Parallel Blob Extraction Using the Multi-core Cell Processor

Praveen Kumar[1], Kannappan Palaniappan[1,⋆], Ankush Mittal[2], and Guna Seetharaman[3]

[1] Dept. of Computer Science, Univ. of Missouri, Columbia, MO 65211, USA
[2] Dept. of Elec. and Comp. Eng., Indian Inst. of Tech. Roorkee, 247667, India
[3] Air Force Research Laboratory, Information Directorate, Rome NY 13441, USA

Abstract. The rapid increase in pixel density and frame rates of modern imaging sensors is accelerating the demand for fine-grained and embedded parallelization strategies to achieve real-time implementations for video analysis. The IBM Cell Broadband Engine (BE) processor has an appealing multi-core chip architecture with multiple programming models suitable for accelerating multimedia and vector processing applications. This paper describes two parallel algorithms for blob extraction in video sequences: binary morphological operations and connected components labeling (CCL), both optimized for the Cell-BE processor. Novel parallelization and explicit instruction level optimization techniques are described for fully exploiting the computational capacity of the Synergistic Processing Elements (SPEs) on the Cell processor. Experimental results show significant speedups ranging from a factor of nearly 300 for binary morphology to a factor of 8 for CCL in comparison to equivalent sequential implementations applied to High Definition (HD) video.

1 Introduction

Real-time applications of image and video processing algorithms have seen explosive growth in number and complexity over the past decade driven by demand from a variety of consumer, scientific and defense applications, combined with the wide availability of inexpensive digital video cameras and networked computing devices. Video object detection forms a core stage of visual computing in a number of applications like video surveillance [1], visual biometrics, activity analysis in video [8], smart rooms for video conferencing, visual effects for film, content-based spatial queries [13], tracking of geospatial structures in satellite imagery [17], and segmentation of cells in biomedical imagery [2], all of which have high computational loads, storage and bandwidth requirements. A critically challenging goal for many of these applications is (near) real-time processing frame rates of 20 to 30 fps (frames per second) and high definition (HD) or better spatial image resolution.

⋆ This work was partially supported by U.S. Army Research Lab/Leonard Wood Institute award LWI-181223 and U.S. National Institute of Health award R33 EB00573.

J. Blanc-Talon et al. (Eds.): ACIVS 2009, LNCS 5807, pp. 320–332, 2009.

New architectures and parallelization strategies for video analysis are being developed due to the increased accessibility of multi-core, multi-threaded processors along with general purpose graphics processing units. For example, IBM's Cell Broadband Engine (BE) is based on an architecture made of eight SPEs delivering an effective peak performance of more than 200 GFlops, using very wide data-paths and memory interchange mechanisms. A good exposition of scientific computing and programming on the Cell BE is provided in[4]. Details of implementing scientific computing kernels and programming the memory hierarchy can be found in [15] and [6], respectively. The potential benefits of multi-core processors can only be harnessed efficiently by developing parallel implementations optimized for execution on individual processing elements requiring explicit handling of data transfers and memory management. The process of refactoring legacy code and algorithms – originally optimized for sequential architectures – to modern multicore architectures invariably requires insight and reanalysis which opens the door for creative innovations in algorithm design, data structures and application specific strategies as demonstrated in this paper.

Research efforts from both academia and industry have shown the strength of the Cell BE for video processing and retrieval [16,9], compression [7,11] and other computer vision applications [5]. However there is not much work reported on parallelizing algorithms/operations for video object detection and extraction on the Cell processor. We have implemented a variety of image and video analysis algorithms including motion estimation, blob segmentation and feature extraction in the context of object detection and tracking using multi-core systems at varied levels of scene complexity and workload. In this paper, we describe our parallel implementation of the morphological processing and connected components labeling (CCL) algorithms for blob extraction optimized for the Cell processor. The rest of the paper is organized as follows: Section 2 gives a brief overview of the moving blob segmentation algorithm and discuss the computation and memory characteristics. Section 3 and 4 gives the detailed description of the proposed parallel implementation for morphological operations and CCL algorithms respectively. Section 5 describes the performance evaluation of our implementation and the speedup achieved followed by the conclusions.

