ICIC Express Letters
Volume **17**, Number **4**, April **2023**

 ISSN 1881-803X
pp. **439–448**

ACCELERATED NOVEL LOCAL TEXTURE FEATURE EXTRACTION ON A GPU

Omkar Ajay Masur[1], Ashwath Rao Badanidiyoor[1]
Naravi Gopalakrishna Kini[1] and Vidya Harekala Chandrashekara[2]

[1]Department of Computer Science & Engineering
[2]Department of Mathematics
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Karnataka State 576104, India
omkar.masur@learner.manipal.edu; { ashwath.rao; ng.kini; vidya.rao }@manipal.edu

Received June 2022; accepted September 2022

Abstract. *Texture features benefit object detection, object recognition, content-based image retrieval, and other tasks. Recently, two new local texture descriptors, Threshold Local Binary AND Pattern and Local Adjacent Neighborhood Average Difference Pattern, have been proposed. Graphical Processing Units (GPUs) are instrumental in speeding up many computationally intensive tasks. We have accelerated these texture feature extractors on a graphical processing unit by proposing parallel implementations of the algorithm in this work. Compute Unified Device Architecture (CUDA) has been used to implement the parallel GPU algorithms. We have also optimized the parallelization by leveraging memory hierarchy in a GPU. The results show that we can use GPUs to achieve a speedup of more than 20.*
Keywords: Computer vision, CUDA, Feature extraction, LANADP, TLBAP

1. **Introduction.** Computer vision aims to describe the world through images by reconstructing and simplifying properties of objects such as shape, illumination and color so that computers can easily identify patterns in them. To do so, various algorithms involving mathematical backgrounds are used. Such techniques open up new possibilities in computer science. Some interesting applications in a plethora of fields include "Plant Recognition" in agriculture [1], detection of a severe case of traumatic bleeding in patients [2], identification of oral squamous cell carcinoma [3], and also in the field of space science where one can train models in the segmentation of remotely sensed images [4]. In all these tasks, texture features play a significant role.

Texture analysis is one of the most important techniques in computer vision and image analysis. The first task in texture analysis is texture feature extraction. Image texture is a function of spatial variation in pixel intensity. The texture can identify local structures in an image and recognize tactile and optical patterns. Hence, they are helpful in image classification, content-based image retrieval, industrial inspection, and medical image analysis. There are four main texture analysis methods: statistical, structural, model-based, and transform-based [5]. Local Binary Pattern (LBP) [6], Threshold Local Binary AND Pattern (TLBAP) [7], and Local Adjacent Neighborhood Average Difference Pattern (LANADP) [7] are statistical textural image descriptors. They can describe the local spatial structure and the local contrast of the image.

LBP is a visual descriptor proposed in [6]. It considers a center pixel and neighbors (usually 8 in a 3×3 window) around it. Using them, it constructs a binary number of 1-byte length by comparing the center pixel with the values of the pixel around it. This

DOI: 10.24507/icicel.17.04.439

process is described in [6]. The bit is given 1 if higher or equal, and 0 if lower. Once this process is done for all the pixels in the window, an intermediate matrix is calculated. This intermediate matrix is convolved with a weight matrix. This weight matrix signifies how much weight each neighbor is allotted. The 1-byte number, which is thus obtained upon convolving the intermediate matrix with the weight matrix, is the feature value of those pixels.

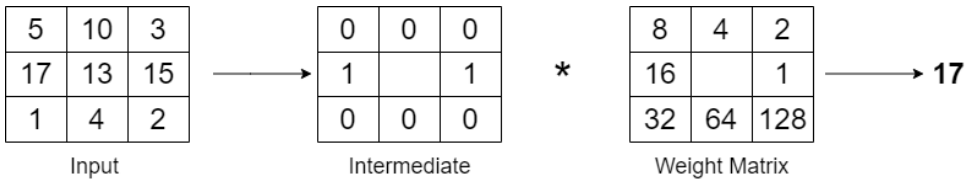

FIGURE 1. Example of LBP

1.1. **Algorithm overview.** LBP extension was proposed in [7] called Threshold Local Binary AND Pattern (TLBAP). It considers a Threshold value (th) ranging from 0 and 1 as an input to the algorithm. It identifies the highest intensity pixel in a particular window, usually of 3×3 size, and multiplies this by the threshold. Let us call this value obtained as $th2$. $th2$ is compared with all the neighbors around the center using the same technique as LBP. The resultant feature vector is called Threshold Local Binary Pattern (TLBP). TLBP is logically ANDed with LBP to get an intermediate matrix Threshold Local Binary AND Pattern. This intermediate matrix is convolved with the

