2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks

Parallel Implementation and Performance Prediction of Object Detection in Videos on the Tilera Many-core Systems

Ya-Fei Hung, Shau-Yin Tseng
SoC Technology Center
Industrial Technology Research Institute
Hsinchu, Taiwan 310
{*yafeihung, tseng*}*@itri.org.tw*

Chung-Ta King, Huan-Yu Liu, Shih-Chieh Huang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
king@cs.nthu.edu.tw

***Abstract*—Object detection plays an important role in intelligent video analysis. Unfortunately, its heavy computational complexity makes it very difficult to process in real time. Some recent studies use multi-core platforms to achieve the required performance. In this paper, we study the problem under the context of many-core platforms, e.g. for application-specific, embedded systems. We first show how object detection can be parallelized for many-core platforms and then discuss how its performance can be predicted for embedded system designs. The parallel algorithm is verified with a real implementation on a 64-core TILERA. Our implementation achieves a speedup of 37.20 with 56 cores and a processing rate of 18 frames per second for full-HD $(1920 * 1080)$ videos. Our performance prediction equation is also evaluated using the implementation and the predicted performance is very close to real results.**

***Keywords*-parallel processing; object detection; speedup; performance prediction; many-core architecture;**

I. Introduction

With the advances of IC technologies, multi-core processors are now commonplace. Many researchers have tried to use multi-core platforms to implement applications requiring a large amount of calculations, such as object detection and object tracking [1], [3], [4] , and achieve satisfactory results. Looking forward, the number of cores per chip will increase, while at the same time applications require even more computing powers, e.g. processing full-HD videos. It is necessary to understand how the many cores can be efficiently utilized and how applications scale with the increasing number of cores. More importantly, we need to be able to predict the performance on such many-core systems so as to use the resources mostly efficiently.

In this paper, we use object detection in videos as an example to study its parallel implementation on many-core platforms and the resultant performance. We also derive its performance equation for predicting the performance and designing the system. The challenges in deriving the equation are the need to account for the overhead of the connection between cores during execution and the different degree of parallelism on different parts of the algorithm. We discuss how these difficulties are overcome in our equation. The parallel algorithm is verified with a real implementation on a 64-core TILERA. Our implementation achieves a speedup of 37.20 with 56 cores and a processing rate of 18 frames per second for full-HD $(1920 * 1080)$ video.

The remainder of the paper is organized as follows. In Section II, we introduce the background of object detection. In Section III, we describe our object detection algorithm. In Section IV, we discuss how to parallelize the object detection algorithm. In Section V, we describe our experiments using a real implementation on the TILERA system. In Section VI, we compare our work with previous works. Section VII is the conclusion.

II. Background

Object detection is to detect where moving objects are in a video. The most common object detection technique is to subtract the background from the input video frames. The result of this step is called *background difference*.

We denote a pixel (x, y) in the current frame to be $C(x, y)$ and that in the background as $B(x, y)$. Then, the background difference $D_b(x, y)$ of pixel (x, y) is calculated as follows.

$$D_b(x, y) = |C(x, y) - B(x, y)| \tag{1}$$

Apparently, if $D_b(x, y) \neq 0$, then the pixel (x, y) is contained within an object.

Calculating *frame difference* is also a commonly used technique in object detection. Frame difference is to calculate the difference of temporally adjacent frames. This information is used to observe the movement of objects. Again, let a pixel (x, y) in the current frame be $C(x, y)$ and that in the previous frame be $P(x, y)$. The frame difference $D(x, y)$ of pixel (x, y) is given below.

$$D(x, y) = |C(x, y) - P(x, y)| \tag{2}$$

After identifying which pixels may belong to an object, the next step is to remove noises and to separate objects that are close to each other but regarded as one object. The general methods to separate the objects are *erosion* and *dilation*. And the method to remove noises is *low-pass filter*.

a	b	c
d	e	f
i	j	k

Figure 1. The mask window of erosion.

a	b	c	d	e
e	g	i	j	k
l	m	n	o	p
q	r	s	t	u
v	w	x	y	z

Figure 2. The mask window of dilation.

