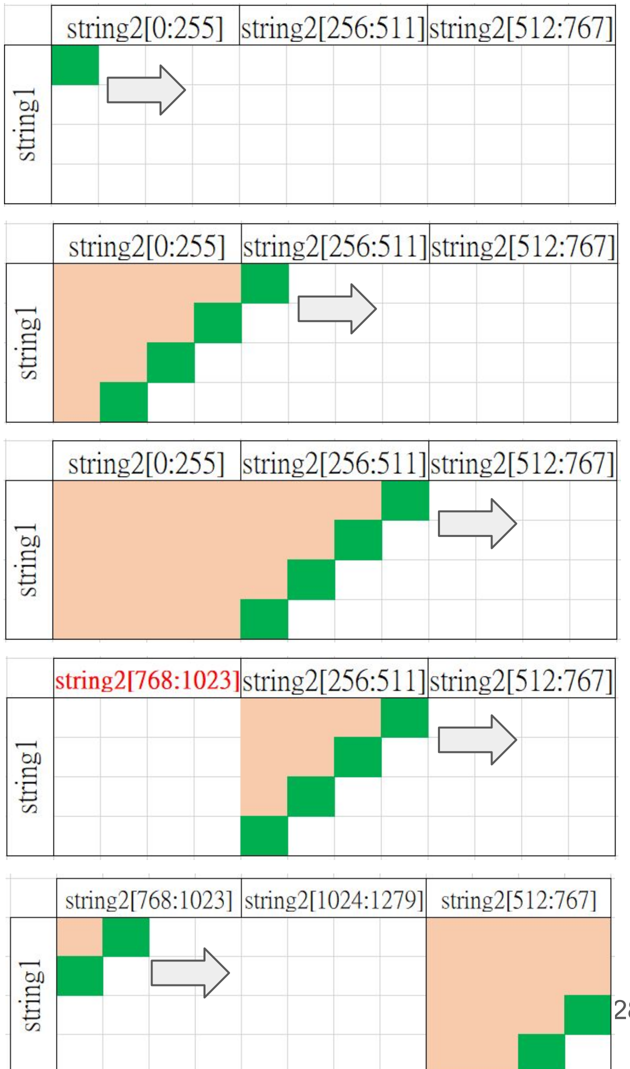
```
void find_max(int indexs[N + 1], int values[N + 1]){
    #pragma HLS PIPELINE

    int max_index_64[64], max_index_32[32], max_index_16[16], max_index_8[8], max_index_4[4], max_index_2[2];
    int max_value_32[32], max_value_16[16], max_value_8[8], max_value_4[4], max_value_2[2];

    for(int i=0;i<N;i++){
        max_index_64[i] = indexs[i] * N + i;
    }

    for(int i = 0; i<32; i++){
        if(values[i*2] >= values[i*2+1]){
            max_index_32[i] = max_index_64[i*2];
            max_value_32[i] = values[i*2];
        }else{
            max_index_32[i] = max_index_64[i*2+1];
            max_value_32[i] = values[i*2+1];
        }
    }

    for(int i = 0; i<16; i++){
        if(max_value_32[i*2] >= max_value_32[i*2+1]){
            max_index_16[i] = max_index_32[i*2];
            max_value_16[i] = max_value_32[i*2];
        }else{
            max_index_16[i] = max_index_32[i*2+1];
            max_value_16[i] = max_value_32[i*2+1];
        }
    }

    for(int i = 0; i<8; i++){
        if(max_value_16[i*2] >= max_value_16[i*2+1]){
            max_index_8[i] = max_index_16[i*2];
            max_value_8[i] = max_value_16[i*2];
        }else{
            max_index_8[i] = max_index_16[i*2+1];
            max_value_8[i] = max_value_16[i*2+1];
        }
    }

    for(int i = 0; i<4; i++){
        if(max_value_8[i*2] >= max_value_8[i*2+1]){
            max_index_4[i] = max_index_8[i*2];
            max_value_4[i] = max_value_8[i*2];
        }else{
            max_index_4[i] = max_index_8[i*2+1];
            max_value_4[i] = max_value_8[i*2+1];
```