2 Moving Blob Extraction Algorithm

The multistage algorithm for extracting blobs or objects that are moving with respect to their background is briefly outlined below:

1. Motion estimation and detection of foreground/moving regions. This step forms the bulk of computation which varies depending on the complexity and robustness of the motion-estimation algorithm used. We use flux tensor computation [1,3] which consists of (separable) 3-D convolutions for calculating spatio-temporal derivatives, followed by 3-D weighted integration to compute the flux tensor trace. Then, a threshold opertation is applied to create a binary image or mask corresponding to moving regions. In our previous work [12], we parallelized this step on Cell BE and our implementation was benchmarked to process 58 frames/second on HD frame size (1920×1080).

2. Consolidation, filtering and elimination using binary morphology. Morphological operations "opening" and "closing" are applied to clean-up spurious responses to detach touching objects and fill in holes for single objects. Opening is erosion followed by dilation and closing is dilation followed by erosion. More details on basic erosion and dilation operator is explained in section 3 in the context of parallel implementation. Opening is applied to remove small spurious flux responses, and closing would merge broken responses. There is high degree of parallelism in this step, and it is computationally expensive step as these operators have to be applied in several passes on the whole image. Therefore, faster parallel implementation is not only intuitive but indispensable for real-time processing.
3. Detection of connected components and postprocessing. In principle, the binary image resulting from Step 2 must have one connected region for each separately moving object. These regions or blobs must be uniquely labeled, in order to uniquely characterize the object pixels underlying each blob. Since there is spatial dependency at every pixel, it is not straightforward to parallelize it. Although the underlying algorithm is simple in structure, the computational load increases with image size and the number of objects − the equivalence arrays become very large and hence the processing time [10]. Furthermore, with all other steps being processed in parallel with high throughput, it becomes imperative to parallelize this step and to avoid it from becoming a bottleneck in the processing stream.
4. Compute image statistics for each blob/object including: bounding box, centroid, area, perimeter etc.

3 Parallel Implementation of Binary Morphological Dilation and Erosion

Morphological operations process an input image by applying a structuring element and producing an output image, where each computed pixel is based on a comparison of the corresponding pixels drawn from the structuring element and the input image. The most basic morphological operations are dilation and erosion; and, both opening and closing are compound operations based on successive application of dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels along object boundaries. The rule for dilation is that the value of the output pixel is the maximum value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to the value 1, the output pixel is set to 1. The rule for erosion is that the value of the output pixel is the minimum value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to 0, the output pixel is set to 0. This is described mathematically as:

$$A \oplus B = \{z|(\hat{B})_z \cap A \neq \emptyset\} \tag{1}$$

$$A \ominus B = \{z|(B)_z \subseteq A\} \tag{2}$$

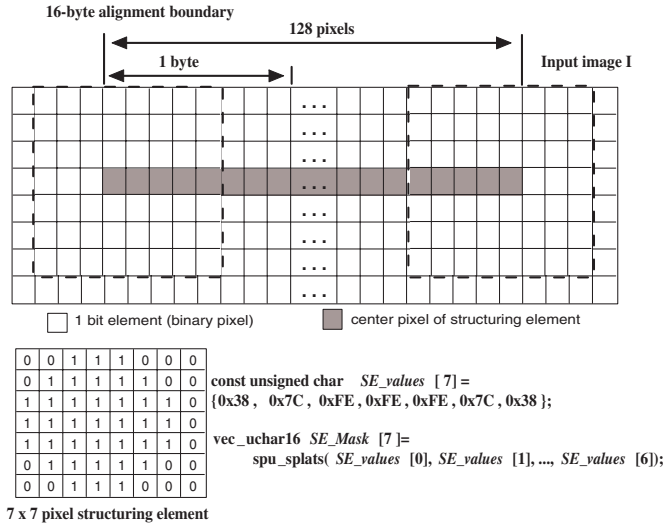

Fig. 1. Structuring element representation and processing on input image

where $\hat{B}$ is the reflection of set B and $(B)_z$ is the translation of set B by point z as per the set theoretic definition.

