Real-Time Object Detection for Multi-Camera on Heterogeneous Parallel Processing Systems

Chih-Sheng Lin, Shih-Meng Teng, Yen-Ting Chen, and Pao-Ann Hsiung
National Chung Cheng University, Chiayi, Taiwan
{lcs98p, tsm100m, cyt100m, pahsiung}@cs.ccu.edu.tw

***Abstract*—In recent years, the need for object detection has significantly increased for multi-camera systems. However, the detection methods in such systems incur high computational cost, which leads to a major challenge in real-time applications. In this work, we propose a Scissor Algorithm for object detection using a multi-core CPU and a graphic processing unit (GPU). Leveraging the features of both the CPU and the GPU, the object detection method was enhanced in two stages: (a) pixel-to-pixel color filtering and (b) grouping. The proposed algorithm can effectively shrink the search area for detection and further improve the process of detection, thus effectively increasing the frame rate for real-time applications. Experimental results demonstrate the real-time performance of the proposed algorithm.**

***Keywords*-Object Detection, Computer Vision, Real-Time, Heterogeneous Parallel Processing, Graphic Processing Unit.**

I. Introduction

Object detection has mostly been applied to the systems of traffic sign [1] [2], face recognition [5] [11], and event-based auto-recording [4] [9] [12]. In most cases, these systems are set with multiple cameras for detecting the object per frame received by each camera which is at different location. As many approaches proposed by researchers and the progress of hardware platforms, the detection accuracy has significantly increased. However, for real-time feature of the mentioned systems, the main concern is the highly computational cost affecting the real-time performance.

There are physical characteristics used to detect objects: color, edge, and shape of objects. To those systems without classifier and training examples, they usually adopt color-based approaches for fast processing of object detection. The color-based approaches first filter each pixel in one video frame with a pre-defined color threshold or range to eliminate the noise and further extract out the candidates of target object, and then to group those pixels which are determined as fragments to form the shape for the candidates of target object. Finally, the target object is detected by the most fitting shape among the candidates.

Multicore CPU and graphic processing unit (GPU) have recently used to achieve the real-time performance of computer vision [3] [10] [13]. However, as the consideration of performance, they are designed for different types of task. For applications of high parallelism and large input data size such as image processing, GPU can achieve the superior performance. On the other hand, multicore CPU can achieve the better performance for those applications of light parallelism or heavy control-oriented. In this work, we propose Scissor Algorithm for object detection using a heterogeneous parallel processing system (HPPS) which consists of a multi-core CPU and a GPU. Leveraging the advantages of CPU and GPU, we enhance the proposed algorithm in two stages: pixel-to-pixel color filtering and grouping. The proposed algorithm can effectively shrink the search area for detection and improve the process of detection further increase frame rate for real-time applications.

This paper is organized as follows: Section II briefly introduces the framework of color-based object detection and the background of Thread Building Block (TBB) and Compute Unified Device Architecture (CUDA). Section III introduces our proposed method - Scissor Algorithm. Experimental results are shown and discussed in Section IV. Section V draws the conclusion.

II. Background

In this section, we first introduce our framework of color-based object detection. In order to explain the factors of performance improvement for color filtering and grouping to form the shape of target object, we then give the background knowledge of TBB and CUDA for further introducing the proposed algorithm.

A. Color-based Object Detection

Figure 1 shows our framework of color-based object detection. The framework mainly consists four parts: Color Filter, Grouping, Shape Filter, and Circle Object. Assume there are a set of video frames from different cameras concurrently, that is, a set of frames $F[height][width]$. We explain each part in details in the followings:

- *Color Filter*: In order to efficiently eliminate the noises which are not targeted, we adopt Color Filter to this process. Given a pre-defined range of color value, denoted as δ, we have the color-filtered matrix $CF[height][width]$ as follow:

$$CF[i][j] = \begin{cases} 0, & \text{if } F[i][j] \text{ is not in } \delta \\ 1, & \text{if } F[i][j] \text{ is in } \delta \end{cases} \tag{1}$$

, where $0 \leq i \leq height$, $0 \leq j \leq width$.

