Efficient GPU Implementation of the Integral Histogram

Mahdieh Poostchi, Kannappan Palaniappan, Filiz Bunyak,
Michela Becchi, and Guna Seetharaman

Dept. of Computer Science, University of Missouri-Columbia, Columbia, Missouri
Air Force Research Laboratory, Rome, NY 13441, USA

Abstract. The integral histogram for images is an efficient preprocessing method for speeding up diverse computer vision algorithms including object detection, appearance-based tracking, recognition and segmentation. Our proposed Graphics Processing Unit (GPU) implementation uses parallel prefix sums on row and column histograms in a cross-weave scan with high GPU utilization and communication-aware data transfer between CPU and GPU memories. Two different data structures and communication models were evaluated. A 3-D array to store binned histograms for each pixel and an equivalent linearized 1-D array, each with distinctive data movement patterns. Using the 3-D array with many kernel invocations and low workload per kernel was inefficient, highlighting the necessity for careful mapping of sequential algorithms onto the GPU. The reorganized 1-D array with a single data transfer to the GPU with high GPU utilization, was 60 times faster than the CPU version for a $1K \times 1K$ image reaching 49 fr/sec and 21 times faster for 512×512 images reaching 194 fr/sec. The integral histogram module is applied as part of the likelihood of features tracking (LOFT) system for video object tracking using fusion of multiple cues.

1 Introduction

The integral histogram extends the integral image method for scalar sums to vector (*i.e.* histogram) sums and enables multiscale histogram-based search and analysis in constant time after a linear time preprocessing stage [1, 2]. The integral histogram is a popular method to speed up computer vision tasks, especially sliding window based methods for object detection, tracking, recognition and segmentation [1–12]. Histogram-based features are widely used in image analysis and computer vision due to their simplicity and robustness. Histogram is a discretized probability distribution where each bin represents the frequency or probability of observing a specific range of feature values for a given descriptor such as intensity, color, edginess, texture, shape, motion, etc. Robustness to geometric deformations makes histogram-based feature representation appealing for many applications. One major drawback of sliding window histograms is their high computational cost, limiting their use for large scale applications such as content-based image retrieval with databases consisting of millions of images or

J.-I. Park and J. Kim (Eds.): ACCV 2012 Workshops, Part I, LNCS 7728, pp. 266–278, 2013.

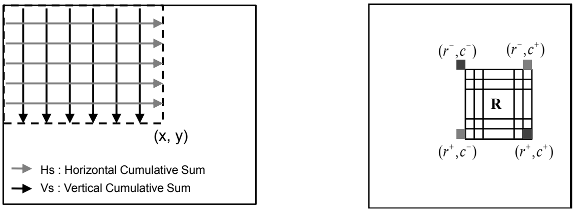

Fig. 1. (a) Computation of the histogram up to location (x, y) using a cross-weave horizontal and vertical scan on the image. (b) Computation of the histogram for an arbitrary rectangular region R (origin is the upper-left corner with y-axis horizontal.)

full motion video archives with billions of frames. Any improvement that leads to a speed-up in integral histogram calculation is imperative especially due to fast trend towards extreme-scale and high-throughput data analysis. Mapping image analysis and computer vision algorithms onto many-core and multicore architectures has benefits ranging from faster processing, deeper search, greater scalability, and better performance especially for recognition, retrieval and reconstruction tasks [13–19]

As far as we know this is the first detailed description and performance characterization of a parallel implementation of the integral histogram for GPU architectures. Previously, a parallelization of the integral histogram for the eight-core IBM Cell/B.E. processor was described in [20] and the scalar *integral image* computation was parallelized for the GPU [21]. Although both the integral image and integral histogram follow the same strategy, the integral histogram uses high memory since the histogram needs to be maintained for every pixel and leads to a 3D array data structure that is difficult to manage on small 48KB on-chip shared memory per stream multiprocessor available on GPUs. This paper presents two parallel implementations of the integral histogram computation, that we have developed, for many-core GPU architectures, using the CUDA programming model [13, 22, 23]. Both methods, GPU Integral Histogram using Multiple Scan-Transpose-Scan (GIH-Multi-STS) and GPU Integral Histogram using Single Scan-Transpose-Scan (GIH-Single-STS) use CUDA SDKs for parallel cumulative sums of rows and columns (prescan) based on a cross-weave scanning mode, 2-D or 3-D transpose kernels and communication-aware data management.

The contributions of this work are designing the best data structure and its layout in GPU memory, finding the kernel configuration that maximizes the resource utilization of the GPUs and minimizes the data movement. Section 2 presents a short review of the integral histogram algorithm. Section 3 explores our parallel integral histogram implementations on GPUs, followed by experimental results including application to tracking and conclusions.

