NOVEL PARALLEL HOUGH TRANSFORM ON MULTI-CORE PROCESSORS

Yen-Kuang Chen, Wenlong Li, Jianguo Li, and Tao Wang
Corporate Technology Group, Intel Corporation

ABSTRACT

After analyzing the performance bottlenecks of the Hough Transform on multi-core processors, this paper proposes a new Hough Transform implementation. The performance of microprocessors improves significantly because of the introduction of multiple cores. To harness the computation power of such multi-core processors, we must effectively execute many threads at the same time. This paper first studies a coarse-grain and a fine-grain parallelization of a straightforward Hough Transform implementation on an 8-core machine. Due to parallelization overheads and memory requirements, these schemes do not fully utilize computation power. After that, we propose a new Hough Transform implementation for parallelization. Experimental data shows that the new Hough Transform exposes a significant amount of concurrency and pretty good data locality. On the 8-core machine, the new implementation has 25% better performance than the old ones.

***Index Terms*—**Parallel algorithms, parallel processing, digital signal processors

1. INTRODUCTION

Hough Transform [1] is a standard tool in image analysis and computer vision for feature extraction. It recognizes global patterns in an image space by identifying local patterns (ideally a point) in a transformed parameter space, i.e. Hough space. The basic idea of this technique is to find curves that can be parameterized, like straight lines, polynomials, circles, ellipse, and so on, in a suitable parameter space. Therefore, detecting curves reduces to detecting local maxima in the Hough space to which a large number of pixels from image space are mapped. The main advantages of Hough Transform are the robustness to discontinuous pixels and noise in real world images.

However, Hough Transform is very time consuming. On a machine with 2.8GHz Intel® Xeon® processor, the Hough Transform takes 1.4 seconds to process a single DVD frame. This is 42 times slower than real-time processing (30 frames per second). Without proper parallelization, it would be very difficult for Hough Transform to achieve real-time or interactive performance in the future.

Because of its significant importance and computation intensive nature, Hough Transform has been attracted many studies for faster execution. Xu et al. proposed the randomized Hough Transform [2]. Roth et al. presented the genetic Hough Transform algorithm [3]. Jiang et al. used simulated annealing and Tabu search to accelerate Hough Transform [4]. These three approaches mainly focus on the algorithm enhancement.

Another way to increase the performance is to run Hough Transform effectively on multi-core processors. The performance of microprocessors improves significantly because of the introduction of multiple cores, e.g., the latest Intel® Core™ 2 Quad processor. Moving forward, we expect a trend of increasing the number of processing cores in a single microprocessor [5]. So, we can increase the performance of Hough Transform by effectively parallelizing it. In [6] and [7], Jin et al. and Chuang et al. parallelize Hough Transform on multi-processor systems. They split the image into sub-images, and distribute them across processors. These two approaches communicate the Hough voting table through the shared memory. Similar to our parallelization on the old Hough Transform, this kind of parallel implementation has synchronization penalty and poor memory performance, as we will describe later.

Our work differs from prior work: (1) We use a number of schemes to parallelize the Hough Transform. Our parallelization goal is to compare different thread-level parallelisms and achieve the best performance moving into the future. (2) We propose a new Hough Transform algorithm, which is lock-free and cache-friendly. By extracting the fine-grain data-level parallelism, we obtain significant performance gain on an 8-core machine.

2. HOUGH TRANSFORM

For Hough Transform, the popular usage is line detection (Hough linear transform). Figure 1 shows the classical Hough linear transform from OpenCV [8] (for ease of comparison, we term this version as *old Hough Transform*). It scans the input binary image, and maps each point in the original x-y image space to a sinusoidal curve in the ρ-θ Hough space by the formula $\rho = x\cos\theta + y\sin\theta$. The ρ represents the distance between the line and the origin, and θ is the angle of the vector from the origin to the closet point (see coordinates in Figure 2). By using this equation and

```
For( j = 0; j < height; j++ )
   for( i = 0; i < width; i++ )
   {
      if( BinImage[j][i] != 0 )
          for( theta = 0; theta < numangle; theta++ )
          {
              rho = i * Cos[theta] + j * Sin[theta];
              atomically_increment(accum[theta][rho]);
          }
   }
```

