**Allocating Memory:**

  int\* matrix;

    cudaMalloc((void\*\*)&matrix, sizeof(int)\*SEQUENCE\_LENGTH\*SEQUENCE\_LENGTH\*numSequences);

**Above we are allocating an array of matrices, a matrix of size SEQUENCE\_LENGTH\*SEQUENCE\_LENGTH for each pair of sequences.**

   int numThreadsPerBlock = SEQUENCE\_LENGTH;

**The block Dimension is the SEQUENCE\_LENGTH (maximum anti-diagonal size for active threads)**

    int numBlocks = numSequences;

    nw\_0 <<<numBlocks, numThreadsPerBlock>>> (**matrix**, sequence1\_d, sequence2\_d, scores\_d);

**Launching the kernel with numSequences blocks.**

**Kernel Implementation:**

\_\_global\_\_ void nw\_0(int\* matrix, unsigned char\* sequence1\_d, unsigned char\* sequence2\_d, int\* scores\_d) {

unsigned long long int base = SEQUENCE\_LENGTH\*SEQUENCE\_LENGTH\*blockIdx.x;

**Calculating the starting index of each matrix.**

unsigned int segment = SEQUENCE\_LENGTH\*blockIdx.x;

**Calculating the starting index of the current sequence**

        unsigned int tidx = threadIdx.x;

int row, col, top, left, topleft, insertion, deletion, match, max;

for(unsigned int i=0; i<SEQUENCE\_LENGTH; ++i)

**Looping across anti-diagonals (half of the output elements)**

{

if(tidx <= i)

**Ensuring that the only participating threads’ index is within the current anti-diagonal bounds**

{

**Calculating the row and column of the output score at matrix[row][col]**

row = i-tidx;

col = tidx;

**Special case for matrix[0][0] (since left, top, topleft are not within the matrix)**

if(col == 0 && row == 0) {

top = DELETION;

left = INSERTION;

topleft = 0;

}

**Special case for the first entry of each row in the matrix (since left and topleft are not within the matrix)**

else if(col == 0) {

top = matrix[base + SEQUENCE\_LENGTH\*(row-1) + col];

left = (row+1)\*INSERTION;

topleft = row\*INSERTION;

}

**Special case for the first entry of each column in the matrix (since top and topleft are not within the matrix)**

else if(row == 0) {

top = (col+1)\*DELETION;

left = matrix[base + SEQUENCE\_LENGTH\*row + (col-1)];

topleft = col\*DELETION;

}

**General case for internal elements**

else {

top = matrix[base + SEQUENCE\_LENGTH\*(row-1) + col];

left = matrix[base + SEQUENCE\_LENGTH\*row + (col-1)];

topleft = matrix[base + SEQUENCE\_LENGTH\*(row-1) + (col-1)];

}

**Calculating scores for later comparison**

insertion = top + INSERTION;

deletion = left + DELETION;

match = topleft;

if(sequence1\_d[segment + col] == sequence2\_d[segment + row]) {

match += MATCH;

}

else {

match += MISMATCH;

}

**Finding the maximum between insertion and deletion**

if(insertion > deletion) {

max = insertion;

}

else {

max = deletion;

}

if(match > max) {

max = match;

}

**Assign the maximum between match and max to the current output element**

matrix[base + SEQUENCE\_LENGTH\*row + col] = max;

}

**Synchronize across anti-diagonals**

\_\_syncthreads();

}

for(int i=SEQUENCE\_LENGTH-1; i>0; --i)

**Looping across the second part of the anti-diagonals**

{

if(tidx < i)

{

**Calculating the row and column of the output score at matrix[row][col]**

row = SEQUENCE\_LENGTH - tidx - 1;

col = SEQUENCE\_LENGTH + tidx - i;

**This time, top, left, and topleft are accessible from the matrix**

top  = matrix[base + SEQUENCE\_LENGTH\*(row-1) + col];

left = matrix[base + SEQUENCE\_LENGTH\*row + (col-1)];

topleft = matrix[base + SEQUENCE\_LENGTH\*(row-1) + (col-1)];

insertion = top + INSERTION;

deletion = left + DELETION;

match = topleft;

if(sequence1\_d[segment + col] == sequence2\_d[segment + row]) {

match += MATCH;

}

else {

match += MISMATCH;

}

if(insertion > deletion) {

max = insertion;

}

else {

max = deletion;

}

if(match > max){

max = match;

}

matrix[base + SEQUENCE\_LENGTH\*row + col] = max;

}

\_\_syncthreads();

}

**Only the thread at index 0 is allowed to save the final score.**

if(tidx == 0)

{

scores\_d[blockIdx.x] = matrix[base + SEQUENCE\_LENGTH\*(SEQUENCE\_LENGTH-1) + (SEQUENCE\_LENGTH-1)];

}

}