eMDPM: EFFIECIENT MULTI-DIMENSIONAL PATTERN MATCHING ALGORITHM FOR GPU

Supragya Raj V [1], Siddha Prabhu Chodnekar [2], Harish T [3] and Harini Sriraman[4]

School of Computing Science and Engineering, VIT University Chennai Campus, Chennai 600127, India
[1]Supragyaraj@gmail.com, [2]sprabhu.chodnekar@vit.ac.in, [3]harini.s@vit.ac.in

Abstract. Parallelizing pattern matching in multi-dimensional images is very vital in many applications to improve performance. With SIMT architectures, the performance can be greatly enhanced if the hardware threads are utilized to the maximum. In case of pattern matching algorithms, the main bottle neck arises due to the reduction operation that needs to be performed on the multiple parallel search operations. This can be solved by using Shift-Or operations. Recent trend has shown the improvement in pattern matching using Shift-Or operations for bit pattern matching. This has to be extended for multiple dimensional images like hyper-cubes. In this paper, we have extended the shift-or pattern matching for multi-dimensional images. The algorithm is implemented for GPU architectures. The complexity of the proposed algorithm is $m * \log(n)/\sigma w$ where m is the number of dimensions, n is the size of the array if the multi-dimensional matrix values are placed in a single dimensional array, σ is the size of the pattern and w is the size of the tile. From the result analysis it is found that the performance is max, when the pattern size matches the tile size and it is less than 64. This restriction is due to the size of the warp considered.

Keywords: Pattern Matching, Parallelism, GPU, Shift-Or Operations

1 INTRODUCTION

With advent of GPUs, execution time of image processing applications has greatly reduced. Pattern searching is an integral part of many applications. Any improvement in the performance of pattern matching of multi-dimensional image will greatly improve the performance of many GPU based applications. Pattern matching can be broadly classified into exact bit pattern matching and approximate bit pattern matching algorithms. The main bottle neck with respect to improving the performance pattern searching is reduction operation. In recent times, shift –or operation is used to improve the efficiency of reduction operation based pattern matching algorithms.

Parallel pattern matching involves tasks identification, communication identification, task agglomeration and mapping of tasks to processors. GPU is a SIMT architecture that can be utilized effectively for pattern matching algorithms. The matching of indi-

vidual patterns can happen in the stream processors available inside the stream multi-processors. The communication involves the reduction process. Agglomeration and mapping depends on the number of processors available. If we consider ‘n’ to be the number of primitive tasks and ‘p’ to be the number of processors, then n/p tasks can be assigned per processor.

2 RELATED WORK

Pattern matching involves the process of identifying all patterns in a given text. There are many practical applications for pattern matching algorithms. The pattern matching algorithms can be classified with respect to the dimensions of the data. Further classification of these algorithms include the type of pattern matching namely, exact pattern matching and approximate pattern matching.

2.1 EXACT AND APPROXIMATE PATTERN MATCHING

Exact pattern matching provides a solution by viewing each text position to be a possible pattern start. For exact pattern matching algorithm, the complexity is O(mn) where m*n is the dimension of the text matrix. Many improvements exist in the literature for exact pattern matching algorithms. Usage of linear automata [1] for the purpose is the most popular technique. This reduces the complexity of the pattern matching algorithm to be O(m+n). Good-suffix heuristic algorithm proposed in [2] provides an improved performance of O(n) with constant space requirement. In [3], bad-character rule strategy is generalized and improves the performance of the pattern matching algorithm with time complexity of O(n/m). Simple pattern searching algorithm proposed in [4] improves the average time complexity to be O(m+n).

Approximate parallel pattern matching algorithms works on the principle that a error can be tolerated by the pattern match. Say for example if a particular pattern match algorithm with pattern size of ‘m’ can tolerate ‘n’ error bits, approximate pattern matching algorithm will find out all the matches with ‘m to m-n’ matching bits. This matching will be particularly useful for bigger images. Parallelizing of the above categories of algorithms can be performed in different ways. This is highlighted in section 2.2.