Figure 1 The Hough Transform in OpenCV

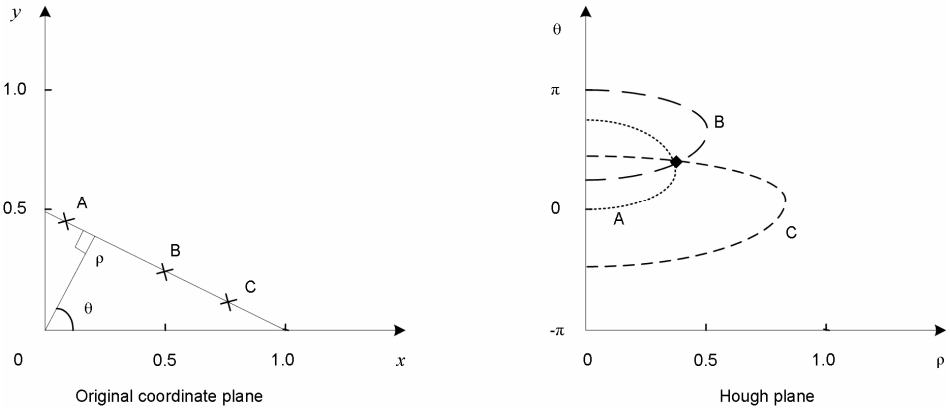

Figure 2 Hough line detection (one point in image space is mapped to one curve in Hough space. The number of points in the mapped curve is determined by the *numangle* in Figure 1)

parameterization, each point and line in the image plane has one unique sinusoidal curve and one unique couple (ρ,θ) in the Hough plane, respectively. As each line goes through many points in image space, the mapped sinusoidal curves of these points will intersect at a point in Hough plane, which gives the distance and angle for the line in image space.

One challenge in Hough Transform is the data dependence, which limits the concurrency exploration among pixels. In Figure 2, the three points (A, B, and C) in the solid line are mapped to three different curves in Hough plane, and the intersected point by these curves in Hough space corresponds to the line that bisects these three points in image space. The philosophy behind Hough Transform implies that, for the line points, their mapped sinusoidal curves will definitely go through one same point in Hough space. The voting at this point indicates there is read-write dependence between iterations (one iteration represents one pixel point). Therefore, if we extract fine-grain data-level parallelism (i.e., split the outside loop), we must protect the parallel accesses to Hough voting table (i.e., accum[theta][rho]) to ensure there is only one thread modifying this table at any time.

Another challenge of Hough Transform is its poor data locality and unfriendly memory access behavior. As shown in Figure 1, the Hough Transform shows good spatial data locality on the input binary image as it reads the data structure in row order. However, it exhibits very poor data locality on the Hough voting table because the points of mapped curve in the Hough space are not continuous. Therefore, it is very difficult for the hardware to accurately predict the memory access pattern. The poor data locality and non-regular access pattern results in high cache miss rates. (See Section 4.2 for memory performance analysis.)

3. PARALLELIZATION

3.1. Coarse-Grain Parallelized Old Hough Transform

One straightforward parallelization of Hough Transform is to process different frames in parallel, that is, same arithmetic Hough linear transform is applied to each frame independently to detect lines. The advantage of such coarse-grain parallelization is to avoid the locking since there is no dependence between frames. However, processing a large number of frames simultaneously will require a huge amount of data (called, working set). When the size of the working set is larger than the size of the microprocessor cache, the parallel performance is often limited.

3.2. Fine-Grain Parallelized Old Hough Transform

To reduce the working set, we can multi-thread the old Hough Transform at a finer granularity. We process only a frame at a time, but we split the outside loop. As the old Hough Transform has potential loop-carried read-write dependence across iterations, we use locks (an atomic primitive) to enforce exclusive access to Hough voting table. However, the atomic primitive will incur expensive synchronization overhead. The parallel speedup will be limited on large-scale multi-core processors.

