Parallel Implementation of the Integral Histogram

Pieter Bellens[1], Kannappan Palaniappan[4], Rosa M. Badia[1,3], Guna Seetharaman[5], and Jesus Labarta[1,2]

[1] Barcelona Supercomputing Center, Spain
[2] Universitat Politecnica de Catalunya, Spain
[3] Intelligence Research Institute (IIIA) Spanish National Research Council (CSIC), Spain
[4] Dept. of Computer Science, University of Missouri, Columbia, Missouri, USA
[5] Air Force Research Laboratory, Information Directorate, Rome, New York, USA

Abstract. The integral histogram is a recently proposed preprocessing technique to compute histograms of arbitrary rectangular gridded (i.e. image or volume) regions in constant time. We formulate a general parallel version of the the integral histogram and analyse its implementation in Star Superscalar (StarSs). StarSs provides a uniform programming and runtime environment and facilitates the development of portable code for heterogeneous parallel architectures. In particular, we discuss the implementation for the multi-core IBM Cell Broadband Engine (Cell/B.E.) and provide extensive performance measurements and tradeoffs using two different scan orders or histogram propagation methods. For 640×480 images, a tile or block size of 28×28 and 16 histogram bins the parallel algorithm is able to reach greater than real-time performance of more than 200 frames per second.

1 Introduction

Regional histograms are widely used in a variety of computer vision tasks including object recognition, content-based image retrieval, segmentation, detection and tracking. Sliding window search methods using histogram measures produce high-quality results but have high computational cost. The integral histogram is a recently proposed preprocessing technique that abates this cost and enables the construction of histograms of arbitrary rectangular gridded (i.e. image or volume) regions in constant time.

The integral histogram effectively enables exhaustive global search using sliding window-based histogram optimization measures to yield high-quality results [1]. Fast histogram computation using the integral histogram speeds up the sequential implementation by up to five orders of magnitude. However, the overall cost remains prohibitive for real-time applications with large images, large search window sizes and a large number of histogram bins. For example, a 512×512 image search using a 1000-bin feature histogram requires about 1Gb of memory and takes about one second [2]. The computation of such dense confidence maps

J. Blanc-Talon et al. (Eds.): ACIVS 2011, LNCS 6915, pp. 586–598, 2011.

remains infeasible for these types of applications. A parallel implementation of the integral histogram would enable methods using global optimization of histogram measures to be competitive with or faster than other approaches in terms of speed. There are a variety of commodity multicore architectures currently available for the parallelization of image- and video-processing algorithms, including IBM's Cell/B.E., GPUs from NVidia and AMD and many-core CPUs from Intel [3]. The dramatic growth of digital video content has been a driving force behind active research into exploiting heterogeneous multi-core architectures for computationally intensive, multimedia analysis tasks. These include real-time (and super-real-time) object recognition, object tracking in multi-camera sensor networks, stereo vision, information fusion, face recognition, biometrics, image restoration, compression, etc. [4–10]

In this paper we focus on a fast parallel integral histogram computation to improve performance for real-time applications. Section 2 defines the integral histogram and two propagation methods. Next the definition is subjected to a block data layout in Section 3 where we identify parallel tasks and task precedence. This description of the parallel integral histogram can easily be encoded in the Star Superscalar (StarSs) programming model (Sections 4 and 5). We discuss some performance results for the Cell Broadband Engine (Cell/B.E.) in Section 6 and elaborate on future directions of this work in Section 7.