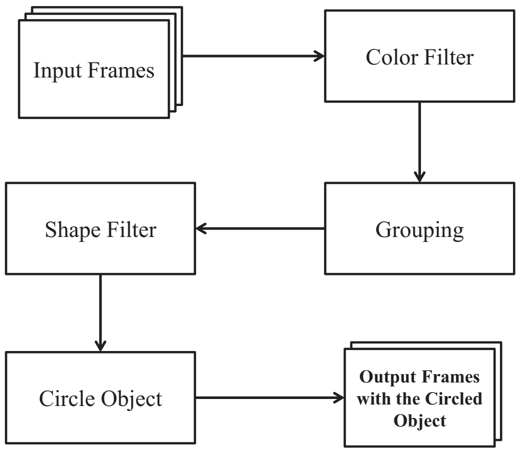

Figure 1. Framework of Color-based Object Detection.

- *Grouping*: Grouping is the most essential part in our framework of object detection. The purpose of Grouping is to assemble those adjacent pixels whose color values are within the given range into a group. The group number GN of every pixel is determined by Table I. There are two conditions explained as follows: `new_number()` is used to set a new group number which the value of a filtered pixel is one and its up and left adjacency are not one, and `connect()` represents the value of current pixel is one and it belongs to the same group with its upper and left adjacency.

Table I
The Group Number Corresponding to the Condition

G[i][j]	Condition
0	$CF[i][j] = 0$
`new_number()`	$GN[i-1][j] = GN[i][j-1] = 0$
$GN[i-1][j]$	$GN[i-1][j] \neq 0$ and $GN[i][j-1] = 0$
$GN[i][j-1]$	$GN[i][j-1] \neq 0$ and $GN[i-1][j] = 0$
`connect()`	$GN[i-1][j] \neq 0$ and $GN[i][j-1] \neq 0$

- *Shape Filter*: The pixels which are belong to the same group are labeled as the same group number and form a segment. Shape Filter eliminates those segments are larger than the given threshold.
- *Circle Object*: By analyzing each segment of candidate object with its upmost, leftmost, rightmost and bottom point, the center of candidate object is determined and the most fitting object will be circled.

In order to analyze the workload of each stage of our proposed framework, we test one video on a single-core CPU platform. Figure 2 shows the feature of target object in our testing video which is a basketball game and our target object is the basketball. More in details, this video consists of 1071 frames and there are 632 frames which contain the basketball set to be detected. The number of segments of size 0 100 is extremely more than other intervals because of the distance and angle between the camera and court. As Figure 3 shown, the elapsed time ratio of Color Filter and Grouping are 21.17% and 46%, respectively. According to Amdahl's Law, the most performance improvement for the framework of object detection will be reached by enhancing Color Filter and Grouping. Note that Other consists of file I/O, format convert, etc.

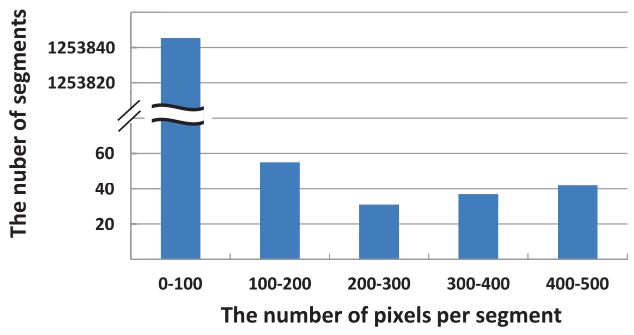

Figure 2. The feature analysis of object in a testing video.