2 Integral Histogram Description

The integral histogram is a recursive propagation preprocessing method used to compute local histograms over arbitrary rectangular regions in constant time [1].

Algorithm 1. Sequential Integral Histogram

```
Input  : Image $\mathbf{I}$ of size $h \times w$
Output : Integral histogram tensor $\mathbf{H}$ of size $h \times w \times b$
Initial H:
$\mathbf{H} \leftarrow 0$
for z=1:b do
    for x=1:w do
        for y=1:h do
            $H(x, y, z) \leftarrow H(x-1, y, z) + H(x, y-1, z)$
            $-H(x-1, y-1, z) + Q(I(x, y), z)$
        end for
    end for
end for
```

The efficiency of the integral histogram approach enables real-time histogram-based exhaustive search in vision tasks such as object recognition and tracking. The integral histogram is extensible to higher dimensions and different bin structures. The integral histogram at position (x, y) in the image holds the histogram for all the pixels contained in the rectangular region defined by the top-left corner of the image and the point (x, y) as shown in Figure 1. The integral histogram for the region defined by the spatial coordinate (x, y) and bin variable b is defined as:

$$H(x, y, b) = \sum_{r=0}^{x} \sum_{c=0}^{y} Q(I(r, c), b) \tag{1}$$

where $Q(I(r, c), b)$ is the binning function that evaluates to 1 if $I(r, c) \in$ bin b, and evaluates to zero otherwise. Sequential computation of integral histograms is described in Algorithm 1. Given the image integral histogram $\mathbf{H}$, computation of the histogram for any test region R delimited by points $\{(r^-, c^-), (r^-, c^+), (r^+, c^+), (r^+, c^-)\}$ reduces to the combination of four integral histograms:

$$h(R, b) = \ H(r^+, c^+, b) - H(r^-, c^+, b) - H(r^+, c^-, b) + H(r^-, c^-, b) \tag{2}$$

Figure 1 illustrates the notation and accumulation of integral histograms using vertical and horizontal cumulative sums (prescan), which is used to compute regional histograms.

3 Parallelization Using Parallel Prefix-Sum (Exclusive Scan)

One basic pattern in parallel computing is the use of independent concurrently executing tasks. The recursive sequential Algorithm 1 is a poor approach to parallelize since row $(r+1)$ cannot be executed until row r is completed, with only intra-row parallelization. The cross-weave scan mode (Fig. 1), enables cumulative sum tasks over rows (or columns) to be scheduled and executed independently allowing for inter-row and column parallelization. The GPU Integral Histogram using Multiple Scan-Transpose-Scan (GIH-Multi-STS) is shown in Algorithm 2.

Algorithm 2. GIH-Multi-STS: GPU Integral Histogram using Multiple Scan-Transpose-Scan

```
Input : Image $\mathbf{I}$ of size $h \times w$
Output : Integral histogram tensor $\mathbf{IH}$ of size $b \times h \times w$
Initialize IH
$\mathbf{IH} \leftarrow 0$
$\mathbf{IH}(\mathbf{I}(\mathbf{w},\mathbf{h}),\mathbf{w},\mathbf{h}) \leftarrow 1$
for z=1 to b do
    for x=1 to h do
        //horizontal cumulative sums (prescan, size of rows)
        $IH(x,y,z) \leftarrow IH(x,y,z) + IH(x,y-1,z)$
    end for
end for
for z=1 to b do
    //transpose the bin-specific integral histogram
    $IH^T(z) \leftarrow$ 2-D Transpose($IH(z)$)
end for
for z=1 to b do
    for y=1 to w do
        //vertical cumulative sums (prescan, size of columns)
        $IH(x,y,z) \leftarrow IH^T(y,x,z) + IH^T(y,x-1,z)$
    end for
end for
```

This approach combines cross-weave scan mode with an efficient parallel prefix sum operation and an efficient 2-D transpose kernel. The SDK implementation of *all-prefix-sums* operation using the CUDA programming model is described by Harris, *et al.* [24]. We apply prefix-sums to the rows of the histogram bins (horizontal cumulative sums or prescan), then transpose the array and reapply the prescan to the rows to obtain the integral histograms at each pixel.