3.3. Fine-Grain Parallelized New Hough Transform

```
int  EdgePixelSize = 0;

for( j = 0; j < height; j++ )
   for( i = 0; i < width; i++ )
   {
     if( BinImage[j][i] != 0 )
     {
            EdgePixels[EdgePixelSize++] = j;
            EdgePixels[EdgePixelSize++] = i;
     }
   }

for(theta = 0; theta < numangle; theta++) //Theta // Hough voting procedure
{
     for(m = 0; m < EdgePixelSize; m+=2) // loop edge pixels
     {
            j = EdgePixels[m];
            i = EdgePixels[m+1];

            rho =  i *Cos[theta] + j*Sin[theta];
            accum[theta][rho]++;
     }
}
```

Figure 3 New Hough Transform algorithm

We propose a new Hough Transform algorithm, as shown in Figure 3. In the new Hough Transform, we first push the edge pixels (non-zero elements) into a buffer, and then perform voting on the Hough voting table. With respect to the old Hough Transform in Figure 1, the new algorithm accesses the Hough voting table in the order of ρ inside and θ outside. With respect to the old Hough Transform in Figure 1, this new algorithm has the following advantages:

First, this new Hough Transform algorithm has good temporal and spatial data localities. Inside loop m, all edge pixels are mapped into the same θ of the Hough voting table. When two edge pixels have the same j value and similar i values, they will be mapped into the same ρ or similar ρ's. In this case, accum[theta][rho] has good temporal/spatial localities. (See Section 4.2 for memory performance analysis.)

Second, besides having good data localities, this new Hough Transform is also lock-free since each outside iteration (θ) touches one row of the Hough voting table, and different outside iterations access different rows. Thus, the Hough voting table is not read-write shared among threads when extracting the fine-grain data-level parallelism (i.e., split the outside loop).

In summary, this new algorithm is cache-friendly and lock-free. Compared to the old Hough Transform, although the new algorithm executes 14% more instructions, the cache-friendly behavior makes it 25% faster than the old Hough Transform on a single processor.

4. SCALING PERFORMANCE ON 8 CORES

Figure 4 shows the parallel performance of 3 parallelized Hough Transform on an 8-core machine. The multi-core platform is a dual-socket quad-core machine, with two Intel® Core™ 2 Quad processors running at 2.33GHz. Although performing well, the old Hough Transform doesn't scale linearly on the 8-core system. On the other hand, our new Hough Transform algorithm scales perfectly. On the 8-core system, our new Hough Transform is 25% faster than the other two schemes for the single-threaded case.

To fully understand the speedup-liming factors on the 8-core system, we characterize the parallel performance (1) from the perspective of the high-level parallelization overhead, e.g., synchronization penalties, load imbalance, and sequential regions, and (2) from the detailed memory behavior, e.g., bus bandwidth.

4.1. Parallel Performance Metrics

In general, our parallelized Hough Transform exposes good parallel performance metrics. Figure 5 depicts the parallel profiling metrics for these three different parallel implementations. The higher the parallel region, the better speedup can be achieved on highly threaded architectures.

Surprisingly, the profiling information suggests we should have very good scalabilities. If we assume the parallel region can scale perfectly, two parallel old Hough Transform implementations should achieve the theoretical speedups of 7.95 and 7.2, respectively. These numbers are higher than the results shown in Figure 4. Thus, we believe the scalability of old Hough Transform is limited by some other factors that are discussed in Section 4.2.

4.2 Memory Subsystem Behaviors

Besides the general parallel performance metrics, the memory subsystem also plays an important role in the scalability—a workload cannot scale well when the instantaneous bandwidth usage is higher than the system's capability.

We perform interval sampling of the memory subsystem behavior over time. Figure 6 shows the bandwidth usage over time for the old Hough Transform and single-threaded new Hough Transform on a single core. As we mentioned in