2 Computation of the Integral Histogram

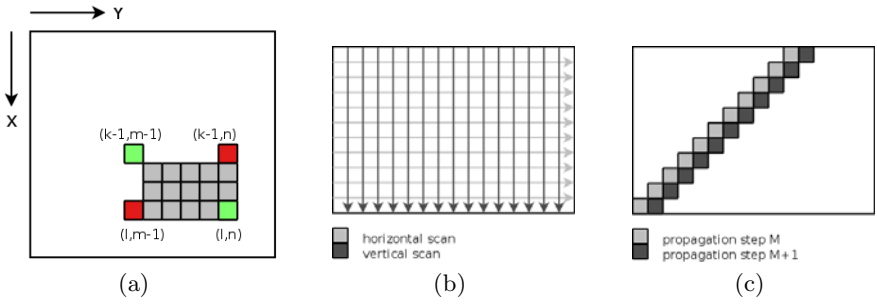
In accordance with the original formulation in [11] we define an image as a function f over a two-dimensional Cartesian space $\mathcal{R}^2$ such that $\mathbf{x} \rightarrow f(\mathbf{x})$ for a pixel $\mathbf{x} \in \mathcal{R}^2$. f can be single-valued or multi-valued. This affects the binning function but not the algorithm itself. The binning function $Q(f(\mathbf{x}), b)$ evaluates to 1 if $f(\mathbf{x}) \in b$ for the bin b, otherwise its value equals 0.

For a sequence of pixels $S = \mathbf{x}^0, \mathbf{x}^1, \ldots, \mathbf{x}^p$ and some $S_{\mathbf{x}^p} \subseteq S$ the integral histogram $H(\mathbf{x}^p, b)$ for bin b is defined as

$$H(\mathbf{x}^p, b) = \sum_{\mathbf{x} \in S_{\mathbf{x}^p}} Q(f(\mathbf{x}), b) \tag{1}$$

S forms the *scan order*. This definition states that the value for a bin b at a pixel $\mathbf{x}^p$ in the integral histogram can be found by applying the binning function to a subset of the pixels preceding $\mathbf{x}^p$ in the scan order. The structure of the computation resembles the propagation of pixel values throughout the image f. To emphasize the two-dimensional nature of f we can identify the vector $\mathbf{x}$ with its spatial coordinates (i, j). This carries over to H and f, that become $H(i, j, b)$ and $f(i, j)$ in this notation.

The integral histogram fixes the scan order so that the computation of the histogram for a rectangular region T of f becomes computationally inexpensive. In that context the computation of the integral histogram, or *propagation*,



(a) (b) (c)

Fig. 1. (a) Intersection or computation of the histogram for the region T defined by $\{(k,m),(k,n),(l,m),(l,n)\}, k < l, m < n$. (b) The cross-weave scan results in two passes over the image. The rows and the columns can be updated independently. (c) The wavefront scan, corresponding to Porikli's active set of points, requires just one pass over the image but has a more complex access pattern which is harder to parallelize.

precedes the computation of a histogram or *intersection*. We only consider scan orders that result in

$$H(i,j,b) = \sum_{x=0}^{i} \sum_{y=0}^{j} Q(f(x,y),b) \tag{2}$$

This means that the integral histogram at (i,j) reflects the values of all the pixels above and to the left of (i,j). The intersection for the region T (Figure 1(a)) delimited by the points $\{(k,m),(k,n),(l,m),(l,n)\}, k < l, m < n$ then reduces to the combination of four integral histograms:

$$\begin{aligned} H(T,b) =& H(k-1,m-1,b) + H(l,n,b) \\ & - H(k-1,n,b) - H(l,m-1,b) \end{aligned} \tag{3}$$

Porikli [11] describes two scan orders or algorithms for propagation: a string scan and a scan using an "active set of points". The string scan does not satisfy condition (2) and gives rise to a different intersection. We formulate a variant here. Same as the original string scan, our cross-weave scan requires two passes over the image but it satisfies condition (2) and can be parallelized. The method using an active set of points corresponds to the wavefront scan in this paper. We use data flow equations to describe the propagation steps instead of a traditional algorithmic description. In the following definitions the left-hand side can be computed only if all of its terms or components have been evaluated previously. The cross-weave scan processes the image in each dimension separately and accumulates the results in the Y- and the X-direction:

$$\begin{aligned} &(1)\ H(i,j,b) = 0 \\ &(2)\ H(i,j,b) = Q(f(i,j),b) + H(i,j-1,b), j > 0 \\ &(3)\ H(i,j,b) = H(i,j,b) + H(i-1,j,b), i > 0 \end{aligned}$$