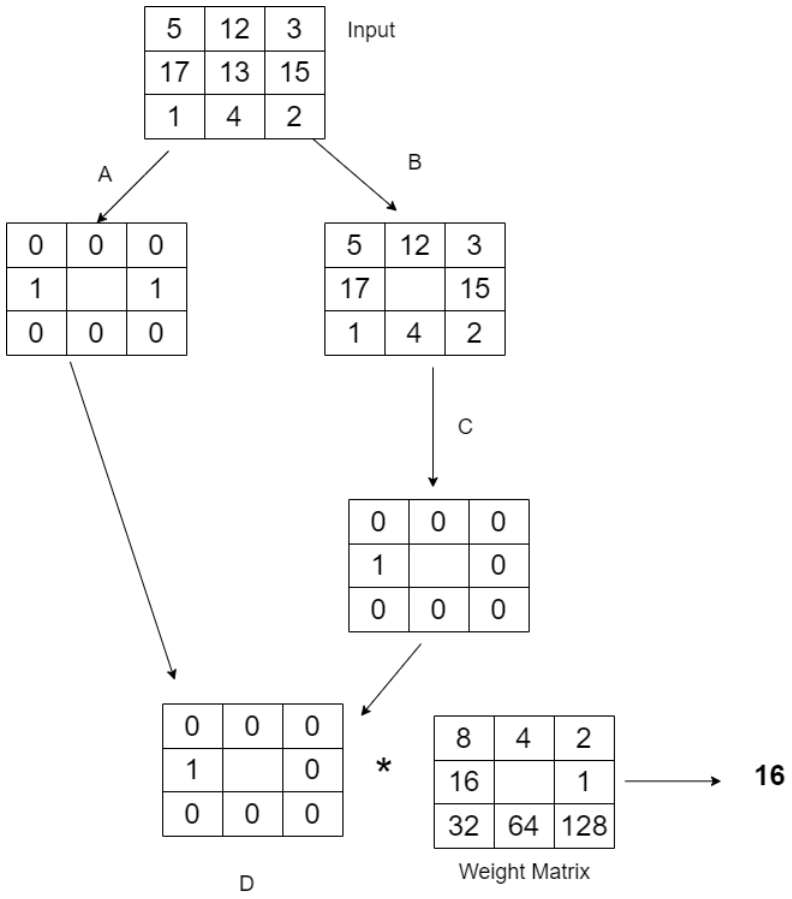

FIGURE 2. Example of TLBAP: (A) LBP on 13; (B) Highest value in neighborhood of 13 is 17 and multiply 17 by threshold 0.9 = 15.3; (C) LBP on 15.3; (D) AND Result of A, C