2.2 PARALLEL BIT PATTERN MATCHING ALGORITHMS

Approximate and exact pattern matching algorithms falls into the category of finite and non-deterministic automata. Parallelizing these algorithms for better performance has evolved and there are many perspectives to this in the literature. Parallel algorithms that consider bit pattern mapping for parallelization is described in this section. The list of popular algorithms is listed in Table 1.

Table 1. Existing Categories of Bit Pattern Matching Algorithms

Technique	Type of algorithm
Shift-Or [9]	Non Deterministic Automata
Shift-And [10]	
Backward Non Deterministic Matching [11]	
Bit Parallel Wide window[12]	
Two Way shift OR[13]	
Reverse Factor [5]	Deterministic Finite Automata
Linear DAWG Matching[6]	
BSDM using Q -grams and shift-xor [7]	
Simplified Forward Backward Oracle Matching [8]	

3 PROPOSED ALGORITHM

Input: A text T of n_1 x n_2 and pattern P of m_1 x m_2
Output: A matrix X of size (n_1-m_1+1) x (n_2-m_2+1) indicating matching positions
Algorithm:

1. **for** i ← 1 to n_1-m_1+1
2. **for** j ← 0 to m_1-1
3. Initialize the masks M_1, M_2…. Mn_2 according to T[i+j] and P[j+1]
4. $A_0 \leftarrow \langle 0,1^m \rangle$
5. $A_k \leftarrow \langle 1, M_k \rangle$, for all $1 \le k \le n2$
6. **for** k← 0 to n_2 **do in parallel**
7. $R_k \leftarrow \sum_{l=0}^{k} A_l$ **(inclusive scan)**
8. **end for**
9. $\langle u_k, S_{j,k} \rangle \leftarrow R_{k,}$ for all $0 \le k \le n2$
10. **end for**
11. **for** j ← m_2 to n_2
12. $y_j \leftarrow S_{i,j} \mid S_{i+1,j} \mid \ldots\ldots \mid S_{i+m1-1,j}$ **(inclusive scan)**
13. $X[i][j-m_2+1] \leftarrow y_j[m_2]$
14. **end for**

end for

4 WORKING OF THE ALGORITHM

The working of the algorithm is explained here with an example. Let us take an image which is grayscale image of dimension 512 x 512 as shown in Fig 1. The pattern to be searched is also attached in Fig 1.

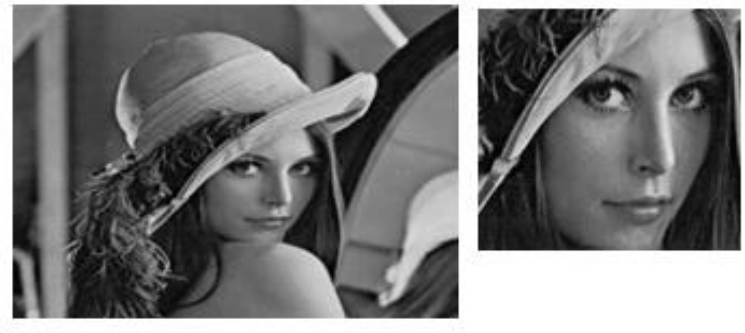

Fig. 1. Original Image and the Pattern Image

Using numerical values to represent the image, the original image is represented in X and the pattern in matrix Y. Final matrix X will be a 313 x 313 matrix of 0's and 1's denoting positions at which there is a match. X and Y matrices are shown in Fig. 2.