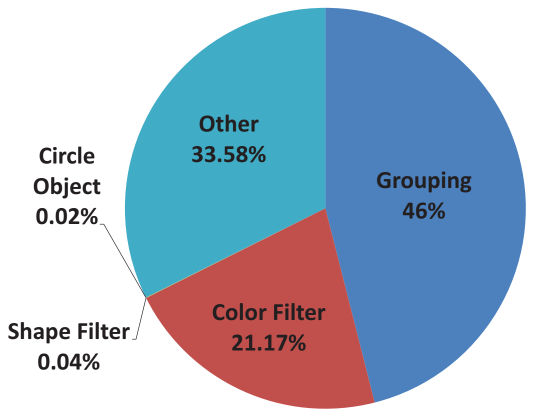

Figure 3. The Elapsed Time Ratio of Functions in Object Detection Algorithm (Serial).

B. Thread Building Blocks (TBB)

Multi-core processors today has become more and more popular, however, writing an efficient scalable parallel program is tedious via existing threading packages. In order to address the barriers Intel Threading Building Blocks (TBB) [8] provides a library, which is based on template and C++ concept of generic programming, to help programmers to leverage the multi-core performance. TBB is not only a thread-replacement library, but also a higher-level, task-based parallelism that abstract platform details and threading mechanism for performance and scalability.

TBB uses template to express parallelism in C++ and the concept of nested parallelism is fully supported by it; with this ability, programmers using TBB can divide their large

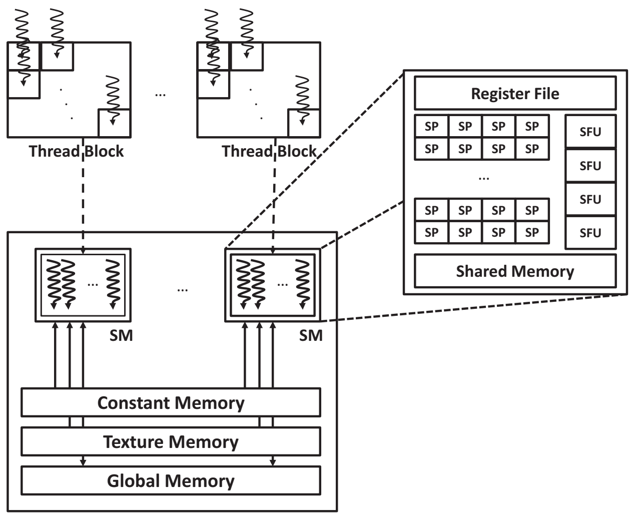

Figure 4. CUDA Architecture.

parallel components into smaller parallel components. The programmers need not to be an expert in threading programming and need not to be concerned with the synchronization, load balancing, and cache optimization in their applications. With the abstraction of platform details, TBB can run both in systems with a single processor core, or systems with multiple processor cores. Additionally, TBB also realizes the concept of scalability of writing an efficient scalable program, that is a program can benefit from the increasing number of processor cores.

C. Compute Unified Device Architecture (CUDA) [6]

Compute Unified Device Architecture (CUDA) is a GPU programming model developed by NVIDIA. CUDA provides a programming interface [7] to utilize the highly-parallel nature of GPUs and hide the complexity of controlling GPUs. A programmer specifies a kernel as a series of instructions and a data set, then the kernel is executed by thread blocks, each of which consists of a number of threads with unique IDs. For independent executions among different thread blocks, synchronization is required to avoid the race condition. Note that it is more expensive for synchronizing thread blocks than synchronizing threads in a thread block.

At runtime, thread blocks are distributed to streaming multiprocessors (SMs) as shown in Fig. 4. Precisely, a thread block is divided into warps of 32 threads. Each warp is executed by an SM within 4 cycles if the input data is cached for computing. For utilizing an SM efficiently, the threads of each warp should execute the same instructions. Otherwise, it will be executed sequentially with overhead for stalling if threads execute different instructions. This scenario is called warp divergence.

As to the type of memory as listed in Table II, each kind of memory is designed from different read or write speeds, hardware, and programming scopes. Global memory, as a bridge between host and device, is shared among all thread blocks. Constant and Texture memories are read-only memories for faster memory accesses. In a thread block, the most essential component is shared memory. Being different from registers used only by a thread, shared memory allows threads in a block to communicate with each other to reduce the overhead of accessing global memory.