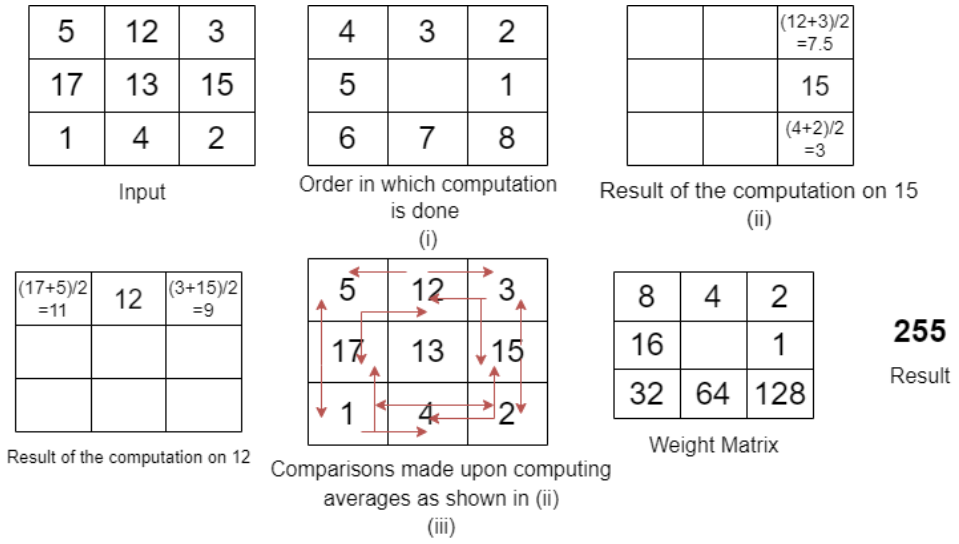

FIGURE 3. Example of LANADP

weight matrix to get a 1-byte number, which is the pixel's feature value. This process is described in Figure 2. Another extension of the LBP algorithm is the Local Adjacent Neighborhood Average Difference Pattern (LANADP). The proposed method explores the relationship of neighboring pixels with their adjacent neighbors in vertical, horizontal, and diagonal directions [7]. Yet again, upon doing all the calculations as explained in [7], the intermediate matrix is calculated, which is convolved with the weight matrix to get the feature value of the pixel. The calculations of the average are done in a cyclic order, and this order is depicted in subfigure (i) in Figure 3. Referring to (ii) from Figure 3, while computing the bit-value for pixel 1 ($value$), an average of pixels 3 and 2 ($avg1$) and pixels 7 and 8 ($avg2$) is considered. Similarly, if we are supposed to calculate the bit-value for pixel 2, the average of pixels 3 and 4, and that of pixels 1 and 8 is considered. The value of the bit is 1 if ($avg_1 >= value$ and $avg_2 >= value$) or ($avg_1 <= value$ and $avg_2 <= value$) else it is 0. The weight matrix is the same weight matrix used in TLBAP.

1.2. **Overview of CUDA.** We propose a parallel implementation of the LANADP algorithm using Compute Unified Device Architecture (CUDA). We chose CUDA and not another parallel programming platform such as OpenCL because CUDA is extraordinarily userfriendly and, at the same time, can make use of some of the most potent graphical processing units in the market by NVIDIA. At the same time, CUDA performs slightly better in terms of kernel execution times [8, 9, 10]. CUDA language works on the principle of the coexistence of a host (CPU) and one or more devices (GPUs). Each CUDA source code contains both the host and device code. Host code is sequential as it runs on the CPU. The device code is characterized by unique CUDA keywords called kernel code. The execution of any CUDA program starts from the host. The kernel code is launched from the host after allocating appropriate memory to variables in the GPU. When a kernel function is launched, it is executed by a large number of threads collectively called a *grid*. The grid of threads is organized in a two-level hierarchy. Every block has a unique Block ID, and every thread has a unique Thread ID.

The data to be processed by the threads is first transferred from the host to the device Global Memory. The threads can access their data portion using their Thread ID and Block ID. However, this Global Memory, although large in size, tends to have very high access times, long latencies, and finite access bandwidth. There are other memories available on a GPU which provide better performance. These memories can be written to or read using various APIs developed in CUDA. The first one is the Constant memory,

which, as the name suggests, is a read-only memory with short latency and high bandwidth. The second type is the shared memory, which is shared by all threads belonging to the same block. Shared memory, being an on-chip memory, is much faster to access than the global memory. However, a pertinent problem while accessing it is synchronization, i.e., ensuring no other thread overwrites the data computed by the other thread or tries to access a piece of data that has not been computed or stored by another thread. Also, shared memory is smaller in size, so care must be taken to load only the required data.

TLBAP and LANADP perform considerably well and provide high accuracy in Content-Based Medical Image Retrieval (CBMIR) [11]. They can diagnose diseases like breast cancer, lung cancer, and tuberculosis. However, as observed through the algorithms described previously, whose sequential algorithms and their respective time complexities have been depicted in Section 3, they are all computationally expensive. If used for a large number of images at once, which is generally the case in real-world applications, it would take considerable time to train a machine learning or deep-learning model. As a result, we propose a much faster parallel implementation of both TLBAP and LANADP in Section 3. We also propose an optimization of these parallel algorithms in Section 3.2. We show the results of each algorithm upon CUDA implementation in Section 4. Finally, Section 5 concludes the paper.

2. **Literature Review.** There have been works by numerous researchers on the proposals, improvement, and optimization of feature extraction and image processing algorithms. Experiments were conducted to compute the GLCM texture feature extraction method on a Cell platform [12]. It is a paper published in 2012 that uses a now-defunct computing method. It, however, establishes the advantages of using a parallel algorithm over a sequential one. [13] gives the details of parallelization of widely used feature detection algorithms SIFT (Scale-Invariant Feature Transform) and SURF (Speeded Up Robust Features) using the OpenCL and OpenGL programming languages. [14] gives an incite into distributed processing of feature descriptors. Even though our work is on parallel processing and not distributed processing per se, it is interesting to note that various feature descriptors can be parallelized as Hadoop uses a data distribution strategy and achieves considerable speedup while processing massive datasets.

Parallel implementation of the GLCM algorithm using CUDA was proposed in [15, 16]. While [15] focuses particularly on MRI images and how speedup varies with various ROIs, [16] focuses on the parallel algorithm, mainly how speedup is achieved for varying values of θ. [17] is similar to our work, where the widely used feature extraction algorithm LBP is parallelized and optimized for various image sizes. [7] introduces two novel texture feature descriptors TLBAP and LANADP. [18, 19, 20] show positive results of parallel implementations of various algorithms. [18, 20] are parallel implementations of local texture features local tiridirectional pattern and local diagonal extrema pattern on a GPU using CUDA. However, as far as TLBAP and LANADP, which are the focus of this paper, are concerned, no other work on proposing a parallel implementation is done.