In a recent paper [14], the authors describe their parallel implementation of morphology using the Cell processor for the OpenCV environment. However, this implementation uses one byte to represent each pixel and thus can process only 16 pixels simultaneously by utilizing the 128-bit SIMD SPE computing unit. Our implementation, for blob processing using a binary input image, represents each pixel by a single bit (packed as bytes) and utilizes bitwise AND/OR SIMD comparison operations to process 128 pixels simultaneously. However, this requires an appropriate bit manipulation strategy for pixel based data access (i.e., bit by bit). The Cell BE processor is optimized for loading data from the local store that is aligned into 128-bit or 16 byte vector cache lines. Thus, it became a challenge to move the required data from local store to the SPU registers.

The data is packed in aligned byte vectors and we execute specialized instructions from SIMD intrinsic library to execute shuffle (*spu_shuffle*) or bit rotate operations (*spu_rlqw*) to access pixel data that is located on arbitrary alignment or the boundary of the 16 byte alignment. The *spu_shuffle* combines the bytes of two vectors as per an organization pattern defined in third vector. The required alignment pattern are stored in static look-up tables in the SPU local store. Figure 1 shows the processing of input image by structuring element and its construction with an example of 7×7 pixel elliptical shaped element. Each row of structuring element is represented using one byte element with appropriate bit pattern stored in a array $values$ and which is replicated to 16 byte vector array $Mask$ to operate on 128 pixel data of I at once. Figure 2 a) shows how to access a 16 byte aligned vector data (*current*) and its neighbor vector data with one byte shift in left (*previous*)

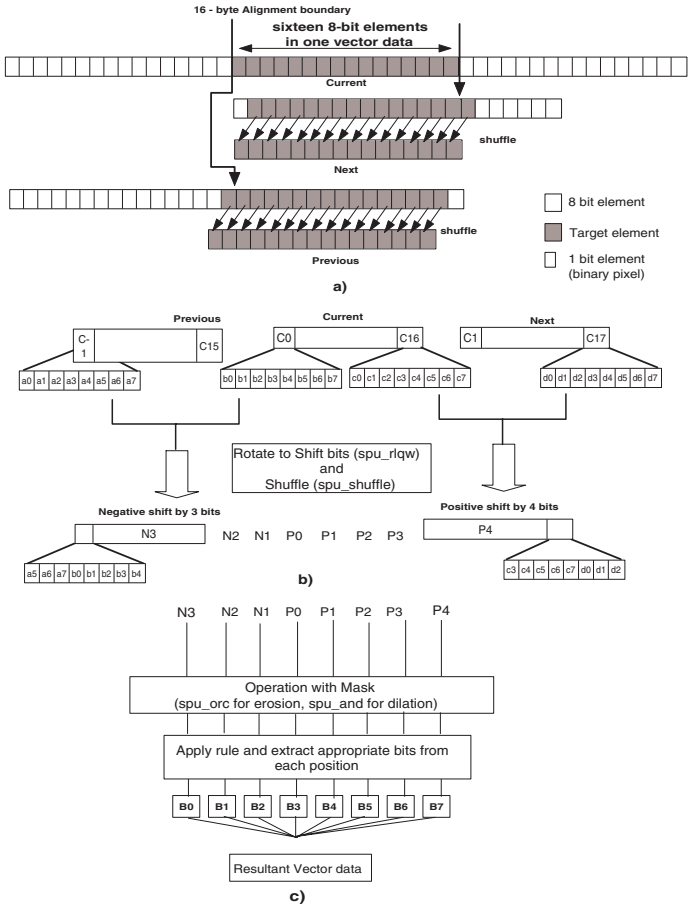

Fig. 2. Data alignment and access method

and right (*next*) direction. Figure 2 b) shows how to get further bit by bit access using bit rotate and shuffle operations and the subsequent comparison operations.