126	86	67	77	89
62	56	62	74	77
57	70	65	78	70
62	69	60	59	62
58	51	42	41	42

126	86	67
62	56	62
57	70	65

Fig 2. X matrix and Y Matrix

Let us consider first row of image as T (126, 86, 67, 77, 89) and first column of pattern as P (126, 86, 67). Objective of the algorithm is to find position of the matching pattern. The initialize masks according to T and P is given as follows.

$$\textit{Initial Mask: } M_1 = 110 \tag{1}$$

since, 1st pixel 126 matches with 1st pixel of pattern $M_1[1]=0$ and rest of the pixels don't match with 126 thus,

$$M_1[2]=1$$
$$M_1[3]=1$$

Similarly in this example,

$M_2=010$; $M_3=011$, $M_4=111$ and $M_5=111$

Then A_0, A_1 till A_5 will be initialized as per algorithm. R_k for all $0 \leq k \leq 5$ using inclusive scan with operator is found out. $x_{1,0}(111)$, $x_{1,1}(110)$, $x_{1,2}(101)$, $x_{1,3}(011)$, $x_{1,4}(111)$, and $x_{1,5}(111)$ are obtained. When MSB is 0 in any $x_{j,k}$ it means that there is a match of pattern with text. In the example, it could be seen that $x_{1,3}(011)$ which corresponds to match of (126,86,67) in pattern with (126,86,67). Similarly, in $x_{1,4}(111)$ MSB is 1 which corresponds to pattern (126,86,67) not matching with (86,67,77). For matching of row wise and column wise first 3 x 3 block of image with pattern we need $x_{2,3}$ and $x_{3,3}$. When we match second row of image T (62,56,79,74,77) with P (62,56,79), masks are M_0 (111), $M_1(010)$, $M_2(101)$, $M_3(010)$, $M_4(111)$ and $M_5(111)$. We get $x_{2,0}(111)$, $x_{2,1}(110)$, $x_{2,2}(101)$, $x_{2,3}(010)$, $x_{2,4}(111)$, and $x_{2,5}(111)$. Also, when we match third row we get $x_{3,0}(111)$, $x_{3,1}(110)$, $x_{3,2}(101)$, $x_{3,3}(011)$, $x_{3,4}(111)$, and $x_{3,5}(111)$. So, next we find value of y_3 as, $y_3 = x_{1,3} | x_{2,3} | x_{3,3} = 011 | 010 | 011 = 011$. These OR operations are done by inclusive scan. First value of final matrix X is obtained as, $X[1][1] = y_3[3]= 0$. This indicates that row wise and column wise first 3 x 3 block of image matches with the pattern exactly. If we try to match row wise second and column wise first 3 x 3 block of image we get $x_{1,3}(111)$, $x_{2,3}(111)$ and $x_{3,3}(111)$.

$$y_3 = x_{1,3} | x_{2,3} | x_{3,3} = 111 | 111 | 111 = 111$$

So $X[2][1] = y_3[3]= 1$. This indicates that row wise second and column wise first 3 x 3 block of image does not match with the pattern exactly. This algorithm goes on till all the blocks have been matched and result stored in X as shown in Fig. 3. As we can see there in only one 0 in the matrix which correctly shows that pattern matched at only one position in image. Consider a new 5 x 5 subsection as shown in Fig. 4 and its 3 x 3 pattern as shown in Fig. 5. Its matching matrix X is shown in Fig. 6. It can see that there are 0's are two positions which means correctly that pattern matches the

image at two positions (row wise first and column wise first block and row wise third and column wise third block).

0	1	1
1	1	1
1	1	1

Fig. 5. Matrix showing Result1

65	86	67	77	89
62	56	62	74	77
57	70	65	86	67
62	69	62	56	62
58	51	57	70	65

Fig .6. Matrix Representation of Image2

65	86	67
62	56	62
57	70	65

Fig. 7. Part Matrix Representation of Pattern 2

0	1	1
1	1	1
1	1	0

Fig. 8. Result Matrix 2

4 RESULTS AND DISCUSSION

The algorithm proposed in the previous section is implemented in a NVIDIA GT200 GPU machine. The host (CPU) will get the image input and pattern to search, from the user. A device global memory (GPU) is allocated and the input values are updated. The parallel process of pattern matching is initialized in GPU. The results are verified for better performance.