3.1 Parallel Prefix Sum Operation on the GPU

The core of the parallel integral histogram algorithm for GPUs is the parallel prefix sum algorithm [24]. The *all-prefix-sums* operation (also refered as a scan) applied to an array generates a new array where each element k is the sum of all values preceding k in the scan order. Given an array $[a_0, a_1, ..., a_{n-1}]$ the prefix-sum operation returns,

$$[0, a_0, (a_0 + a_1), ..., (a_0 + a_1 + ... + a_{n-2})] \tag{3}$$

The parallel prefix sum operation on the GPU consists of two phases: an *up-sweep* (or reduce) phase and a *down-sweep* phase (see Fig. 2). *Up-sweep* phase builds a balanced binary tree on the input data and performs one addition per node. Scanning is done from the leaves to the root. In the *down-sweep* phase the tree is traversed from root to the leaves and partial sums from the up-sweep phase are aggregated to obtain the final scanned (prefix summed) array. Prescan requires only $O(n)$ operations: $2*(n-1)$ additions and $(n-1)$ swaps.

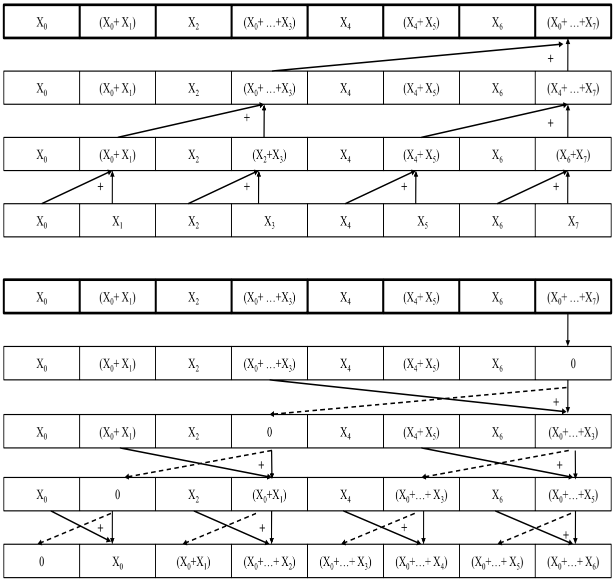

Fig. 2. Parallel prefix sum operation, commonly known as exclusive scan or prescan [24]. Top: Up-sweep or reduce phase applied to an 8-element array. Bot: Down sweep phase.

The GPU-based prefix sum (prescan) operation moves data from CPU memory to off-chip global GPU memory then exploits the on-chip shared memory for each row operation [24].

3.2 GPU-Based Transpose Kernel

The integral histogram computation requires two prescans over the data. First, a horizontal prescan that computes cumulative sums over rows of the data, followed by a second vertical prescan that computes cumulative sums over the columns of the first scan output. Taking the transpose of the horizontally prescanned image histogram, enables us to reapply the same (horizontal) prescan algorithm effectively on the columns of the data. We used the optimized transpose kernel described in [25] that uses zero bank conflict shared memory and guaranties that global reads and writes are coalesced. Figure 3 shows the data flow in the transpose kernel. A tile of size BLOCK_DIM ∗ BLOCK_DIM is written to the GPU shared memory into an array of size BLOCK_DIM ∗ (BLOCK_DIM + 1). This pads each row of the 2-D block in shared memory so that bank conflicts do not occur when threads address the array column-wise. Each transposed tile is written back to the GPU global memory to construct the full histogram transpose. The SDK 2-D transpose kernel needs to be launched from the host b times in order to transpose the integral histogram tensor. In order to allow a single transpose operation, we transform the existing 2-D transpose kernel into a 3-D transpose kernel by using the bin offset in the indexing. The 3-D transpose kernel

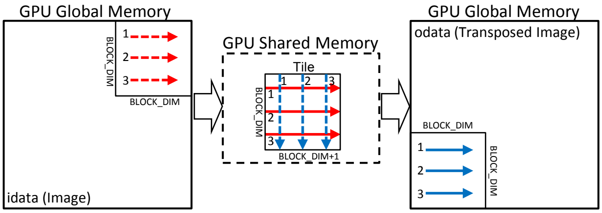

Fig. 3. Data flow between GPU global memory and shared memory while computing the coalesced transpose kernel; stage 1 in red, stage 2 blue, reads are dashed lines, writes are solid lines.

is launched using a 3-D grid of dimension $(b, w/\text{BLOCK\_DIM}, h/\text{BLOCK\_DIM})$, where BLOCK_DIM is the maximum number of banks in shared memory (32 for all graphics card used).