The horizontal pass (step (2)) updates the histograms independently for each row, as does the the vertical pass (step (3)) for each column. These passes can be reordered. After the last step condition (2) holds. Propagation using a wavefront scan performs a single pass over the input image. The integral histogram is computed by propagating an anti-diagonal wavefront calculation. The histogram for each pixel combines the histograms of its top, left and top left neighbor, while incrementing the associated bin:

$$\begin{aligned} (1)\ H(i,j,b) &= 0 \\ (2)\ H(i,j,b) &= H(i-1,j,b) + H(i,j-1,b) \\ &\quad -H(i-1,j-1) + Q(f(i,j),b) \end{aligned}$$

This minor diagonal, pixel-level scan does not have a straightforward parallel interpretation like the cross-weave scan.

3 Parallelization Using Tiles and Scan Propagation

We derive a tiled or block-based version of the integral histogram that accesses data in d-dimensional chunks, where $d = 2$ for images. The calculations on tiles or blocks and the associated data transfers can then be expressed as a set of partially ordered tasks for the StarSs programming model (Section 4). The block data layout enables scalable and generalizable access to large multidimensional datasets in a regular manner with predictable and consistent performance.

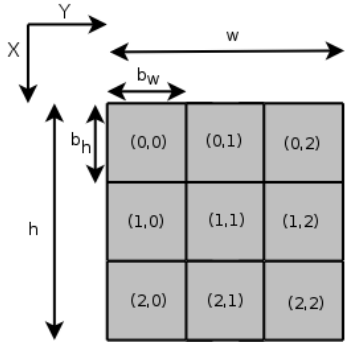


Fig. 2. Tiled image (integral histogram) or block data layout for a $w \times h$ image. Each tile contains $b_w \times b_h$ pixels (histograms).

Figure 2 illustrates the tiled or block data layout for the image and the integral histogram. We preserve the usual coordinate system, but now a coordinate pair identifies a tile instead of individual pixels or histograms. An image of dimensions $w \times h$ has an integral histogram of $(w \times h) \times b_c$ bins, with b_c the number of bins in a histogram. It admits a division into blocks of $b_w \times b_h$ pixels, whereas the integral histogram can be decomposed in tiles of $b_w \times b_h$ histograms. Conversely, each tile of the integral histogram holds $b_w \times b_h \times b_c$ bins. The image can be padded to eliminate boundary conditions. In the block data layout the image as well as the integral histogram consist of $w_B \times h_B$ blocks with $w_B = \lceil w/b_w \rceil$ and $h_B = \lceil h/b_h \rceil$ blocks. We distinguish between blocks and individual elements using a slightly different notation. Blocks $f_{i,j}$ and $H_{i,j}$ correspond to the tiles at row i and column j in the block data layout of the image and the integral histogram respectively, while (i, j) designates a pixel or a histogram depending on the context.

The chunking of image data into equal-sized 2D tiles requires propagation of partial integral histogram information inside and between tiles. The propagation for the parallel implementation naturally extends the cross-weave pattern or the wavefront pattern from Section 2. The wavefront scan processes the elements from left to right and from top to bottom within a block $H_{i,j}$ and iterates over the bins for each histogram. On the level of tiles the computation of $H_{i,j}$ requires $f_{i,j}$ as input, together with $H_{i,j-1}, H_{i-1,j}$ and $H_{i-1,j-1}$. In the first pass the cross-weave scan computes $H_{i,j}$ with $H_{i,j-1}$ and $f_{i,j}$ as input. The second pass updates $H_{i,j}$ using $H_{i-1,j}$. We will restrict our attention to the case of the wavefront scan. The development of the algorithm for the cross-weave scan is analogous.