4.1 TILE BASED DATA DECOMPOSITION

Tile based data decomposition is used for improved performance in the GPU. Tiles of size w*w are shared among multiple SPs(Stream Processors) for processing .This improves the performance by reducing the redundant reads otherwise needed in the system. The parallelism is performed by considering the data are stored as a single linear array of values. The single linear array values are then loaded to the shared memory of the SP to be used by multiple threads of the same block. To synchronize the operation, after each tile read, a __synchthreads() function is called for performing a barrier like wait till all the threads in the same block has completed loading input to the tile. In the considered algorithm, this tile based data load has greatly improved the

performance of the system. The data update is done in the shared memory is shown in Fig. 9.

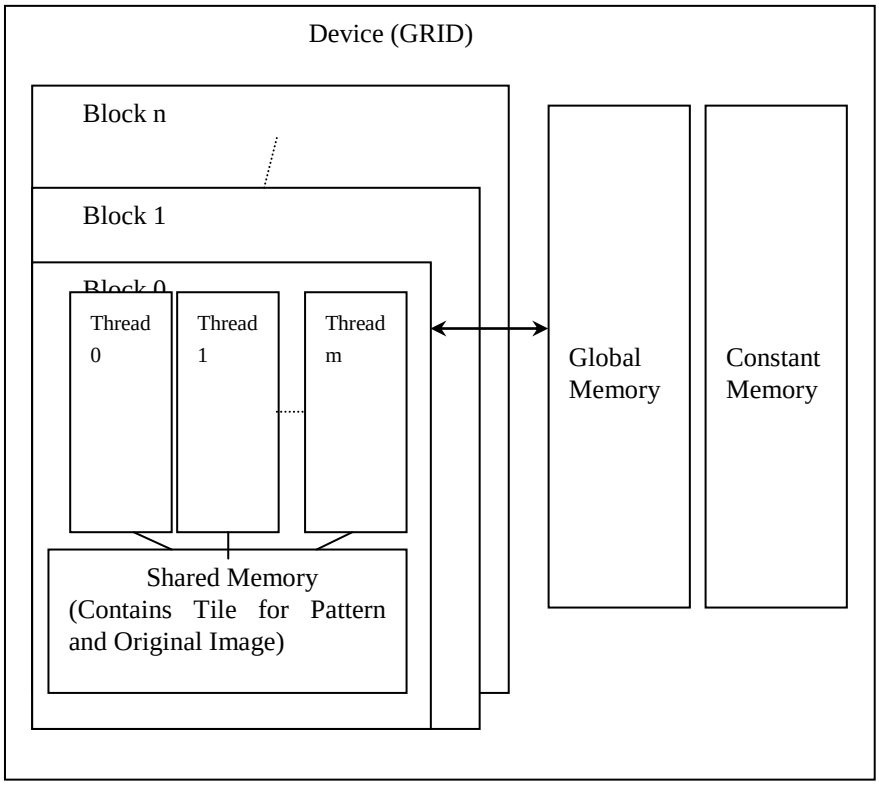

Fig. 9. Storing of Image and Pattern as Tile in Shared memory of GPU

4.2 RESULT ANALYSIS

For multi-dimensional images, pattern matching algorithms has plenty of possibility to parallelize. This helps in utilizing the GPU to the maximum and improves the performance of the algorithm. The main overhead with respect to parallelism is the increased communication overhead. But in the proposed implementation, the latency of this access is reduced by utilizing the shared memory. For the implemented system, the CGMA (Compute to Global Memory Access) ratio is 3:1. The compute for global memory access includes the following list of tasks.

- Shift-Or operation to perform pattern matching
- Or operation to update row-wise results of the tile
- Reduction with multiple rows

In the above case, row wise operations are performed with in thread of the tile. This can also be performed with respect to columns.