3. **Methodology.** A seminal paper introduces two novel local texture descriptors: Threshold Local Binary AND Pattern and Local Adjacent Neighborhood Average Difference Pattern. The sequential algorithm for TLBAP texture feature extraction is shown in [21], developed as per the one proposed in [7]. It takes as input of an image, and a parameter threshold between 0 and 1. It has two outer *for* loops in lines 8 and 9, which iterate over all the non-border pixels of the input image. Lines 13 to 16 find the maximum in the 3×3 neighborhood. Upon finding the maximum, we multiply it with the input threshold value in line 17. Using this value, we find the TLBAP feature value in lines 18 to 22 which is stored in the output array. Since the *for* loops on lines 8 and 9 iterate over the non-border pixels, the algorithm's time complexity is $\Theta(n^2)$.

Algorithm 1, proposed by us, is the parallel version of sequential algorithm proposed in [21]. It requires the same parameters as its sequential counterpart. The algorithm determines TLBAP features and stores the determined features in the output array $outF$. The implementation of the parallel TLBAP algorithm is run with runtime thread configuration with a block size equal to 16×16, and 32×32. The grid size is equal to the ceil of quotient when divided by the block size in the two directions. Line 1 defines the weight matrix. During implementation in CUDA, it is stored in the constant memory to ensure it is accessed efficiently. In lines 2 to 9 of the algorithm, we determine the center pixel from the available center pixels the thread has to work on. Once the center pixel is determined, we obtain the maximum value among the neighbors of the center pixel. In lines 24 to 33, we obtain the TLBAP pattern. In line 34, we store the determined pattern in the output array $outF$.

Algorithm 1 Parallel threshold local binary and pattern feature extraction

Input: img, height, width, th (between 0 and 1) # $0 < th <= 1$
Weight matrix is loaded into constant memory as power2
Output: outF
1: $power[3][3] \leftarrow \{\{8,4,2\},\{16,0,1\},\{32,64,128\}\}$ # Weight matrix as shown in Figure 2
2: $thidx\_x, thidx\_y \leftarrow$ thread index in the x and y direction within the block
3: $blkidx\_x, blkidx\_y \leftarrow$ block index in the x and y direction within the grid
4: $blkdim\_x, blkdim\_y \leftarrow$ number of threads in the x and y direction within the block
5: $glob\_row\_id \leftarrow thidx\_y + blkidx\_y * blkdim\_y$
6: $glob\_col\_id \leftarrow thidx\_x + blkidx\_x * blkdim\_x$
7: **if** $glob\_row\_id < height$ && $glob\_col\_id < width$ **then**
8: **if** $glob\_row\_id != 0$ and $glob\_col\_id != 0$ and $glob\_col\_id != width - 1$ and $glob\_row\_id != height - 1$ **then**
9: max $\leftarrow$inputImage$[(glob_row_id - 1) * width + glob_col_id]$
10: # Find maximum in the local 3×3 neighborhood
11: **for** i from $glob\_row\_id - 1$ to $glob\_row\_id + 1$ **do**
12: **for** j from $glob\_col\_id - 1$ to $glob\_col\_id + 1$ **do**
13: **if** $img[i * width + j] >$ max **then**
14: max $\leftarrow img[i * width + j]$
15: **end if**
16: **end for**
17: **end for**
18: $th2 \leftarrow th *$ max
19: $centrePx \leftarrow img[glob\_row\_id * width + glob\_col\_id]$ # determine feature value using calculated threshold
20: $ans \leftarrow 0$
21: $power\_i \leftarrow 0$ # Used for iterating over weight matrix
22: **for** i from $glob\_row\_id - 1$ to $glob\_row\_id + 1$ **do**
23: $power\_j \leftarrow 0$ # Used for iterating over weight matrix
24: **for** j from $glob\_col\_id - 1$ to $glob\_col\_id + 1$ **do**
25: **if** $img[i * width + j] > centrePx$ and $img[i * width + j] > th2$ **then**
26: $ans \leftarrow ans + power2[power\_i][power\_j]$
27: **end if**
28: $power\_j += 1$
29: **end for**
30: $power\_i += 1$
31: **end for**
32: $outF[(glob\_row\_id - 1) * (width - 2) + (glob\_col\_id - 1)] \leftarrow$ ans
33: **end if**
34: **end if**

The sequential algorithm for LANADP was proposed in [21]. We give the algorithm an image as the input. The output LANADP features are stored in the variable *lanadpFeatures*. Lines 8 and 9 in the algorithm are nested *for* loops that iterate over the image's non-border elements. Lines 10 to 18 compute the LANADP feature value for pixel, which is stored in the output array on line 18. Just like TLBAP sequential algorithm, the time complexity of this algorithm is $\Theta(n^2)$ as the *for* loops iterate over all the non-bordering pixels.