3.3 Data Structure and Implementation Strategy

An image with dimensions $h \times w$ produces an integral histogram tensor of dimensions $h \times w \times b$, where b is the number of bins in the histogram. This tensor can be represented as a 3-D array which in turn can be mapped to an 1-D row major ordered array for efficient access as shown in Figure 4. Both implementations, GIH-Multi-STS and the improved GPU Integral Histogram using Single Scan-Transpose-Scan (GIH-Single-STS), start by prescanning each row. Since the maximum number of threads per block is 1024 and each thread processes two elements, each row can be divided into segments up to 2048 pixels. If the size of row is smaller than 2048 then the size of the thread block will be reduced to the $w/2$. The GIH-Multi-STS implementation uses the 3-D data structure. Exclusive prefix sum (prescan) kernel (see Section 3.1) is applied to the data one row at a time. This approach suffers from many kernel invocations in the horizontal/vertical scan and 2-D transpose phases, from little work per kernel and eventually GPU under-utilization (Algorithm 2). To reduce the total number of kernels invocations from $(w + h)b + b$ to only 3 invocations, the GIH-Single-STS implementation uses a 1-D row ordered format array and launches the prescan kernel *once* using a 1-D grid of size $(b * h * w)/(2 * \text{Num\_Threads})$. Padding is applied to shared memory addresses to avoid bank conflicts by adding an offset of 32 to each shared memory index. After prescanning each row (horizontal scan),

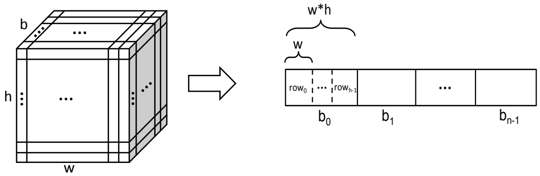

Fig. 4. Integral histogram tensor represented as 3-D array data structure (left), and equivalent 1-D array mapping (right)

Algorithm 3. GIH-Single-STS: GPU Integral Histogram using Single Scan-Transpose-Scan

```
Input : Image $\mathbf{I}$ of size $h \times w$, number of bins $b$
Output : Integral histogram tensor $\mathbf{IH}$ of size $b \times h \times w$
Initialize IH
        $\mathbf{IH} \leftarrow 0$
        $\mathbf{IH(I(w,h),w,h)} \leftarrow 1$
for all $b \times h$ blocks in parallel do
    //horizontal cumulative sums
    Prescan($IH$)
end for
//transpose the histogram tensor
        $IH^T \leftarrow$ 3D_Transpose($IH$)
for all $b \times w$ blocks in parallel do
    //vertical cumulative sums
    Prescan($IH^T$)
end for
```

the prescanned array is transposed to compute (column) cumulative sums in the second pass using a 3-D transpose kernel (Algorithm 3). We implemented and evaluated two parallel GPU integral histogram computation approaches: parallel GIH-Multi-STS, and parallel GIH-Single-STS and compared them to a sequential CPU-only implementation. Our experiments were conducted on a 2.0 GHz Quad Core Intel CPU (Core i7-2630QM) and two GPU cards: an NVIDIA Tesla C2070 and an NVIDIA GeForce GTX 460. The former is equipped with fourteen 32-core SMs and has about 5GB of global memory, 48KB shared memory with compute capability 2.0. The latter consists of seven 48-core SM and is equipped with 1GB global memory, 48KB shared memory with compute capability 2.1.

The parallel GIH-Multi-STS implementation exploits the work efficient prescan operation to calculate for each bin the cumulative sums of rows, one row at a time. Therefore, the scan kernel is launched $b \times h$ times for horizontal scan and $b \times w$ times for vertical scan. The efficient 2-D transpose kernel is launched b times to transpose the integral histogram tensor after horizontal scan. The GIH-Multi-STS is based on many kernel invocations, each of them performing a small amount of work and therefore greatly under-utilizing the many-cores on the GPU. In addition, the all-prefix-sum kernel works very well only on very large array consisting of millions of elements. Therefore, we proposed the GIH-Single-STS to increase the amount of work performed by each kernel invocation and reduce the number of scan kernel invocations by a factor of $(h + w)b$. This can be easily achieved by modifying the kernel configuration without rewriting the kernel code (array indices are derived from block and thread indices). Since the maximum number of threads per block is 1024 and each thread processes two elements, each row can be divided into segments up to 2048 pixels. If the size of row is smaller than 2048 then the size of the thread block will be reduced to the $w/2$ for horizontal scan and $h/2$ for vertical scan as well. Therefore, the number of blocks for horizontal scan will be $((b \times h \times w)/(2 \times \mathit{threadblock}))$.