4 Parallel Implementation of CCL

Connected Components Labeling (CCL) scans an image and groups its pixels into components based on pixel connectivity. In the first step, the image is scanned pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions, i.e. regions of adjacent pixels which share the same set of intensity values and temporary labels are assigned. CCL works on binary or graylevel images and different measures of connectivity (4-connectivity, 8-connectivity etc.) are possible. For our blob segmentation, the input is binary image and 8-connectivity measure is considered. After completing the scan, the

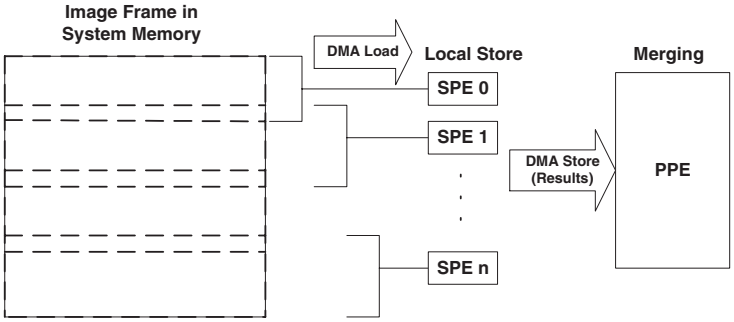

Fig. 3. Parallelization model for connected components labeling on the IBM Cell BE

equivalent label pairs are sorted into equivalence classes and a unique label is assigned to each class. In the final step, a second scan is made through the image, during which each label is replaced with its associated equivalence class label.

The proposed CCL parallelization approach for the Cell architecture belongs to the class of divide-and-conquer algorithms. The PPE runs the main process which divides the image into multiple regions and allocates the labeling task to multiple threads running on SPE's then merges the results from each SPE to generate the final label across the entire image. Each SPE performs DMA data loading from the system memory and labels the allocated region independently. The data can be partitioned into blocks of rows, columns or tiles. Dividing the image data into tiles would increase the number of cases required for the algorithm to handle during merging. Division of the image into blocks of columns allows only one row per DMA transfer requiring a list of DMA request for fetching the entire data. Therefore, dividing the image into blocks of rows is preferred because a bulk of data (several rows) can be transferred per DMA request issue. Some components may span multiple regions, so to ensure that such components get a correct equivalence label in the merge step, each region allocated to one SPE has just one overlapping row with top and bottom adjacent regions. Figure 3 shows the main phases of our parallelization approach.

The resultant array of local labels within each region is put back into main memory through DMA store operation. The PPE uses a list of pointers $Region[i]$ to point to arrays that maintain the local labels with respect to SPE_i. Initially, index for each array element is the local label before equivalence resolution whereas the array element itself is the local label after equivalence resolution in the corresponding regions. Since the labeling starts from value 1, the array location for index 0 is used to store the total number of distinct labels after resolution of equivalence class of labels. To connect each region with its neighboring regions to generate the actual label within the entire image, the PPE updates the array elements in $Region[i]$ by adding the total labels T that reached at the end of $Region[i-1]$. Figure 4 depicts the an example of list of labels for $Region[i]$ which shows that local label 1, 2 and 5 are equivalent with local label 1 and their global label within the entire image is $T+1$; local label 3, 4, and 6 are equivalent with local label 2 and their global label is $T+2$ where T is the total labels reached