The parallel version of the sequential LANADP Algorithm is proposed in Algorithm 2. Lines 1 to 12 are helper functions. *CIRCULARINDEX* takes as input an index i, and returns the circular index corresponding to the index. It is assumed that the circular

Algorithm 2 An algorithm for parallel LANADP feature extraction

```
1: function circularIndex(i)
2:     if i < 1 then
3:         return 8 + i
4:     else if i > 8 then
5:         return (i%9) + 1
6:     else
7:         return i
8:     end if
9: end function
10: function getGlobalID(row, col, width)          # width is the width of the image
11:     return row * width + col
12: end function
Input: img, height, width
Output: outF
13: power[9] ← {0, 1, 2, 4, 8, 16, 32, 64, 128}          # Flattened weight matrix
14: thidx_x, thidx_y ← thread index in the x and y direction within the block
15: blkidx_x, blkidx_y ← block index in the x and y direction within the grid
16: blkdim_x, blkdim_y ← number of threads in the x and y direction within the block
17: glob_row_id ← thidx_y + blkidx_y * blkdim_y
18: glob_col_id ← thidx_x + blkidx_x * blkdim_x
19: if glob_row_id < height && glob_col_id < width then
20:     if glob_row_id != 0 and glob_col_id != 0 and glob_col_id != width − 1 and glob_row_id != height − 1 then
21:         int mapping[9]
22:         mapping[1] ← img[getGlobalID(glob_row_id, glob_col_id + 1, width)]
23:         mapping[2] ← img[getGlobalID(glob_row_id − 1, glob_col_id + 1, width)]
24:         mapping[3] ← img[getGlobalID(glob_row_id − 1, glob_col_id, width)]
25:         mapping[4] ← img[getGlobalID(glob_row_id − 1, glob_col_id − 1, width)]
26:         mapping[5] ← img[getGlobalID(glob_row_id, glob_col_id − 1, width)]
27:         mapping[6] ← img[getGlobalID(glob_row_id + 1, glob_col_id − 1, width)]
28:         mapping[7] ← img[getGlobalID(glob_row_id + 1, glob_col_id, width)]
29:         mapping[8] ← img[getGlobalID(glob_row_id + 1, glob_col_id + 1, width)]
30:         ans ← 0
31:         for i from 1 to 8 do
32:             avg_1 ← ((mapping[circularIndex(i + 1)] + mapping[circularIndex(i + 2)]))/2
33:             avg_2 ← ((mapping[circularIndex(i − 1)] + mapping[circularIndex(i − 2)]))/2
34:             if (avg_1 >= value and avg_2 >= value) or (avg_1 <= value and avg_2 <= value) then
35:                 ans = ans + power2[i]
36:             end if
37:         end for
38:         outF[(glob_row_id − 1) * (width − 2) + (glob_col_id − 1)] ← ans
39:     end if
40: end if
```

array is of length 9. *GETGLOBALID* takes as inputs row, column, and the total width and returns the Global ID corresponding to the parameters. These helper functions are stored on the device during CUDA implementation. The algorithm is run on a runtime configuration of grid sizes of 16×16 and 32×32. Inputs to the algorithm are the image matrix and dimensions of the image in the form of height and width. The output of the algorithm is the LANADP feature. Line 13 defines the flattened form of the weight matrix. It is done so for ease of access later in the algorithm. It is stored in the constant memory during implementation in CUDA. In lines 14 to 21, we determine the center pixel from the available center pixels the thread has to work on. Once the center pixel is determined, we obtain the neighbors of the center pixel and store it in 1D array *mapping*. In lines 34 to 40, we obtain the LANADP pattern and store in the output array *outF* in line 40.

Without the use of shared memory, there is no need for synchronization. Individual threads perform the computation directly and store the result in the output buffer. The computation performed is precisely the same in both shared memory and without the shared memory method. The only difference is in the type of memory from which the threads read the data during computation.

3.1. **Input images.** The amount of computation involved in extracting TLBAP and LANADP is directly proportional to the number of non-border pixels in the input image. The number of non-border pixels in an input image of size $H \times W$ is $(H-2) \times (W-2)$.