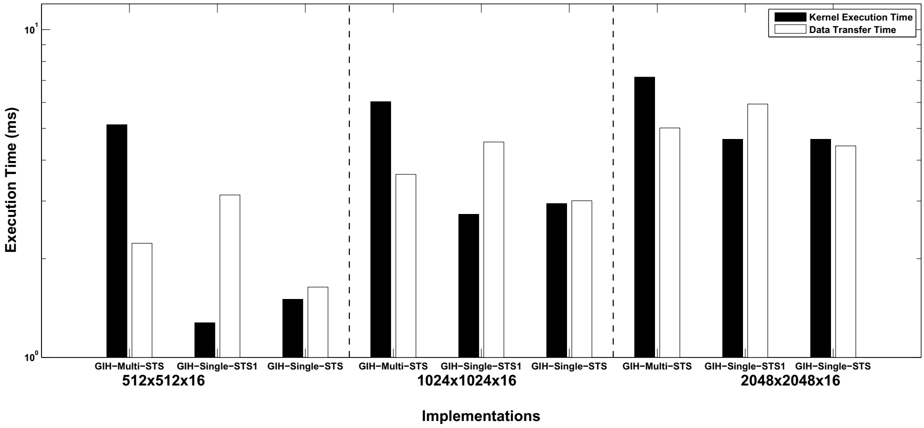

Fig. 5. Kernel execution time versus data transfer time for different image sizes

GIH-Single-STS also benefits from the modified 2-D transpose kernel which performs a single 3-D transpose operation by using the bin offset in the indexing. GIH-Single-STS is divided into three phases: a single horizontal scan, a 3-D transpose, and a vertical scan.

The initial implementation of GIH-Single-STS had several unnecessary data transfer between host and device after each phase. In the first implementation, the integral histogram tensor was being transferred to the GPU before invoking the kernel and then sent back to the CPU before launching the next kernel; these extra data transfers lead to reduced performance (referred to as GIH-Single-STS1). However, the GPU is specialized for compute-intensive, highly parallel computation and the overhead of communication between host and device cannot be hidden or double-buffered by non data-intensive kernels. In the improved GIH-Single-STS implementation, the integral histogram computations start after transferring the image to the GPU, complete the calculation of the integral histogram on the GPU then transfer the final integral histogram tensor back to the CPU, removing the extra communication overhead. In addition, the number of threads is automatically determined based on the image size to ensure maximum occupancy per kernel.

Figure 5 shows the kernel execution time versus data transfer time for GIH-Multi-STS, GIH-Single-STS1 (implementation with extra data transfers) and GIH-Single-STS for different image sizes. We see that the GIH-Multi-STS is compute bound (that is, the kernel execution time is larger than the CPU to GPU data transfer time), this method under utilizes the GPU, whereas the GIH-Single-STS1 is data-transfer-bound. The results show that the data transfer time for GIH-Single-STS1 is on average five times worse than GIH-Single-STS. The final GIH-Single-STS implementation shows a balance between data transfer and kernel execution time (Fig. 5).

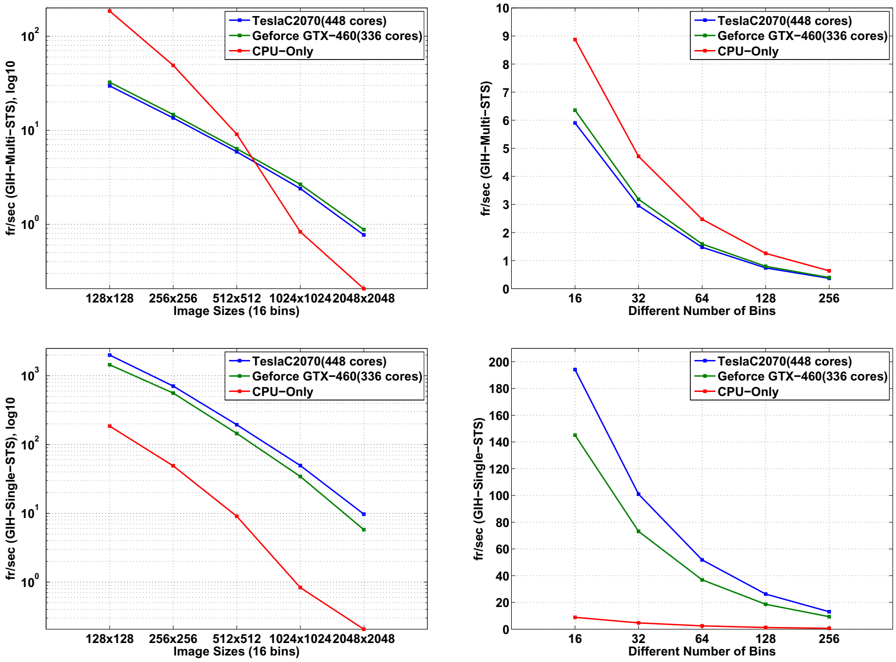

Fig. 6. Frame rate of GIH-Multi-STS, GIH-Single-STS and CPU-only integral histogram implementations: (UL) GIH-Multi-STS frame rate for different image sizes, (UR) GIH-Multi-STS frame rate for different number of bins, (LL) GIH-Single-STS frame rate for different image sizes, (LR) GIH-Single-STS frame rate for different number of bins for 512x512 image size.