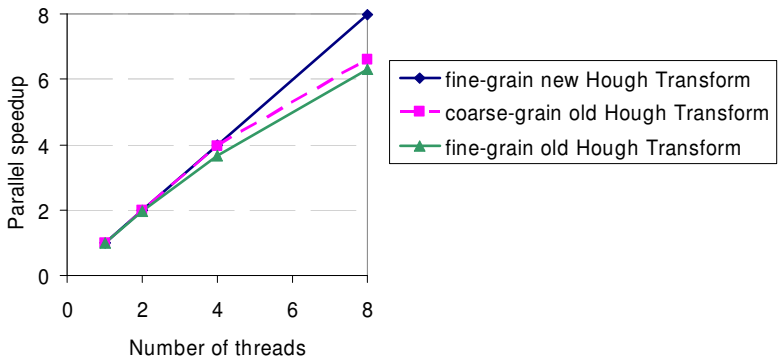

Figure 4 Scalability of parallel Hough Transform on 8-core machine

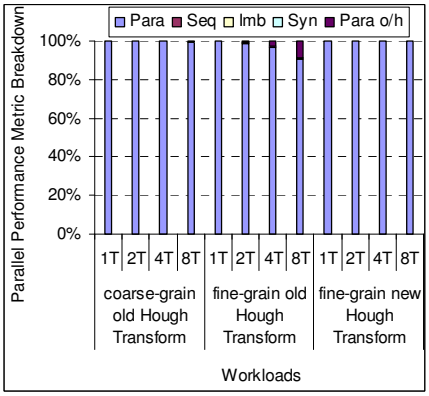

Figure 5 Execution time breakdown

Section 2, the old Hough Transform is not cache-friendly. Furthermore, there are some bursty memory access behaviors—the instantaneous bandwidth usage is much higher than the average bandwidth. When the bandwidth demand is higher than the system's capability, the memory performance becomes the bottleneck of scalability.

In contrast, as we mentioned in Section 3.3, the new Hough Transform is cache-friendly, and has low bandwidth demand. The instantaneous memory bandwidth consumption is always lower that the system's capacity. Therefore, the scaling performance of new Hough Transform is not affected by the memory subsystem on the 8-core machine.

5. CONCLUSION

This paper demonstrated how to parallelize an image processing algorithm on multi-core processors. Realizing the performance potential on multi-core processors requires the applications to expose a significant amount of thread-level parallelism. The scalability of the old Hough Transform is limited by parallel overheads and memory requirements. Our study suggests it is important to choose appropriate sequential algorithm and parallelize it with fine-grain parallelism to minimize memory requirements. Our new Hough Transform (1) has better memory performance and (2) does not use locks. Although the new Hough Transform executes 14% more instructions than the old Hough Transform, the good memory subsystem performance makes the new algorithm 25% faster than the old algorithm even when it is running on a single core. The new Hough Transform algorithm will scale also well on future platform with large number of cores, such as 64 cores [9]. Although this paper only studies one program, the methodology in analyzing and identifying the non-scaling performance is applicable to other workloads targeting at multi-core processors.

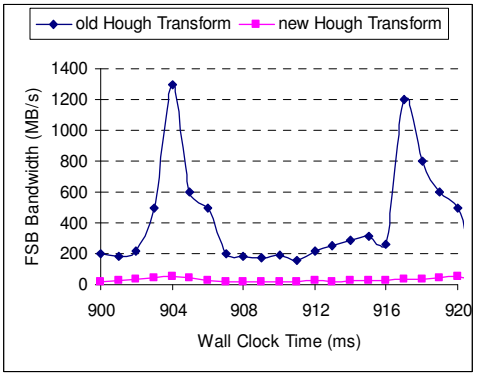

Figure 6 Bandwidth usage over time

6. REFERENCES

[1] P Hough. Machine Analysis of Bubble Chamber Pictures. In International Conference on High Energy Accelerators and Instrumentation. CERN, 1959.

[2] L Xu, et al. Randomized Hough Transform (RHT): Basic Mechanism, Algorithms, and Computational Complexities. CVGIP: Image Understanding, 57(2):131–154, 1993.

[3] G Roth, et al., "Geometric Primitive Extraction Using a Genetic Algorithm," IEEE Trans. on PAMI, 9(16):901–905, 1994.

[4] T Jiang, et al., "Geometric Primitive Extraction Using Tabu Search," in Int'l Conf. on Pattern Recognition, vol. 2, pp. 266–279, 1996.

[5] S Vangal, et al., "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," in Proc. of the 2007 IEEE Intl. Solid-State Circuits Conf., 2007.

[6] L Jin, et al., "Parallel Solution of Hough Transform and Convolution Problems—a Novel Multimodal Approach," ACM Symp. on Applied computing, 1992.

[7] H Chuang, et al., "An Efficient Hough Transform on SIMD Hypercube," in Int'l Conf. on Parallel and Distributed Systems, 1994.

[8] Intel Corp. Intel® Integrated Performance Primitives. http://www.intel.com/software/products/ipp

[9] W Li, et al., "Parallelization, Performance Analysis, and Algorithm Consideration of Hough Transform on Chip Multiprocessors," in Workshop on Design, Architecture and Simulation of Chip Multi-Processors, Dec. 2007.