Let ‘m’ be the number of dimensions of the image considered, ‘n’ represents the total image size, ‘σ’ represents the size of the pattern and ‘w’ represents the dimension of the tile. The time complexity of the proposed algorithm is given by equation 2.

$$Execution\ Time = (m * \log_{w}(\sigma)/wlogn) \quad (2)$$

Comparison between the performances of proposed algorithm with cuShiftOr is shown in Fig. 10. The performance of the proposed algorithm is better when the pattern size is less than 64 and the tile size matches the pattern size.

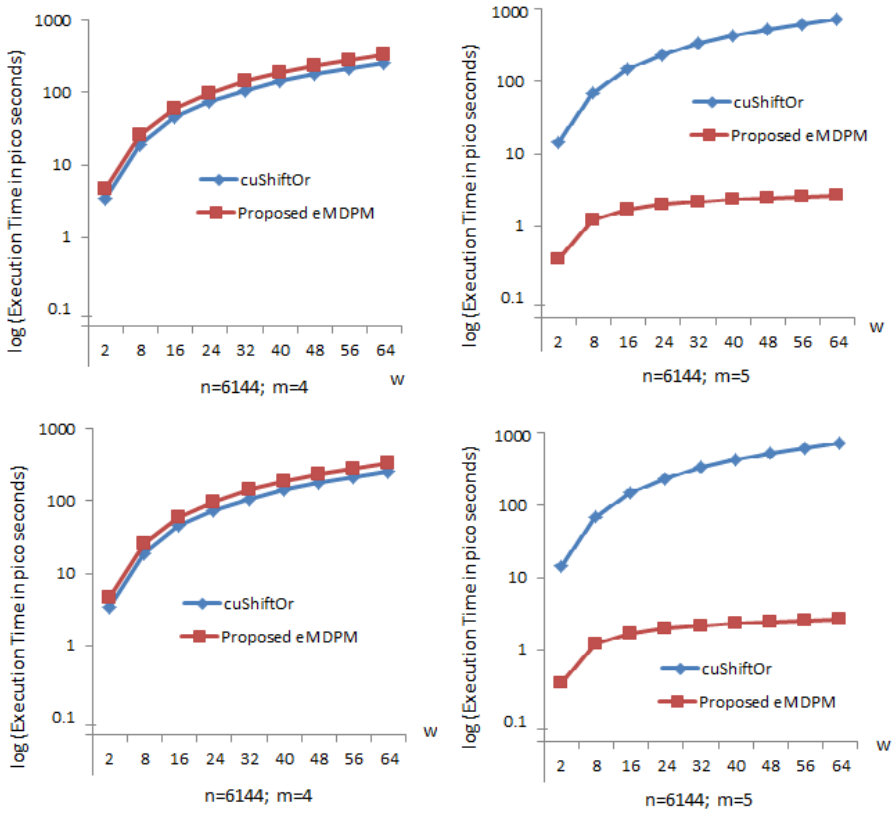

Fig. 10. Comparison of cuShiftOr for various Dimensions with Proposed eMDPM

In Fig 11, the performance of eMDPM algorithm for different values of ‘w’ is shown. For this analysis, n is considered to be 6144 and ‘m’ is set to be 3. For other values also, the algorithm can be analyzed. Here for a sample ‘m’ and ‘n’ the analysis is shown.