FIGURE 4. Input image of size 256×256

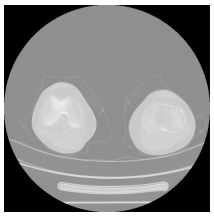

FIGURE 5. Input image of size 512×512

FIGURE 6. Input image of size 1024×1024

Hence, we have considered input images of varying sizes in our experiments. We have considered input medical images of sizes 256×256, 512×512, and 1024×1024. The input images are shown in Figures 4, 5 and 6.

3.2. **Optimization using shared memory.** CGMA is a good metric for measuring the efficiency of the kernel function. A high CGMA ratio is desirable. In the above algorithms, to increase CGMA, the number of operations per access to global memory needs to be increased or the number of accesses to the global memory needs to be reduced. The number of operations involved in both the TLBAP algorithm and LANADP algorithm is fixed; hence, only decreasing accesses to global memory is needed. Instead of kernels accessing data from global memory an alternative memory named shared memory can be used to store input data. Each thread can first copy the necessary image data from global memory to shared memory. Once all threads carry out this, they can access image data from shared memory. This will reduce memory access time and will achieve better CGMA and speedup. Algorithm 3 is used by each thread to load the image into the shared memory.

Once the appropriate part of the image is loaded by all threads in the block, the individual threads perform the appropriate computation using Algorithm 1 or Algorithm 2, with the difference being instead of accessing the data through the global memory, the data is accessed using the shared memory, into which data was loaded using Algorithm 3. The result is stored in the output buffer. The time complexity of either of the algorithms does not change as the data is loaded into the shared memory in $\Theta(1)$ time.

We have implemented both TLBAP and LANADP algorithms using CUDA C version 10.2. We have taken readings on a GPU on a server with a 2-Dual Core Intel Xeon processor powered with 48GB Memory. The GPU is NVIDIA GeForce GTX with 1392 MHz Core, 768 CUDA Cores, Pascal Architecture and 7Gb/s Memory speed. Also, we

Algorithm 3 Algorithm for storing data in shared memory

function GETGLOBALID($row, col, width$) # width is the width of the image
    return $row * width + col$
end function
Input: img, height, width
Output: sharedDat
$thidx\_x, thidx\_y \leftarrow$ thread index in the x and y direction within the block
$blkidx\_x, blkidx\_y \leftarrow$ block index in the x and y direction within the grid
$blkdim\_x, blkdim\_y \leftarrow$ number of threads in the x and y direction within the block
$glob\_row\_id \leftarrow thidx\_y + blkidx\_y * blkdim\_y$
$glob\_col\_id \leftarrow thidx\_x + blkidx\_x * blkdim\_x$
$sharedDat[NUMBER\_OF\_THREADS + 2][NUMBER\_OF\_THREADS + 2]$
$glob\_row\_id \leftarrow glob\_row\_id + 1$
$glob\_col\_id \leftarrow glob\_col\_id + 1$
$local\_row \leftarrow thidx\_y$
$local\_col \leftarrow thidx\_x$
$shared\_row \leftarrow thidx\_y + 1$
$shared\_col \leftarrow thidx\_x + 1$ # Variables to access data within shared memory
$sharedDat[shared\_row][shared\_col] \leftarrow img[getGlobalID(glob\_row\_id, glob\_col\_id, cols)]$
if $local\_row == 0$ **then**
    $sharedDat[shared\_row - 1][shared\_col] \leftarrow img[getGlobalID(glob\_row\_id - 1, glob\_col\_id, cols)]$
    if $locals\_col == 0$ **then**
        $sharedDat[shared\_row - 1][shared\_col - 1] \leftarrow img[getGlobalID(glob\_row\_id - 1, glob\_col\_id - 1, cols)]$
    end if
    if $local\_col == blkdim\_x - 1$ **then**
        $sharedDat[shared\_row - 1][shared\_col + 1] \leftarrow img[getGlobalID(glob\_row\_id - 1, glob\_col\_id + 1, cols)]$
    end if
end if
if $local\_row == blkdim\_y - 1$ **then**
    $sharedDat[shared\_row + 1][shared\_col] \leftarrow img[getGlobalID(glob\_row\_id + 1, glob\_col\_id, cols)]$
    if $local\_col == 0$ **then**
        $sharedDat[shared\_row + 1][shared\_col - 1] \leftarrow img[getGlobalID(glob\_row\_id + 1, glob\_col\_id - 1, cols)]$
    end if
    if $local\_col == blkdim\_x - 1$ **then**
        $sharedDat[shared\_row + 1][shared\_col + 1] \leftarrow img[getGlobalID(glob\_row\_id + 1, glob\_col\_id + 1, cols)]$
    end if
end if
if $local\_col == 0$ **then**
    $sharedDat[shared\_row][shared\_col - 1] \leftarrow img[getGlobalID(glob\_row\_id, glob\_col\_id - 1, cols)]$
end if
if $local\_col = blkdim\_x - 1$ **then**
    $sharedDat[shared\_row][shared\_col + 1] \leftarrow img[getGlobalID(glob\_row\_id, glob\_col\_id + 1, cols)]$
end if

have implemented optimization of the algorithms using shared memory. The results are presented in Section 4.