Erosion and dilation are usually executed on binary images. We denote a pixel (x, y) in the input binary image as $B_{in}(x, y)$. For each pixel $B_{in}(x, y)$, we consider a window mask such as that shown in Figure 1. In the figure, e is $B_{in}(x, y)$ and a, b, c, d, f, i, j, k are the neighbors. If all the nine pixels are 1, then the value of $B_{in}(x, y)$ is 1. Otherwise, $B_{in}(x, y)$ is 0. Dilation is just the opposite. For each pixel $B_{in}(x, y)$, we consider a window mask such as that shown in Figure 2. The pixel n is $B_{in}(x, y)$. In the binary image, if $B_{in}(x, y)$ is 1, then all the pixels within this $5 * 5$ mask window are 1.

The low-pass filter is to average all values within the mask window to be the value of the center pixel. Figure 3 shows an example of the low-pass filter with the size of mask window to be $5 * 5$.

$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$

Figure 3. The window mask of low-pass filter.

III. Object detection algorithm

In this section, we describe the object detection algorithm used in this paper, which is according to the sequential algorithm proposed in [2]. That algorithm is shown in Figure 4. The main procedure consists of three parts:

1) Frame difference and background difference.
2) Post-processing.
3) Background update.

The details are described in the following subsections.

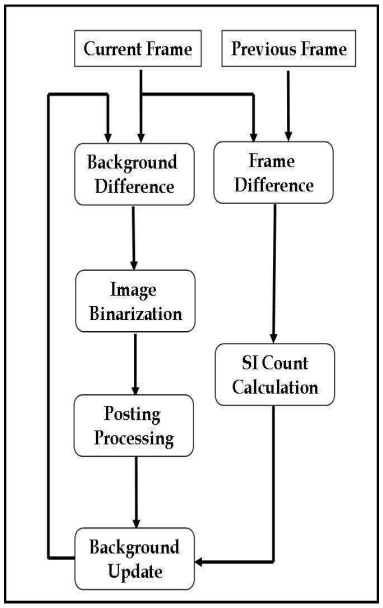


Figure 4. The object detection algorithm.

A. *Frame difference and background difference*

The first step in the algorithm is to calculate the frame difference between the previous frame and the current frame. The result of frame difference is used to calculate *stationary indexes* (*SI counter* in Figure 4), which indicate how many frames the pixels have been stationary. This information is used for background update discussed in section III-C.

Then, we compute the background difference to identify the objects. The results are binarized by a threshold T_b.

$$B_{in}(x, y) = \begin{cases} 1, & \text{if } D_b(x, y) \geq T_b \\ 0, & \text{if } D_b(x, y) < T_b \end{cases} \tag{3}$$

$B_{in}(x, y)$ is the binary image after calculation and $D_b(x, y)$ is the result of background difference. In the binary image, if the value of the pixel is 1, this pixel is inside an object; otherwise it is in the background.

B. *Post-processing*

The second step is to post-process of the binary image $B_{in}(x, y)$. The goal is to separate the connected objects and filter out the noises. In this step, we execute erosion first, followed by dilation and low-pass filter.

In our algorithm, the size of the mask window of erosion is $3{*}3$ and that of dilation is $5{*}5$. However, the mask window of low-pass filter changes according to the inputs. By using different sizes and shapes of the mask window for low-pass filter with an appropriate threshold, we can filter out the noises and smooth out the shape of the objects. Generally, if the shape of an object is square, a square mask window should be used. If the shape of an object is rectangle, a rectangle mask window should be used. Moreover, for small objects, the mask window should not be too large. However, for large objects, the size of the window should be according to the size of the noises. If the noises are small, we use small mask windows. If the noises are large, we use large mask windows to filter out the noises. For different inputs, we can change the mask window of low-pass filter to get better results.

C. *Background update*

The final step is to update the background based on the stationary index, which indicates how long this pixel has been stationary. We define a threshold T_s to determine how long a pixel has been stationary. For example, suppose T_s is set to 30. That means if a pixel has been stationary for 30 frames, it is classified as being in the background. Note that T_s can change with different inputs and requirements. For example, suppose that a person in the video stands talking for some time. If we do not want the person to be classified as background, the value of T_s should be larger.