Figure 6 summarizes the frame rate performance of the two GPU implementations compared to the sequential CPU-only implementation. The frame rate is defined as the maximum number of images processed per second. Since we use double buffering, the frame rate equals $1/(kernel\_execution\_time)$ for compute-bound cases, or $1/(data\_transfer\_time)$ for data-transfer-bound cases. Considering double buffering timing, our GIH-Single-STS achieves 194 fr/sec to compute 16-bin integral histograms for a 512×512 image and 94 fr/sec for $1K \times 1K$ image using the NVIDA Tesla C2070 GPU.

Figure 7 reports the speedup of our GPU implementations of the integral histogram compared to a sequential CPU implementation. The speedup takes into consideration the overlapping of computation and communication used by double buffering. The speedup of the improved GIH-Single-STS for a 16-bin integral histogram for a $1K \times 1K$ image is 60 times on an NVIDIA Tesla C2070 GPU and varies between 15 times to 25 times for a 512×512 image depending on the number of bins and the type of GPU.

Figures 8 shows feature maps for the target and search window with corresponding likelihood maps produced by the integral histogram-based likelihood

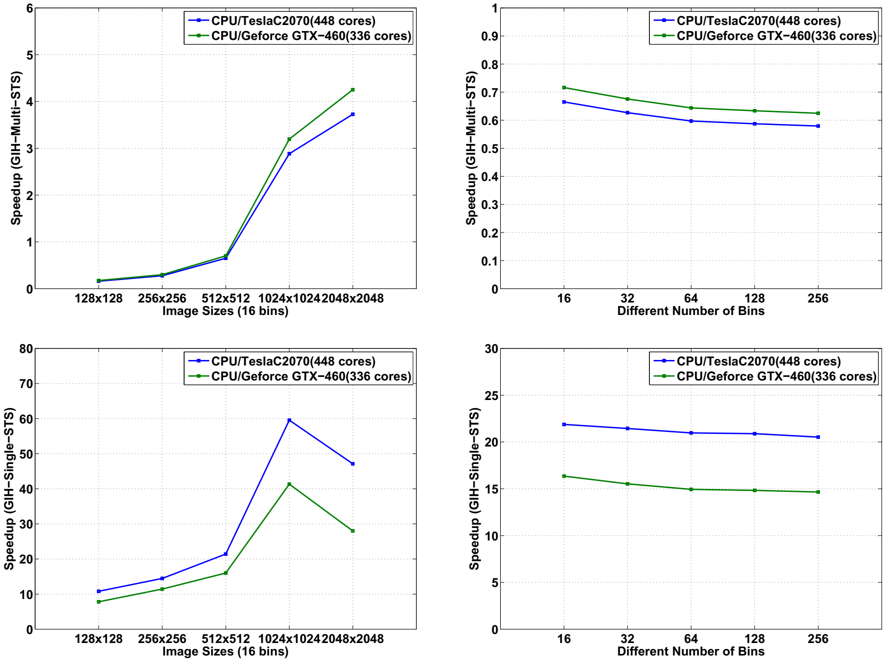

Fig. 7. Speedup of the two GPU designs over CPU on two NVIDA graphic cards: (UL) Speedup of GIH-Multi-STS (with respect to CPU-only) with different image sizes, (UR) Speedup of GIH-Multi-STS with varying number of bins, (LL) Speedup of GIH-Single-STS for different image sizes, (LR) Speedup of GIH-Single-STS with varying number of bins for 512x512 image size.

estimation approach. Figure 9 shows sample tracking results and fused likelihood maps for sample frames from an aerial wide area image sequence.