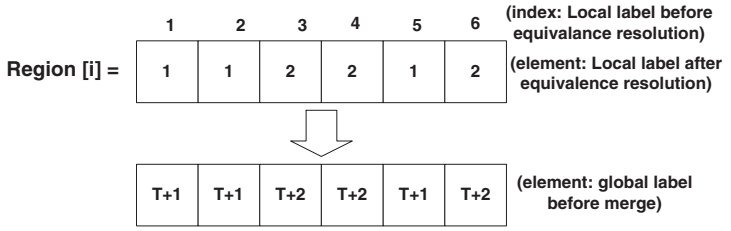

Fig. 4. An example of list of labels in Region[i]

at the end of $Region[i-1]$. While updating the labels in the global list we also need to resolve equivalences of pixel labels for the overlapping row between regions. To do this we use a list of pointers, $OverlapBottom[i]$ and $OverlapTop[i]$ to store the local labels for the overlapping pixel in the bottom and the top row respectively, of the region processed by SPE_i. Now for any pixel k in the overlapping row between regions i and $i-1$, the local labels as calculated by SPE_i and SPE_{i-1} are stored in $OverlapTop[i][k]$ and $OverlapBottom[i-1][k]$ which should be equivalent. We use this information to resolve equivalence of labels across different regions during merge phase of algorithm.

PPE Implementation. The part of the proposed algorithm implemented on the PPE side is presented in Algorithm 1. In the beginning, four buffers are created: $Frame, Region, OverlapTop$ and $OverlapBottom$. These buffers are used to store the frame pixels to be sent to the SPEs and the results obtained from each SPE. The image is divided into N (=number of available SPE's) regions with an overlap of one row at the top and bottom of the region with the adjacent region. The first region and the last region does not have any overlap at the top and bottom respectively. Correspondingly, the address locations in the buffers is determined and send as control block to the SPE thread. The SPE performs DMA operations to load the input data and store the results at these locations. PPE waits for the SPE to finish computing and notify the total number of local labels through mailbox communication. Then it updates the labels by going through each $Region[i]$ one by one, adding the total labels reached till previous region and resolving the equivalence by calling merge function.

SPE Implementation. On the SPE side, the first stage of the algorithm is essentially like any standard sequential algorithm which scans the allocated region of the image pixel by pixel, assigning a temporary label to a new pixel and marks the labels of connected pixels as equivalent. However, the algorithm used to resolve the equivalence class is implemented in a way that utilizes the SIMD instructions. The standard sequential algorithm which efficiently resolves the equivalence class of labels using *Union-Find* algorithm by trying to minimize the height of the search tree is not ideal for an SPE implementation which delivers most of its computational capacity by issuing vectorized SIMD instructions.

Algorithm 1. Parallel Connected Components Labeling: PPE side

1: Allocate $Frame, Region, OverlapTop, OverlapBottom$ and control block cb buffer
2: Divide image into N regions along with the overlap
3: **for** each SPE thread i **do**
4: Create thread i to run on SPE i
5: Load the control blocks $cb[i]$ with the corresponding address location for $Frame, Region, OverlapTop, OverlapBottom$ buffer to the thread
6: **end for**
7: **for** each SPE thread i **do**
8: Get Total number of local labels from SPE i mailbox into $Labels[i]$
9: Wait for thread joining and destroy the context
10: **end for**
11: Initialize total number of labels $T \leftarrow 0$
12: **for** each region i **do**
13: **for** each label index $j = 1$ to $Labels[i]$ **do**
14: $Region[i][j] \leftarrow T + Region[i][j]$
15: **end for**
16: $T \leftarrow T + Region[i][0]$
17: **if** region is not first **then**
18: Call $Merge(i, T, Region, OverlapTop, OverlapBottom)$ function to update global labels and value of T
19: **end if**
20: **end for**

The choice of an algorithm for implementing an application on SPE depends more on how the operations can be grouped for issuing in SIMD fashion to utilize the 128-bit SIMD units which can operate on 16 8-bit integers, eight 16-bit integers, four 32-bit integers, or four single precision floating-point numbers in a single clock cycle. Hence, we choose an alternative algorithm that resolves equivalences by expressing equivalent relations as a binary matrix and apply the Floyd-Warshall algorithm to obtain transitive closure as shown in Algorithm 2. An interesting point to note in this algorithm is that although it takes $O(n^3)$ OR operations, these can be implemented very efficiently on SPE's using SIMD bitwise OR operations reducing approximately 16 separate OR operations to one packed OR operation. The corresponding vectorized code on SPE side follows:

Listing 1.1. SIMDized SPE Code for Equivalence-Class Resolution

```
for (j=1;j<n;j++){
  for (i=1;i<n;i++){
     if (T[i*n+j]==1){
    vec_uchar16* v1=(vec_uchar16*) &T[i*n];
    vect_uchar16* v2=(vect_uchar16 *) &T[j*n];
     for (k=1;k<size/16;k++){
        v1[k]=spu_or(v1[k],v2[k]);}
} } }
```

Algorithm 2. Floyd-Warshall Algorithm for Equivalence Resolution of Classes

```
Start with T ← A
n ← no.oflabels
for each j from 1 to n do
  for each i from 1 to n do
    if T(i, j) = 1 then
      for each k from 1 to n do
        T(i, k) = T(i, k)ORT(j, k)
      end for
    end if
  end for
end for
```

Due to memory constraints on the local store, only a portion of the image data can be brought into the local store at a time. This required the implementation to handle the spatial dependancy on the previous row pixel, whenever a new block of rows is fetched. This was done by using two arrays *OverlapTop* and *OverlapBottom* to store the labels of the pixels in the top and bottom row for the current block of rows. The bottom row from the previous block of rows needs to be combined with the first row from the next block to check pixel connectivity; likewise the *OverlapBottom* array for the current block of rows is updated to propagate information to the next block of rows. Finally when the scan is complete, *OverlapTop* and *OverlapBottom* arrays are sent to the PPE for use in merging labels in the adjacent block regions. The SPE also sends out the array of labels after resolving label equivalences in the buffer $Region[i]$.

5 Experimental Results

We evaluated and measured the execution time of our implementation of morphological processing and connected components labeling on the SONY PS-3 which has only six active Cell SPE processors out of eight. In our experiments with morphological processing, we compared our performance with that reported in Cell OpenCV.The experiment was conducted by using all the 6 SPE's and varying the size and shape of the structuring element. Table 1 shows the measured execution time of one dilation or erosion operation averaged over 10 executions when the input image size is 1024×768. The original code implemented only on the PPE represents the sequential performance baseline and the optimized code implemented on the SPEs represents the parallel performance achieved on the multicore Cell.

An interesting point to note is that for a given size of structuring element, regardless of the structuring element shape our implementation has roughly constant cost, whereas the sequential and OpenCV implementations become significantly slower as reflected in Table 1. This is because our representation of the structuring element uses the same number of bits for a given window size irrespective of shape which has a dramatic impact on the morphology computation

Table 1. Comparison of execution time of one Erosion/Dilation operation and showing speed up with respect to the sequential implementation

Structuring Element		**PPE original optimized code(ms)**	**Proposed SPE code**		**Cell OpenCV performance (ms)**	
Size	Shape		Time (ms)	Speedup	Time (ms)	Speedup
7 x 7	Ellipse	89.806	0.311	289.6	0.973	92.4
5 x 5		47.74	0.290	164.6	0.562	84.9
3 x 3		16.324	0.261	62.5	0.308	52.9
7 x 7	Rectangle	24.235	0.310	78.2	0.337	71.9
5 x 5		18.782	0.280	67.1	0.289	64.9
3 x 3		14.341	0.260	55.2	0.276	51.9

for elliptical structuring elements. As shown in Table 1, our implementation successfully achieved very high speed up ratios (ranging from about 55 times to 290 times faster for different workload sizes) outperforming the reported Cell OpenCV benchmarks. The speedup increased for larger sized structuring elements i.e., as the amount of computation increased.