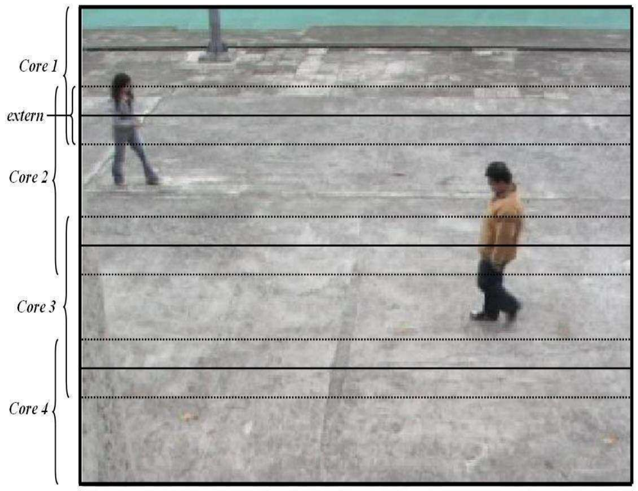


Figure 5. A frame is partitioned into four regions for four cores.

IV. Parallel implementation

In this section, we discuss our parallelization of the object detection algorithm on many-core architectures. We adopt the data partition strategy to parallelize the algorithm. Data partition is to split up the input data into many partitions and assign each partition to execute on one core.

In our implementation, if the input frame size is $height * width$, we divide it into $core\_number$ partitions vertically, where $core\_number$ is the number of cores used. Figure 5 shows the partition method. To reduce the communication overhead between cores, we fetch overlapping pixels at the top and the bottom of each partition. The amount of overlapping data to access is $extern * width$ pixels. Figure 5 also shows the overlapping data. Note that the number of extern is according to the mask window size of low-pass filter. Each core, besides the cores executing the top and bottom partitions of a frame, loads $\frac{height*width}{core\_number+externwidth}$ pixels at the beginning. Hence, the amount of data in all cores during execution is $(height*width)+extern*width*(core\_number-1)$ pixels.

By focusing on the data, we can derive an equation to characterize the speedup of the parallel object detection algorithm. First, the optimal speed up with $core\_number$ cores is $core\_number$. However, with reduplicated data to process on the cores, the speedup of the many-core processor would be limited as follows.

$$speed\_up = \frac{core\_number}{(\frac{extern*(core\_number-1)}{height} + 1)} \quad (4)$$

With different sizes of mask windows, we can calculate and predict the speedups with different number of cores. Figure 6 shows the predicted speedup using Equation (4). The different curves show the speedup with different values of extern. In the next section, the predicted speedups will be compared with results from actual executions on real machines.

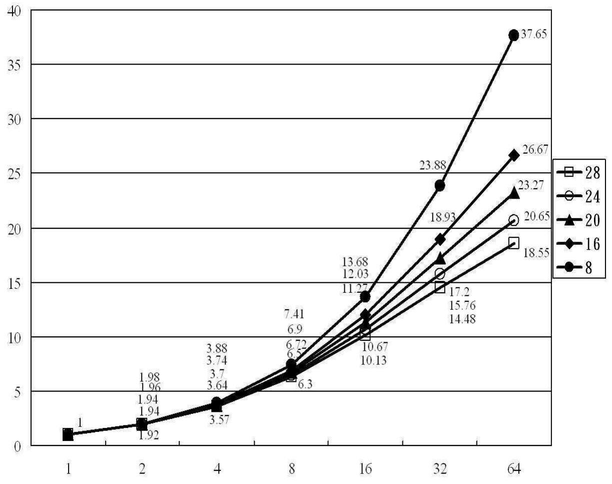
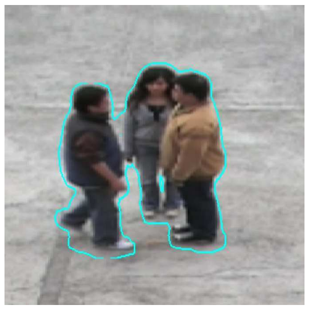


Figure 6. The speedup predicted by Equation (4).