4 Conclusions

We have presented two parallel implementations of the integral histogram using the cross-weave scanning approach for GPU architectures, utilizing the CUDA programming model. The poor performance of the GIH-Multi-STS (*prescan*) implementation which was slower than the sequential version and the first implementation of GIH-Single-STS, clearly demonstrates that in parallelizing sequential image analysis algorithms on the GPU, data structures, GPU utilization and communication patterns need careful consideration. The GIH-Single-STS (*efficient communication*) implementation reduced the severe communication overhead bottleneck, by transferring an image size 1-D array instead of an integral histogram 3-D array and increased the GPU utilization. The GIH-Single-STS exploits an efficient prescan and 3-D transpose operation with maximum occupancy per kernel. It achieved frame rate of 185 for standard images 640 × 480 for 16 bins integral histogram computations which outperforms results presented

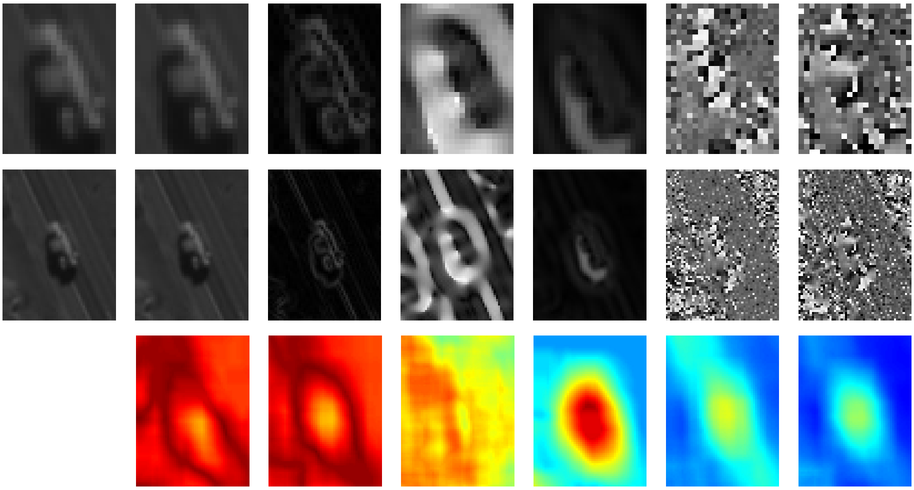

Fig. 8. Top row shows the car template and associated raw target features for intensity, gradient magnitude, Hessian shape index, normalized curvature index, Hessian eigenvector orientations, and oriented gradient angles. Row 2 shows the predicted search window and associated raw features. Row 3 shows the corresponding likelihood maps combining target template with the associated search window features using integral histogram.

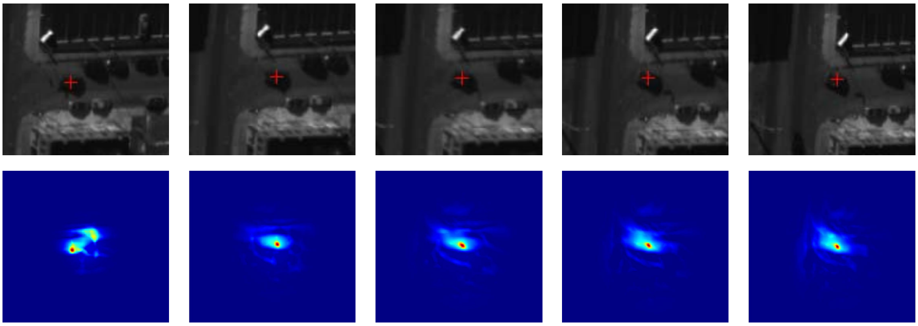

Fig. 9. LOFT tracking results are shown for the first five frames for car C4_1_0 from CLIF aerial wide-area motion imagery [26]. Top row shows the tracked car locations and the bottom row shows the fused likelihood maps used by LOFT [8] to determine the best target location in each corresponding frame.

for 8 SPEs (120 fr/sec for cross-weave scan and 172 for wavefront scan mode) in [20]. However, in most cases our performance is data-transfer-bound. One approach to further improve the time and memory efficiency of the GPU-based integral histogram method is to develop our custom parallel scan kernel for the horizontal and vertical cumulative sum computations without transpose phase for each tile of integral histogram tensor.

Acknowledgement. This research was partially supported by U.S. Air Force Research Laboratory (AFRL) under agreement AFRL FA8750-11-1-0073. Approved for public release (case 88ABW-2012-1016). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of AFRL or the U.S. Govt. The U.S. Government is authorized to reproduce and distribute reprints for Govt. purposes notwithstanding any copyright notation thereon.