4. **Results.** Tables 1 and 2 show the readings for experiments conducted on TLBAP sequential and parallel algorithms proposed. As observed from them, the sequential execution time increases with the increase in the size of the image. A considerable speedup is achieved on the proposed parallel algorithms compared with the sequential algorithms for the exact image size. We have also tested the shared memory implementation of the algorithm on the exact image sizes and the same grid sizes of 16×16 and 32×32. Shared

TABLE 1. TLBAP results

Image size	Computation time (ms)				
	Sequential	Parallel			
		Without shared memory		With shared memory	
		16×16	32×32	16×16	32×32
256×256	10.110870	4.250779	4.404883	1.937791	1.978193
512×512	38.947736	16.874831	17.565300	7.375525	7.583423
1024×1024	142.142200	62.278262	60.771245	28.955355	30.359479

Table 2. LANADP results

Image size	Computation time (ms)				
	Sequential	Parallel			
		Without shared memory		With shared memory	
		16×16	32×32	16×16	32×32
256×256	18.052515	1.162166	1.217686	0.678101	0.788468
512×512	77.558890	4.259022	4.596100	2.562525	3.100219
1024×1024	277.199182	16.304927	17.311025	9.658553	12.147245

Table 3. Speedup of TLBAP and LANADP

Image size	Speed up (*sequential time/parallel time*)			
	TLBAP		LANADP	
	Non-shared memory	Shared memory	Non-shared memory	Shared memory
256×256	2.3785922	5.217729	15.533508	26.6221624
512×512	2.3080371	5.2806730	18.2104929	30.2665886
1024×1024	2.28237262	4.909012	17.000945	28.6998665

memory is faster to access, and hence it significantly boosts performance compared to the non-shared memory implementation.

Table 3 shows the speedup, i.e., $\frac{sequential\ time}{parallel\ time}$ for both the non-shared memory implementation and the shared memory implementation on the 16×16 grid size. Comparing only one is sufficient since, as observed, both the 16×16 and 32×32 grid sizes give almost the same time.

The time complexity of the parallel algorithms proposed is $\Theta(1)$ per active thread. This is significantly better than the original time complexity. This is also evident from the timings obtained, as shown in Section 4. We achieve a speedup of over 2 for TLBAP and over 18 for LANADP. In order to increase the CGMA ratio, we have also optimized these parallel algorithms using shared memory. The time complexity of the optimized algorithm does not change from the parallel version. The results justify the usage of shared memory, proving that the algorithm indeed runs faster. Speedup of over 5 is obtained for the TLBAP using the optimized version, and over 28 for LANADP optimized version. The observed speedup proves that our proposed novel approach to computing TLBAP and LANADP features is more efficient than the naive sequential implementation. Hence, it could easily be used to train classification models like those proposed in [11] more efficiently.

5. **Discussion and Conclusion.** Graphical processing units have become almost necessary in today's computationally intensive applications such as machine learning, and image processing. It is essential for developers and researchers working in such fields to adapt to this change and develop algorithms suitable for processing on a GPU. As a result, we propose parallel implementations of two texture descriptors, namely LANADP and TLBAP. Using them and leveraging memory optimizations, we achieve speedups of more than 20, compared to sequential counterparts. It is proven that based on comparisons with CPU, the calculation of TLBAP and LANADP is feasible and much more efficient. We can also develop parallel algorithms for multiple other local texture feature descriptor algorithms namely Local Gradient Hexa Pattern, Local Directional Gradient Pattern, Local Quadruple Pattern, Centre Symmetric Quadruple Pattern using similar approaches. This would make real world applications like machine learning and deep learning faster and more efficient.

REFERENCES

[1] C. Yang, Plant leaf recognition by integrating shape and texture features, *Pattern Recognition*, vol.112, 107809, https://www.sciencedirect.com/science/article/pii/S0031320320306129, 2021.

[2] L. Yang et al., Traumatic bleeding detection based on fusion of 3D shape and local texture features, *J. Clin. Med. Img.*, vol.5, no.15, pp.1-12, 2021.