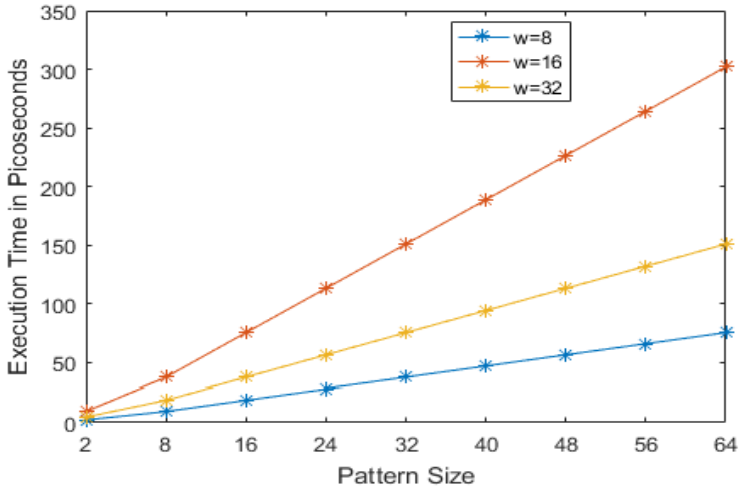

Fig.12 eMDPM : Execution Time Versus Tile Size

6 CONCLUSION AND FUTURE WORK

In this paper an efficient multi-dimensional parallel pattern matching algorithm is proposed and analyzed. The performance of the proposed algorithm with multi-dimensional images is much improved compared to existing algorithms. The implementation of the proposed algorithm is done using GPU. The proposed algorithm works well with GPU architecture by increasing CGMA ratio. Proposed algorithm makes use of tiles for data caching which improves the performance of pattern matching in GPU based architectures. The proposed algorithm is an exact bit pattern matching one. This can be extended for approximate pattern matching in the future. The role of warp size on the efficiency of the pattern matching algorithm for bigger sized patterns can also be explored in the future work.

REFERENCES

1. Mitani, Y., Ino, F. and Hagihara, K., 2017. Parallelizing exact and approximate string matching via inclusive scan on a GPU. *IEEE Transactions on Parallel and Distributed Systems*, *28*(7), pp.1989-2002
2. Agrawal, J., Diao, Y., Gyllstrom, D. and Immerman, N., 2008, June. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 147-160). ACM.
3. Cantone, D., Cristofaro, S. and Faro, S., 2010, August. A Space-Efficient Implementation of the Good-Suffix Heuristic. In *Stringology* (pp. 63-75).
4. Sahli, M. and Shibuya, T., 2012. Max-shift BM and max-shift horspool: Practical fast exact string matching algorithms. *Journal of information processing*, *20*(2), pp.419-425.

5. Park, B. and Won, C.S., 2014, June. Fast binary matching for edge histogram descriptor. In *Consumer Electronics (ISCE 2014), The 18th IEEE International Symposium on* (pp. 1-2). IEEE.
6. Hirvola, T. and Tarhio, J., 2017. Bit-Parallel Approximate Matching of Circular Strings with k Mismatches. *Journal of Experimental Algorithmics (JEA)*, *22*, pp.1-5.
7. Pfaffe, P., Tillmann, M., Lutteropp, S., Scheirle, B. and Zerr, K., 2016, August. Parallel String Matching. In *European Conference on Parallel Processing* (pp. 187-198). Springer, Cham.
8. Faro, S., 2016, June. Evaluation and improvement of fast algorithms for exact matching on genome sequences. In *International Conference on Algorithms for Computational Biology* (pp. 145-157). Springer, Cham.
9. Oladunjoye, J.A., Afolabi, A.O., Olabiyisi, S.O. and Moses, T., 2017. A Comparative Analysis of Pattern Matching Algorithm Using Bit-Parallelism Technique. *IUP Journal of Information Technology*, *13*(4), pp.20-36.
10. Fredriksson, K., 2003. Shift-or string matching with super-alphabets. *Information Processing Letters*, *87*(4), pp.201-204.
11. Kida, T., Takeda, M., Shinohara, A. and Arikawa, S., 1999, July. Shift-And approach to pattern matching in LZW compressed text. In *Annual Symposium on Combinatorial Pattern Matching* (pp. 1-13). Springer, Berlin, Heidelberg.
12. Goyal, R. and Billa, S.L., Cavium Inc, 2017. *Generating a non-deterministic finite automata (NFA) graph for regular expression patterns with advanced features*. U.S. Patent 9,563,399.
13. Hirvola, T. and Tarhio, J., 2017. Bit-Parallel Approximate Matching of Circular Strings with k Mismatches. *Journal of Experimental Algorithmics (JEA)*, *22*, pp.1-5.
14. Alfred, V., 2014. Algorithms for finding patterns in strings. *Algorithms and Complexity*, *1*, p.255.