V. Experiments

We have implemented our parallel object detection algorithm on a 64-core TILERA. Each core runs at 700MHz with a 16 KB L1 cache (8KB for instruction and 8KB for data) and a 64 KB L2 cache. During execution, users can only use 56 among the 64 cores. The input video in our experiments is in full-HD. After the video is input, we down-sample it to $960 * 540$ and before output, up-sample the video to $1920 * 1080$. We used two videos in our experiments. Video 1 has a better result when the mask window of low-pass filter is $15 * 5$ (see Figure 7 (a)), while video 2 is better with a $5 * 5$ mask window (see Figure 7 (b)).


(a) Output of Video 1

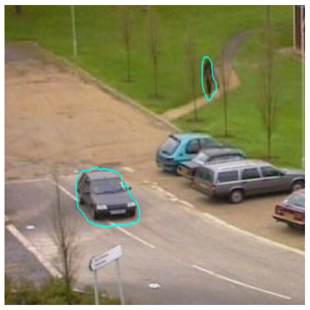

(b) Output of Video 2

Figure 7. Outputs of the experiments.

For Video 1, the speedups with different cores and *extern* are shown in Figure 8. When *extern* is 28, we can get the best detection result. The processing speed is 10.2 frames per second with full-HD videos and the speedup is 21.47 on 56 cores. For Video 2, the speedups with different cores and *extern* are shown in Figure 9. When *extern* is 8, we can get the best detection result. The processing speed is 18.1 frames per second with full-HD videos and the speedup is 37.2 on 56 cores.

Each curve in Figures 8 and 9 shows a different size of the mask window of low-pass filter. The top curve corresponds to *extern* $= 8$ (after down-sampling, the mask window size is $5 * 5$). The bottom curve is for *extern* to be 28, with a mask window of size $15 * 5$. Note that using more cores, the overlapping data fetched by the cores also increase. For example, when *extern* is 8, the redundant data fetched in the 2-core case are $(8 * width)$ pixels and the redundant data of 56-core case are $(55 * 8 * width)$ pixels. We can thus see that using more cores, *extern* would have more effects upon the speedup.

To compare the predicted speedups in Figure 6 and the experimental data in Figures 8 and 9, we can see that they are very close. This shows that Equation (4) can accurately predict the speedup performance. For some cases, we can get better speedups than those predicted. This is because the fewer cores used, the more data would be in one core. In the extreme case in which a single core is used, the amount of data will be too large to fit into the cache. Some data must be saved in the external memory and loaded into cache when needed. This process will take time and thus slows down the execution.

VI. Comparison with previous approaches

Many researchers have implemented object detection and object tracking algorithms on multi-core platforms. In [4], Trista et al. implemented articulated body tracking on Intel Xeon with eight cores. They experimented on VGA $(640 * 480)$ video and obtained a 6.54 speedup with eight cores. The parallelization strategy used was data domain parallelization by frame and different particles.

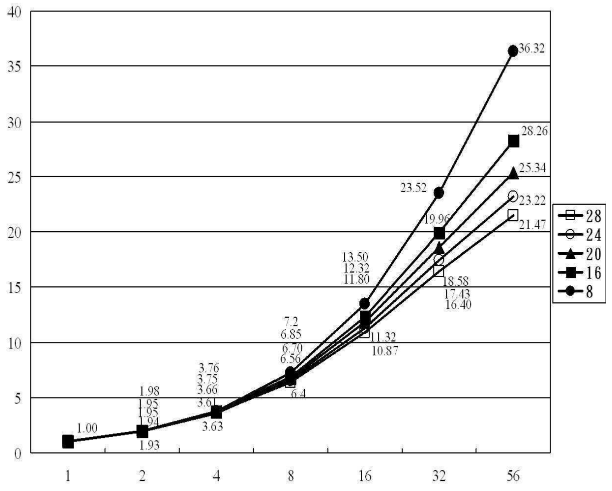


Figure 8. The data of experiments of Video 1.