計算完所有對角線後，我們會採用max tree，搭配pipleline，分成6級運算，以比較N個值中的最大值，並取得max value index，這樣傳回host就可以做backtracking。

scalable design

```
    int rest_size = (host_DATABASE_SIZE+NUM_ELEM-1)/NUM_ELEM - 3;
    int replace_idx = 3;
    int tail = 0;

    num_diag_for: for(int num_diagonals = 0; num_diagonals < N + host_M - 1; num_diagonals++){
#pragma HLS inline region recursive
#pragma HLS PIPELINE

        calculate_diagonal(num_diagonals, string1, string2, northwest, north, west, directions_

        compare(num_diagonals, north, max_index, max_value);

        update_database(string2, shift_db, num_diagonals);
//      store_diagonal(directions_index,  direction_matrix_g, compressed_diag);
//      directions_index ++;
        if(rest_size > 0){
            if(num_diagonals - (N-1) - tail >= NUM_ELEM){
                memcpy(string2 + replace_idx%3, string2_g + replace_idx, 64);
                tail += NUM_ELEM;
                replace_idx++;
                rest_size--;
            }
        }
    }
```

原本硬體的kernel_N與kernel_M需與string1與string2的長度一致。
而在真實的smithwaterman應用中，序列長度不固定。
我們透過實作circular buffer，將string2分段讀取，即可在固定的硬體資源運算任意長度的 string2 (database)。



final version

優化前 (systolic array + input compression + shift register + dual phsical ports):

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices		336	1.120E3		337		no	90	3	0	0	15120	~0	119775	13	0.00
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for		129	430.000	4	1	127	yes									

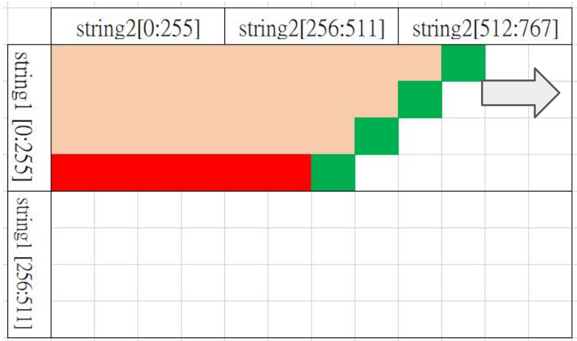
優化後 (systolic array + input compression + shift register + dual phsical ports + find max + scalable design):

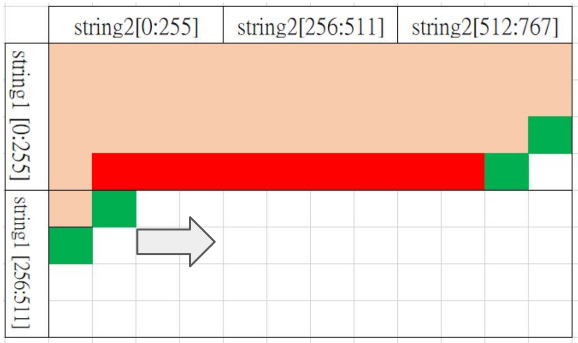
Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
compute_matrices							no	62	2	0	0	39165	2	181938	20	0.00
find_max_1		2	6.666		1		yes	0	0	0	0	4920	~0	7282	~0	
Loop 1		4	13.332	2	1	3	yes									
init_dep_for		65	217.000	1	1	65	yes									
num_diag_for				76	1		yes									

雖然scalable design比之前的design多出一倍多的運算資源，但是scalable design可以在不更改硬體的情況下，應付不同的database長度，且可算出max index。

Future Work

Fully scalable design:





需要使用到長度 = M - NCU的buffer儲存分割處的value。

(M = string2 length, NCU = number of compute unit)