Table II
COMPARISONS OF MEMORY TYPE

Attributes	**Register**	**Shared Memory**	**Const./Text. Memory**	**Global Memory**
Scope	Thread	Block	Grid	Grid
Hardware	On-chip	On-Chip	DRAM	DRAM
Access Type	Read-Write	Read-Write	Read-Only	Read-Write
Access Latency	Immediate	4 cycles	Immediate	400-600 cycles

III. SCISSOR ALGORITHM

In this section, we will introduce two parts of Scissor Algorithm for color-based object detection: GPU side and CPU side, respectively. As shown in Figure 5, every video frame was sent to server for detecting object by each camera concurrently. Then, the frames were processed by three stages on GPU: Format Converter, Color Filter, and Region Projector. As to CPU side, there is a stage: Region-based Collector.

Algorithm 1 shows the GPU part of Scissor Algorithm. In Algorithm 1, line 3 represents every thread fetch $YCrCb$ value of each pixel from global memory. Line 7 to 11 represents every $YCrCb$ value was filtered by the pre-defined range in parallel. Region Projector was shown in line 14 to 18. Note every stage on GPU part of Scissor Algorithm is highly parallel, that is, each unit of data to be processed is independent with its adjacency.

As to the CPU part of Scissor Algorithm, it is shown in Algorithm 2. We mainly focus on Region-based Collector which efficiently shrink the region for grouping by $ProVec_x$ and $ProVec_y$ sent from GPU. Note that line 18 is Grouping which is processed by `parallel_for` of TBB library.

IV. EXPERIMENTS

We discuss two major experiments for Scissor Algorithm compared to coarse-grained parallel version of our proposed framework of color-based object detection. Our environmental setup is shown in Table III and the $height$ and $width$ of a frame is 720 and 1080, respectively. Moreover, the feature of each video sequence from every camera in our experiments is very similar as Figure 2 which we discussed in Section II.

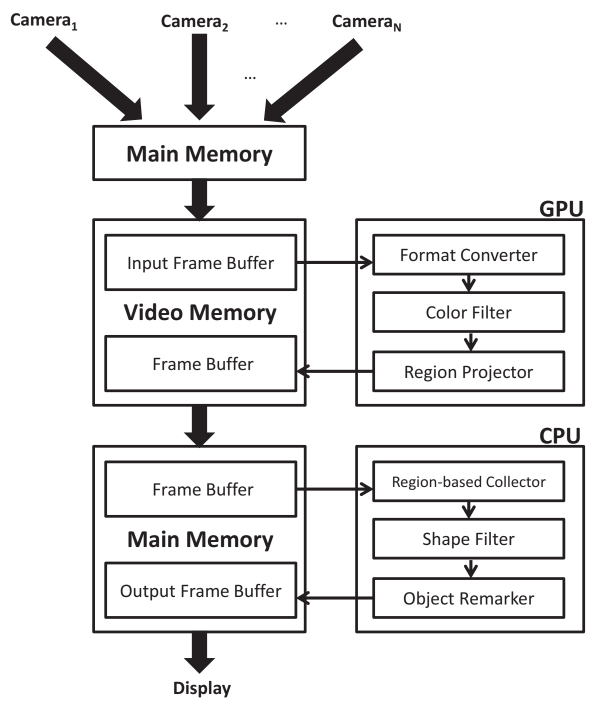

Figure 5. The flow of proposed object detection method on a heterogeneous parallel processing platform

Table III
THE ENVIRONMENTAL SETUP OF EXPERIMENTS

CPU	Intel(R) Core(TM) i7-2600 CPU (4 cores@3.40GHz)
GPU	NVIDIA Tesla C2050 (448 CUDA cores@1.15GHz, 2.5 GB Global Memory)
MEM	DDR3 16 GB Memory
OS	Linux kernel 2.6.1