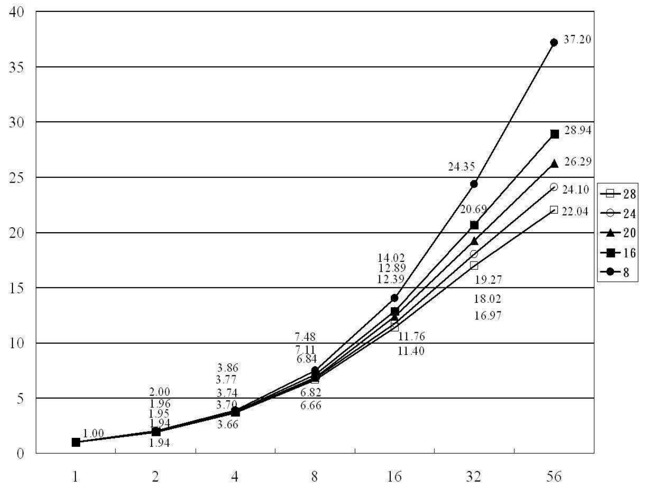


Figure 9. The data of experiments of Video 2.

In [1], Patricia et al. implemented moving object detection on Intel XeonMP with eight cores. They segmented the input frame into tens of regions and determined which pixel having a closer relation with the object. They can process 51.1 frames per second using CIF $(352 * 288)$ videos and achieve a speedup of 6.64.

In [3], Hernry et al. presented an implementation of object tracking on an IC3D/Xetal SIMD processor, which consists of 320 RISC processors. This is the only previous work that used more than eight cores. The main method was to parallelize the histogram computation. When processing VGA videos, their implementation can process object tracking at 30 frames per second.

In all these previous works, the input videos were VGA, and most of them only use eight-core platforms. In contrast, our parallel object detection algorithm can process full-HD $(1920 * 1080)$ videos. Ours and [3] were executed on many-core architectures. Table I. summarizes the differences.

TABLE I. COMPARISON WITH PREVIOUS WORKS.

	input size	***fps***	***speed up with 8 cores***
Trista et al. [4]	VGA		6.54
Patricia et al. [1]	CIF	51.1	6.64
Hernry et al. [3]	VGA	30	
our work	full-HD	18.1	7.48

VII. Conclusion

In this paper, we discuss the design of a parallel object detection algorithm for many-core architecture and the corresponding performance prediction equation. To achieve the better result, we change the size of low-pass filter according to the inputs. The algorithm is implemented and executed on Tilera. It can achieve a processing speed of 18.1 frames per second with full-HD videos and a speedup of 37.20 on 56 cores. The experiments also show that the predicted speedups using our derived performance equation are very close to the actual data. Therefore, we can use the equation to predict the performance of this implementation.

We plan to extend this work by parallelizing applications of different characteristics on the many-core architecture to better understand such machines. We also plan to better understand many-core systems, such as Tilera, to exploit further opportunities to achieve an even higher speedups.

References

[1] Patricia P. Wang, Wei Zhang, Jianguo Li, Yimin Zhang, *Realtime Detection of Salient Moving Object: A Multi-core Solution*, IEEE Proceedings of International Conference on Acoustics, Speech, and Signal Processing,PP. 1481-1484, 2008.

[2] Shao-Yi Chien, Yu-Wen Huang, Bing-Yu Hsieh, Shyh-Yih Ma, and Liang-Gee Chen, *Fast Video Segmentation Algorithm With Shadow Cancellation, Global Motion Compensation, and Adaptive Threshold Techniques*, IEEE Transactions on Multimedia,PP. 732-748, 2004.

[3] Henry Medeiros, Xinting Gao and Johnny Park, *A Parallel Implementation of the Color-Based Particle Filter for Object Tracking*, Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems, 2008.

[4] Trista P. Chen, Dmitry Budnikov, Christopher J. Hughes, and Yen-Kuang Chen, *Computer Vision on Multi-core Processors: Articulated Body Tacking*, Proceedings of IEEE International Conference on Multimedia and Expo,PP. 1862-1865, 2007.