References

1. Porikli, F.: Integral histogram: A fast way to extract histograms in cartesian spaces. In: IEEE CVPR, vol. (1), pp. 829–836 (2005)
2. Sizintsev, M., Derpanis, K.G., Hogue, A.: Histogram-based search: A comparative study. In: IEEE CVPR, pp. 1–8 (2008)
3. Viola, P., Jones, M.J.: Robust real-time face detectin. Int. J. Computer Vision 57, 137–154 (2004)
4. Wei, Y., Tao, L.: Efficient histogram-based sliding window. In: IEEE CVPR, pp. 3003–3010 (2010)
5. Zhu, Q., Yeh, M.C., Cheng, K.T., Avidan, S.: Fast human detection using a cascade of histograms of oriented gradients. In: IEEE CVPR, vol. (2), pp. 1491–1498 (2006)
6. Adam, A., Rivlin, E., Shimshoni, I.: Robust fragments-based tracking using the integral histogram. In: IEEE CVPR, pp. 798–805 (2006)
7. Palaniappan, K., et al.: Efficient Feature extraction and likelihood fusion for vehicle tracking in low frame rate airborne video. In: 13th Conf. Information Fusion, pp. 1–8 (2010)
8. Pelapur, R., Candemir, S., Bunyak, F., Poostchi, M., Seetharaman, G., Palaniappan, K.: Persistent target tracking using likelihood fusion in wide-area and full motion video sequences. In: 15th Int. Conf. Information Fusion, pp. 2420–2427 (2012)
9. Erdem, E., Dubuisson, S., Bloch, I.: Fragments Based Tracking with Adaptive Cue Integration. Computer Vision and Image Understanding (7), 827–841 (2012)
10. Mosig, A., Jaeger, S., Chaofeng, W., Ersoy, I., Nath, S.K., Palaniappan, K., Chen, S.S.: Tracking cells in live cell imaging videos using topological alignments. Algorithms in Molecular Biology (4), 10 (2009)
11. Kolekar, M.H., Palaniappan, K., Sengupta, S., Seetharaman, G.: Semantic concept mining based on hierarchical event detection for soccer video indexing. Special Issue on Multimodal Information Retrieval (4), 298–312 (2009)
12. Palaniappan, K., Rao, R., Seetharaman, G.: Wide-area persistent airborne video: Architecture and challenges. In: Distributed Video Sensor Networks: Research Challenges and Future Directions, pp. 349–371 (2011)
13. Park, I.K., et al.: Design and performance evaluation of image processing algorithms on GPUs. IEEE Parallel and Distributed Systems 22(1), 91–104 (2011)
14. Grauer-Gray, S., Kambhamettu, C., Palaniappan, K.: GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In: 5th IAPR Workshop on Pattern Recognition in Remote Sensing (ICPR), pp. 1–4 (2008)
15. Palaniappan, K., et al.: Parallel flux tensor analysis for efficient moving object detection. In: Int. Conf. Information Fusion, pp. 1–8 (2011)
16. Palaniappan, K., Bunyak, F., Nath, S.K., Goffeney, J.: Parallel Processing Strategies for Cell Motility and Shape Analysis. In: High-Throughput Image Reconstruction and Analysis, vol. (3), pp. 39–87 (2009)

17. Kumar, P., Palaniappan, K., Mittal, A., Seetharaman, G.: Parallel Blob Extraction Using the Multi-core Cell Processor. In: Blanc-Talon, J., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2009. LNCS, vol. 5807, pp. 320–332. Springer, Heidelberg (2009)
18. Palaniappan, K., Vass, J., Zhuang, X.: Parallel robust relaxation algorithm for automatic stereo analysis. In: SPIE Proc. Parallel and Distributed Methods for Image Processing II, vol. (3452), pp. 958–962 (1998)
19. Palaniappan, K., Faisal, M., Kambhamettu, C., Hasler, A.F.: Implementation of an automatic semi-fluid motion analysis algorithm on a massively parallel computer. In: 10th IEEE Int. Parallel Processing Symp., pp. 864–872 (1996)
20. Bellens, P., Palaniappan, K., Badia, R.M., Seetharaman, G., Labarta, J.: Parallel Implementation of the Integral Histogram. In: Blanc-Talon, J., Kleihorst, R., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2011. LNCS, vol. 6915, pp. 586–598. Springer, Heidelberg (2011)
21. Bilgic, B., Horn, B.K.P., Masaki, I.: Efficient integral image computation on the GPU. In: IEEE Intelligent Vehicles Symposium (IV), pp. 528–5338 (2010)
22. Kirk, D.: Nvidia CUDA software and GPU parallel computing architecture. In: ACM Proc. 6th Int. Symp. Memory Management (ISMM), pp. 103–104 (2007)
23. Nvidia Corp.: CUDA C Programming Guide 4.0 (2011)
24. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA GPU. In: Gems, vol. 3, ch. 39, pp. 851–876 (2007)
25. Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in CUDA Nvidia CUDA SDK Application Note (2009)
26. Air Force Research Laboratory: Columbus Large Image Format (CLIF) dataset over Ohio State University (2007)