Algorithm 1 Scissor Algorithm (GPU side)

```
1: Transfer Image Data: Cameras to Main Memory
2: Transfer Image Data: Main Memory to Global Memory
3: Fetch $y$, $cb$, $cr$ from global memory
4: Allocate $ProVec_x$ and $ProVec_y$ on Global Memory
5: Allocate $CF$ on shared memory
6: //Thread Indexing
7: $tid\_x = blockIdx.x \times blockDim.x + threadIdx.x$;
8: $tid\_y = blockIdx.y \times blockDim.y + threadIdx.y$;
9: //Color Filter
10: if ($y[tid\_x][tid\_y]$ is not in $\delta_y$) or
    ($cb[tid\_x][tid\_y]$ is not in $\delta_{cb}$) or
    ($cr[tid\_x][tid\_y]$ is not in $\delta_{cr}$) then
11:     $CF[tid\_x][tid\_y] = 0$;
12: else
13:     $CF[tid\_x][tid\_y] = 1$;
14: end if
15: Sync_barrier
16: //Region Projector
17: Fetch $CF$ from global memory
18: if $CF[tid\_x][tid\_y] = 1$ then
19:     $ProVec_x[tid\_x] = 1$;
20:     $ProVec_y[tid\_y] = 1$;
21: end if
22: Transfer $CF$, $ProVec_x$, and $ProVec_y$: Global Memory
    to Main Memory
```

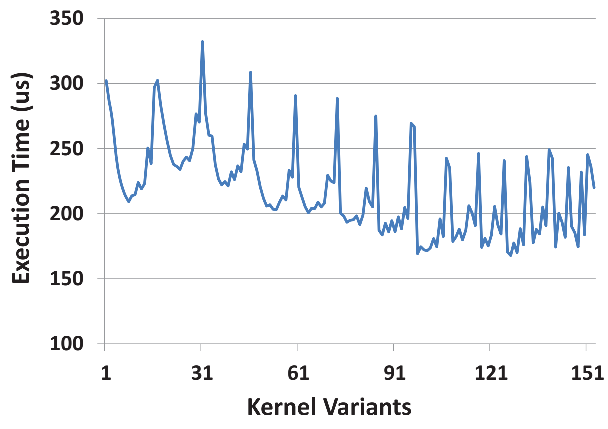

Figure 6. The autotuning sweep of Scissor Algorithm (GPU part).

A. Performance Optimization on GPU

The first experiment is to find out the parameters for optimal performance of Scissor Algorithm. As mentioned in Section II-C, the dimension of a thread block which is an important parameter for performance is determined by a programmer. Therefore, the two dimension of tid_x and tid_y of a tile in $F[height][width]$, $blockDim.x$ and $blockDim.y$, are the parameters for tuning. Figure 6 shows the autotuing sweep of the mentioned parameters, and each unique combination forms a kernel variant. By exploring the feasible kernel variants, we find that the optimal kernel variant of ($blockDim.x$,$blockDim.y$) is (40,9).

B. Coarse-grained Parallelism versus Scissor Algorithm

The second experiment is the comparison of coarse-grained version of the mentioned framework and Scissor Algorithm. The scenario of coarse-grained parallelism is each video frame processed by one CPU thread in parallel. Figure 7 shows the comparison of the two mentioned methods. We can see the trend of speedup of Scissor Algorithm to coarse-grained parallelism is increasing while the number of cameras is increasing. The speedup is at most 6.15 when the number of camera is 8.

V. CONCLUSIONS

In this paper, we proposed Scissor Algorithm for object detection on a heterogeneous parallel processing system

Algorithm 2 Scissor Algorithm (CPU side)

```
1:  Fetch $ProVec_x$ and $ProVec_y$ from Main Memory
2:  //Region-based Collector
3:  while ($ProVec_x[iter.x] = 1$) and ($iter.x < width$) do
4:      $Table[thread\_index][0] = iter.x;$
5:      if ($ProVec_x[iter.x + 1] = 0$) and ($iter.y < height$)
        then
6:          $Table[thread\_index][1] = iter.x;$
7:          while $ProVec_y[iter.y] = 1$ do
8:              $Table[thread\_index][2] = iter.y;$
9:              if $ProVec_y[++iter.y] = 0$ then
10:                 $Table[thread\_index][3] = iter.y;$
11:                 $thread\_index++;$
12:             end if
13:         end while
14:         $iter.y++;$
15:     end if
16:     $iter.x++;$
17: end while
18: Grouping($Table[thread\_index][0]$,
    $Table[thread\_index][1]$,$Table[thread\_index][2]$,
    $Table[thread\_index][3]$);
```