APPENDIX

```
Microsoft Visual Studio Solution File, Format Version
12.00
# Visual Studio 15
VisualStudioVersion = 15.0.27004.2005
MinimumVisualStudioVersion = 10.0.40219.1
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") =
"string_match", "string_match\string_match.vcxproj",
"{681FDBB9-B74A-4F45-8BFD-30D5AC2A7917}"
EndProject
Global
        GlobalSection(SolutionConfigurationPlatforms) =
preSolution
                Debug|x64 = Debug|x64
                Release|x64 = Release|x64
        EndGlobalSection
        GlobalSection(ProjectConfigurationPlatforms) =
postSolution
                {681FDBB9-B74A-4F45-8BFD-
30D5AC2A7917}.Debug|x64.ActiveCfg = Debug|x64
                {681FDBB9-B74A-4F45-8BFD-
30D5AC2A7917}.Debug|x64.Build.0 = Debug|x64
                {681FDBB9-B74A-4F45-8BFD-
30D5AC2A7917}.Release|x64.ActiveCfg = Release|x64
                {681FDBB9-B74A-4F45-8BFD-
30D5AC2A7917}.Release|x64.Build.0 = Release|x64
        EndGlobalSection
        GlobalSection(SolutionProperties) = preSolution
                HideSolutionNode = FALSE
        EndGlobalSection
        GlobalSection(ExtensibilityGlobals) = postSolution
                SolutionGuid = {5CDD9C4E-78C9-460D-8301-
46707B65CD2E}
        EndGlobalSection
EndGlobal
```