[3] Z. Yang, J. Shang, C. Liu, J. Zhang and Y. Liang, Identification of oral squamous cell carcinoma in optical coherence tomography images based on texture features, *Journal of Innovative Optical Health Sciences*, vol.14, no.1, 2140001, 2021.

[4] S. G. A. Usha and S. Vasuki, Significance of texture features in the segmentation of remotely sensed images, *Optik*, vol.249, 168241, https://www.sciencedirect.com/science/article/pii/S0030402621017666, 2022.

[5] L. Armi and S. F. Ershad, Texture image analysis and texture classification methods – A review, *CoRR*, http://arxiv.org/abs/1904.06554, 2019.

[6] D.-C. He and L. Wang, Texture features based on texture spectrum, *Pattern Recognition*, vol.24, no.5, pp.391-399, https://www.sciencedirect.com/science/article/pii/0031320391900527, 1991.

[7] R. Biswas, S. Roy and D. Purkayastha, An efficient content-based medical image indexing and retrieval using local texture feature descriptors, *International Journal of Multimedia Information Retrieval*, vol.8, no.12, 2019.

[8] A. Asaduzzaman, A. Trent, S. Osborne, C. Aldershof and F. N. Sibai, Impact of CUDA and OpenCL on parallel and distributed computing, *2021 8th International Conference on Electrical and Electronics Engineering (ICEEE)*, pp.238-242, 2021.

[9] T. Imankulov, B. Daribayev and S. Mukhambetzhanov, Comparative analysis of parallel algorithms for solving oil recovery problem using CUDA and OpenCL, *International Journal of Nonlinear Analysis and Applications*, vol.12, no.1, pp.351-364, 2021.

[10] J. Fang, A. L. Varbanescu and H. J. Sips, A comprehensive performance comparison of CUDA and OpenCL, *2011 International Conference on Parallel Processing*, pp.216-225, 2011.

[11] M. Rashad, S. Nooh, I. Afifi and M. Abdelfatah, Effective of modern techniques on content-based medical image retrieval: A survey, *International Journal of Computer Science and Mobile Computing*, vol.11, 2022.

[12] A. Shahbahrami, T. Pham and K. Bertels, Parallel implementation of gray level co-occurrence matrices and haralick texture features on cell architecture, *The Journal of Supercomputing*, vol.59, pp.1455-1477, 2012.

[13] S. H. Kang, S.-J. Lee and I. K. Park, Parallelization and optimization of feature detection algorithms on embedded GPU, *International Workshop on Advanced Image Technology*, vol.108, pp.164-167, 2014.

[14] A. K. Sabarad, M. H. Kankudti, S. Meena and M. Husain, Color and texture feature extraction using Apache Hadoop framework, *2015 International Conference on Computing Communication Control and Automation*, pp.585-588, 2015.

[15] H.-Y. Tsai, H. Zhang, C.-L. Hung and G. Min, GPU-accelerated features extraction from magnetic resonance images, *IEEE Access*, vol.5, pp.22634-22646, 2017.

[16] H. Hong, L. Zheng and S. Pan, Computation of gray level co-occurrence matrix based on CUDA and optimization for medical computer vision application, *IEEE Access*, vol.6, pp.67762-67770, 2018.

[17] A. R. Badanidiyoor and G. K. Naravi, $\theta(1)$ time complexity parallel local binary pattern feature extractor on a graphical processing unit, *ICIC Express Letters*, vol.13, no.9, pp.867-874, 2019.

[18] B. A. Rao, G. N. Kini, P. K. Aithal, K. Vaishnavi and U. N. Kamath, Parallel local tridirectional feature extraction using GPU, in *Proceedings of the International Conference on Paradigms of Communication, Computing and Data Sciences*, M. Dua, A. K. Jain, A. Yadav, N. Kumar and P. Siarry (eds.), Singapore, Springer Singapore, 2022.

[19] C. R. Karthik, A. G. Shanbhag, B. A. Rao, P. K. Aithal and G. N. Kini, Parallelization of cocktail sort with MPI and CUDA, in *Proceedings of the International Conference on Paradigms of Communication, Computing and Data Sciences*, M. Dua, A. K. Jain, A. Yadav, N. Kumar and P. Siarry (eds.), Singapore, Springer Singapore, 2022.

[20] B. A. Rao and N. G. Kini, Parallelization of local diagonal extrema pattern using a graphical processing unit and its optimization, in *Recent Trends in Mathematical Modeling and High Performance Computing*, V. K. Singh, Y. D. Sergeyev and A. Fischer (eds.), Cham, Springer International Publishing, 2021.

[21] B. Rao and N. Kini, Algorithms for extracting various local texture features, *Journal of Physics: Conference Series*, vol.2161, no.1, 2022.