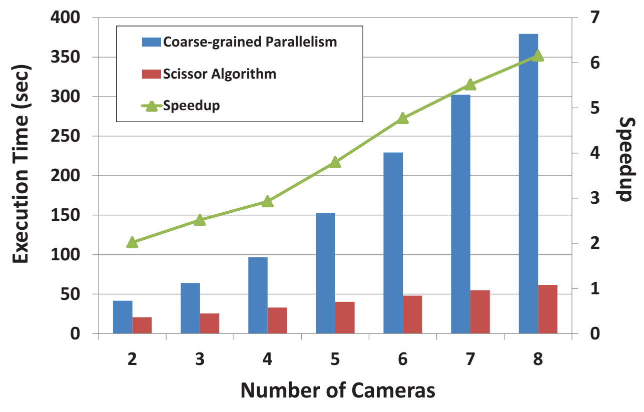

Figure 7. The comparison of coarse-grained parallelism and Scissor Algorithm

which consists of a multi-core CPU and a GPU. By leveraging the advantages of CPU and GPU, Scissor Algorithm reaches 6.15 times faster than the coarse-grained parallelism of color-based object detection. In the object detection system with multiple cameras, Scissor Algorithm is suitable for satisfying the soft real-time requirements. Our future work is to analyze the critical factors based on various GPU architectures and applications of object detection for optimal real-time performance.

References

[1] C. Bahlmann, Y. Zhu, V. Ramesh, M. Pellkofer, and T. Koehler. A system for traffic sign detection, tracking, and recognition using color, shape, and motion information. In *Proceedings of Intelligent Vehicles Symposium*, pages 255–260. IEEE, 2005.

[2] S. Estable, J. Schick, F. Stein, R. Janssen, R. Ott, W. Ritter, and Y. Zheng. A real-time traffic sign recognition system. In *Proceedings of the Intelligent Vehicles Symposium*, pages 213–218. IEEE, 1994.

[3] S. Fukui, Y. Iwahori, and R. Woodham. Gpu based extraction of moving objects without shadows under intensity changes. In *IEEE World Congress on Computational Intelligence Evolutionary Computation*, pages 4165–4172, 2008.

[4] M. Hu, M. Chang, J. Wu, and L. Chi. Robust camera calibration and player tracking in broadcast basketball video. *IEEE Transactions on Multimedia*, 13(2):266–279, 2011.

[5] S. Li and A. Jain. *Handbook of face recognition*. Springer-Verlag New York Inc, 2011.

[6] C. Lin, W. Liu, W. Yeh, L. Chang, W. Hwu, S. Chen, and P. Hsiung. A tiling-scheme viterbi decoder in software defined radio for gpus. In *International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM)*, pages 1–4. IEEE, 2011.

[7] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*. 2010.

[8] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[9] J. Ren, J. Orwell, G. Jones, and M. Xu. Tracking the soccer ball using multiple fixed cameras. *Computer Vision and Image Understanding*, 113(5):633–642, 2009.

[10] S. Sinha, J. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 22(1):207–217, 2011.

[11] M. Turk and A. Pentland. Face recognition using eigenfaces. In *Proceedings of Computer Vision and Pattern Recognition*, pages 586–591. IEEE, 1991.

[12] H. Zhang, Y. Wu, and F. Yang. Ball detection based on color information and hough transform. In *International Conference on Artificial Intelligence and Computational Intelligence*, volume 2, pages 393–397. IEEE, 2009.

[13] L. Zhang and R. Nevatia. Efficient scan-window based object detection using gpgpu. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–7, 2008.