For evaluating our implementation of the connected components labeling algorithm, we compared the performance with different versions of the sequential and parallel code. We implemented sequential code using *Union-Find* algorithm (called $sequential\_UnionFind$) and *Floyd-Warshall* algorithm (called $sequential\_FW$) for execution only on the PPE. The experimental results for proposed parallel connected components labeling algorithm are presented with and without SIMD instructions for resolution of equivalence class labels, called $parallel\_CCL$ and $parallel\_CCL\_SIMD$ respectively, with execution on PPE and all 6 SPE's. Figure 5 shows the results for four different implementations of the CCL algorithm on different sized images, with the number of regions set to 250 and the total amount of foreground pixels set to 30% for all images. It was observed that $sequential\_FW$ gave better performance than $sequential\_UnionFind$

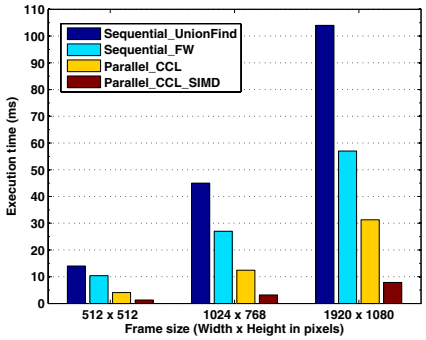

Fig. 5. Execution time per frame for four different implementations of CCL

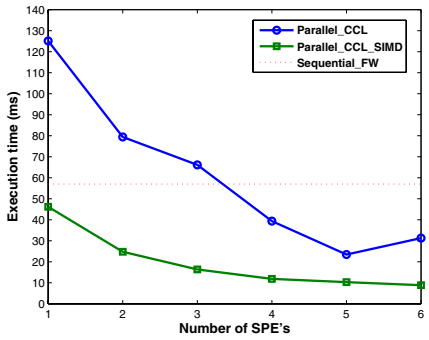

Fig. 6. Variation in parallel CCL performance across different number of SPEs

probably because branch instructions and recursions in $sequential\_UF$ code are not efficiently supported for execution. The average speedup for the parallel implementation without using SIMD instructions is about 2 times, increasing to 8 times with SIMD instructions. Figure 6 shows how the parallel performance scales as the number of SPES is varied. It can be observed that the execution time for the parallel implementation without SIMD instructions is greater than the serial execution time up to 3 SPEs due to the overhead of thread creation, data communication, merging the results, etc. which does not scale up linearly for a low number of SPE cores. Moreover the variation is not smooth as the total time is bounded by the maximum of the execution time over all SPEs. By varying the number of SPE's, the data partitioning area/boundary changes, which can cause increase or decrease in the number of regions and foregrounds pixels to be processed by any SPE depending on the distribution pattern in the image. This also explains the unexpected increase in the execution time for 6 SPE's as compared to 5. However, the optimized performance using SIMD instructions is always superior to the sequential version and scales up consistently with increasing number of SPE's.

6 Conclusions

This paper describes parallel algorithms for binary image morphology and connected components labeling suitable for SPMD multi-core architectures such as the Cell processor. The binary morphology operations were optimized for execution on the Cell SPE by representing each pixel as a single bit (packed into bytes) and utilizing bitwise comparison using AND/OR SIMD operations to process 128 pixels simultaneously. Novel bit level data access and alignment techniques for Cell BE were proposed in this context. The parallelization approach for the proposed CCL Cell implementation required a customized data partitioning and merging algorithm split between the SPEs and the PPE respectively using a fast SIMD version of the Floyd-Warshall equivalence resolution algorithm that was superior to the standard Union-Find search algorithm. Our implementation using 6 SPEs achieves a speedup of up to 290 times for processing erosion/dilation

operation using 7×7 pixel-sized elliptical structuring element with input images of 1024×768 and a speed up of about 8 times for connected components labeling on input images of 1920×1080 with 250 regions and 30% foreground pixels. Our future work will examine implementations on other multi-core architectures like GPUs as well as parallelizing other video processing tasks like normalized cross-correlation for image registration.