CUDA Program for Proposed Algorithm

```
#include <cuda.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
using namespace std;
int*  preprocessor(int *pattern,int n,int
*orginal_image,int n1,int n2,int row_no) // take the
each row of the pattern and and return a array of the
mask
```

```
{
      int mask_0=0;
                                                      // input
-- pattern - row of the pattern orginal_image - image
matrix
      int i = n-1;
                                                      //output
-- mask - array of mask for the perticular row of pat-
tern for entire image

      while (i >= 0)
      {
            mask_0 = mask_0 + pow(2, i)*(pattern[i]);
            i--;
      }
      int mask_1 = (~mask_0) & (int (pow(2, n) - 1));
      int n_row_count = (n1 - n)+1;
      int* mask = new int[n_row_count*(n2+1)];
      int j,k;
      for (j = 0;j < n_row_count;j++)
      {
            mask[(j*(n2 + 1))] = pow(2,n)-1;
            for(k=1;k<(n2+1);k++)
            {

                  if (orginal_image[((j+row_no)*n2)+(k-
1)] == 1)
                        mask[(j*(n2 + 1))+k] = mask_1;
                  else
                        mask[(j*(n2 + 1)) +k] = mask_0;
            }
      }
      return mask;
}


__global__ void string_match_rows(int *test_a,int
*shift_value,int *size) //parallel part
{

//take the mask and return value calculated by the al-
goritham
      int n = size[1];
      int m = size[0];
      int limit_value = size[2];
```

```
      int i;
      int t_id = threadIdx.x;
      int b_id = (blockIdx.x*blockDim.x) ;

      int tem_val;
      for (i=1; ((i+t_id) < (m) &&
shift_value[t_id+b_id+i] < n); i=i*2)
      {
            tem_val = (test_a[b_id+t_id] <<
shift_value[b_id + i + t_id]);
            tem_val = tem_val & limit_value;

            test_a[b_id + i + t_id] =
test_a[b_id+i+t_id] | tem_val;

            shift_value[b_id + i + t_id] =
shift_value[b_id + i + t_id] + shift_value[b_id +
t_id];
      }
}

int main()
{

      const int image_row_no= 4;
      const int image_col_no = 7;
      const int arraySize = image_col_no+1;
      const int pattern_no_row = 3;
       const int  pattern_no_col = 3;

       int size[3] = { array-
Size,pattern_no_row,(int(pow(2, pattern_no_row) - 1))
};

       int *a;
                                                /* vari-
able for mask used bellow */
       int pattern[3] = { 1,0,1 };
                                    /* place holder for
pattern-row*/
       int c[4][arraySize];
                                          /* result in
host returned to this variable */
```

```
int b[2][arraySize] = { { 0,1,1,1,1,1,1,1 },{
0,1,1,1,1,1,1,1 } }; /* shift values for the operation
initialllized*/

      int *d_a,*d_b,*d_p_size;
                                        /* device vari-
ables allocation and copying done bellow */

      int org_pattern[pattern_no_row*pattern_no_col] =
{ 1,0,1,              /* orginal pattern inserted here
*/
                                     1,1,1,
                                     1,1,1 };
      int org_img[image_row_no*image_col_no] = {
1,1,1,0,1,1,1,              /* orginal image inserted
here */
                                   1,1,1,1,1,0,0,
                                   1,1,1,1,1,1,1,
                                   0,0,0,0,0,0,0};

      int n_row_size = (image_row_no - pattern_no_row)
+ 1;                  /* resultant array size calcula-
tion */

      cudaMalloc((void **)&d_a, (n_row_size * array-
Size * sizeof(int))); /* allocation for devicce varia-
bles*/

      cudaMalloc((void **)&d_b, (n_row_size * array-
Size * sizeof(int))); /* allocation for devicce varia-
bles*/

      cudaMalloc((void **)&d_p_size, 3 * sizeof(int));
/* allocation for devicce variables*/


      for(int z=0;z<pattern_no_row;z++){     // loop
to go through the pattern row for parallel part

            for (int q = 0;q < pattern_no_col;q++) //
loop to go through the pattern row for preprocessor
            {
                  pattern[q] = org_pattern[(z * 3 )+
q];
            }
```

```
a= preproces-
sor(pattern,pattern_no_col,org_img,image_row_no,image_
col_no,z); //preprocessor function above


      cudaMemcpy(d_a, a, (n_row_size * arraySize *
sizeof(int)),cudaMemcpyHostToDevice);        // copy-
ing values to device    -- image
      cudaMemcpy(d_b, b, (n_row_size * arraySize *
sizeof(int)), cudaMemcpyHostToDevice);       // copy-
ing values to device   -- shift value
      cudaMemcpy(d_p_size, size,  3* sizeof(int),
cudaMemcpyHostToDevice);                              //
copying values to device     -- size of arrays


      string_match_rows << <n_row_size, arraySize >> >
(d_a,d_b,d_p_size);                           // call-
ing the kernal(function on the gpu)
      cudaMemcpy(c, d_a, (n_row_size * arraySize *
sizeof(int)),cudaMemcpyDeviceToHost);        // copy-
ing back result

      cout << "result from pattern row " << z << " only
row " << z << " and " << z + 1 << " of image is re-
quired as the resultant size is calculated by (no of
image rows - no of pattern rows), here it is 2 hence
only two rows are calculated" << endl<<endl;

      for (int j=0;j<2;j++)

                  //printing the result
      {
            for (int i = 0;i < arraySize;i++)
            {

      // result are printed as the array obtained by
the operation
                  cout<< (c[j][i])<<"-";
      }
                                                      //
each pattern row is printed separately
            cout << endl;
                                                      //
it is given in the output so you can understand
```

```
      }
      cout << endl;
       }

      int l;
      cin>>l;
    return 0;
}
```