References

1. Bunyak, F., Palaniappan, K., Nath, S., Seetharaman, G.: Flux tensor constrained geodesic active contours with sensor fusion for persistent object tracking. J. Multimedia 2(4), 20–33 (2007)
2. Bunyak, F., Palaniappan, K., Nath, S.K., Baskin, T.I., Dong, G.: Quantitative cell motility for *in vitro* wound healing using level set-based active contour tracking. In: Proc. 3rd IEEE Int. Symp. Biomed. Imaging (ISBI), April 2006, pp. 1040–1043 (2006)
3. Bunyak, F., Palaniappan, K., Nath, S.K., Seetharaman, G.: Geodesic active contour based fusion of visible and infrared video for persistent object tracking. In: IEEE Workshop Applications of Computer Vision (WACV 2007), page Online (2007)
4. Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., Bosilca, G.: A rough guide to scientific computing on the Playstation 3. Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville (2007)
5. Chen, T.P., Budnikov, D., Hughes, C.J., Chen, Y.-K.: Computer vision on multi-core processors: Articulated body tracking. In: IEEE Int. Conf. Multimedia and Expo., pp. 1862–1865 (2007)
6. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.-Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the memory hierarchy. In: Proc. ACM/IEEE Conf. Supercomputing (2006)
7. Hidemasa, M., Munehiro, D., Hiroki, N., Yumi, M.: Multilevel parallelization on the Cell/B.E. for a motion JPEG 2000 encoding server. In: Proc. 15th Int. Conf. Multimedia, pp. 942–951 (2007)
8. Kolekar, M.H., Palaniappan, K., Sengupta, S., Seetharaman, G.: Semantic concept mining based on hierarchical event detection for soccer video indexing. J. Multimedia (2009)
9. Liu, L., Kesavarapu, S., Connell, J., Jagmohan, A., Leem, A., Paulovicks, L., Sheinin, B., Tang, V.L., Yeo, H.: Video analysis and compression on the STI Cell Broadband Engine processor. In: IEEE Int. Conf. Multimedia and Expo. (2006)
10. Manohar, M., Ramapriyan, H.K.: Connected component labeling of binary images on a mesh connected massively parallel processor. Computer Vision, Graphics, and Image Processing 45(2), 133–149 (1989)
11. Momcilovic, S., Sousa, L.: A parallel algorithm for advanced video motion estimation on multicore architectures. In: Int. Conf. Complex, Intelligent and Software Intensive Systems, pp. 831–836 (2008)
12. Palaniappan, K., Kumar, P., Ersoy, I., Davis, S.R., Bunyak, F., Linderman, M., Seetharaman, G., Linderman, R.: Parallel flux tensor for real-time moving object detection. Submitted to International Conference on Image Processing (2009)

13. Shyu, C.R., Klaric, M., Scott, G., Barb, A., Davis, C., Palaniappan, K.: GeoIRIS: Geospatial information retrieval and indexing system – content mining, semantics, modeling, and complex queries. IEEE Trans. Geoscience and Remote Sensing 45(4), 839–852 (2007)
14. Sugano, H., Miyamoto, R.: Parallel implementation of morphological processing on cell/be with opencv interface. In: 3rd Int. Symp. Communications, Control and Signal Processing, pp. 578–583 (2008)
15. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: Scientific computing kernels on the Cell processor. Int. J. Parallel Progrm. 35, 263–298 (2007)
16. Yu, J., Wei, H.: Video processing and retrieval on cell processor architecture. In: Ma, L., Rauterberg, M., Nakatsu, R. (eds.) ICEC 2007. LNCS, vol. 4740, pp. 255–262. Springer, Heidelberg (2007)
17. Zhou, L., Kambhamettu, C., Goldgof, D., Palaniappan, K., Hasler, A.F.: Tracking non-rigid motion and structure from 2D satellite cloud images without correspondences. IEEE Trans. Pattern Analysis and Machine Intelligence 23(11), 1330–1336 (2001)