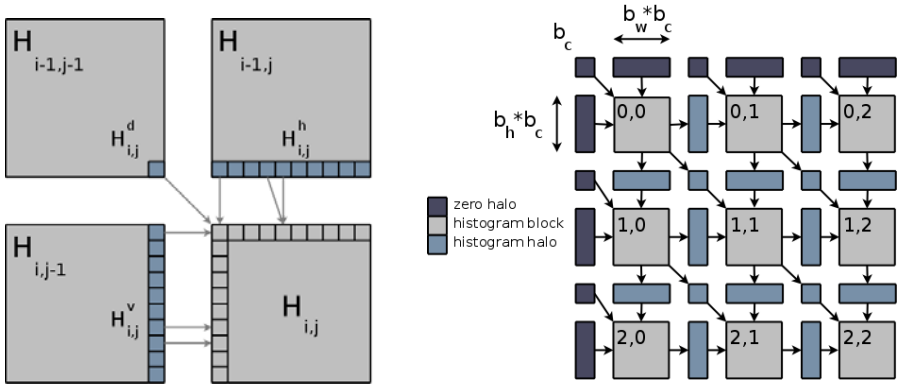


Fig. 3. (a) Inter-block dependencies for block $H_{i,j}$ with communication halos or aprons for propagation between tiles. For some histograms we detailed the inter-block dependencies with arrows. (b) Tiled layout for the integral histogram. Each block contains $b_w \times b_h$ histograms. The halos duplicate the histograms at the borders and are used to pass histograms to the neighboring blocks.

The aforementioned dependence between blocks stems from the dependence between elements contained in those blocks. Figure 3(a) identifies three sets or *halos* for a tile $H_{i,j}$, namely $H^d_{i,j}, H^h_{i,j}$ and $H^v_{i,j}$. The subscript identifies the tile whose edges require the histograms in these halos. Halos or *aprons* enable the flow of information between adjacent tiles [12].The singleton $H^d_{i,j}$ contains the histogram $(b_h - 1, b_w - 1)$ of the diagonally opposite block $H_{i-1,j-1}$, while $H^v_{i,j} = \{(l, b_w - 1) \in H_{i,j-1} | l = 0, \ldots, b_h - 1\}$ and $H^h_{i,j} = \{(b_h - 1, l) \in H_{i-1,j} | l = 0, \ldots, b_w - 1\}$. Element $(0,0)$ of $H_{i,j}$ depends on $H^d_{i,j}$, elements $(k, 0), k = 0, \ldots, b_h - 1$ need $H^v_{i,j}$ and finally the propagation for elements $(0, k), k = 0, \ldots, b_w - 1$ uses $H^h_{i,j}$.

We can consequently identify each tile $f_{i,j}$ (or $H_{i,j}$) with a unit of computation or a *task* $t_{i,j}$. This map defines the task precedence via the tile dependencies: $t_{i,j}$ is eligible for execution if all tasks $t_{i-k,j-l}$ with $0 < k \leq i, 0 < l \leq j$ have

been computed. In particular, $t_{i,j}$ accepts as input arguments the tile $f_{i,j}$ and the halos $H^h_{i,j}, H^v_{i,j}$ and $H^d_{i,j}$. Its output consists of the integral histogram block $H_{i,j}$ and the halos $H^h_{i+1,j}, H^v_{i,j+1}$ and $H^d_{i+1,j+1}$. Figure 3(b) explicitly allocates buffers for the halos to illustrate the data flow of the computation. Strictly considered these halos serve no practical purpose: they are conceptual constructs rather than actual data structures. We will see in Section 5 however that a physical instantiation of the halos helps in the implementation of the algorithm in StarSs.

4 Star Superscalar (StarSs) Parallel Programming Model

The StarSs programming model [13, 14] provides a convenient way to parallelize sequential code for various parallel architectures. It generally suffices that the user adds pragmas to the original code to mark the functions (or *tasks*) intended to execute on the parallel resources. The StarSs source-to-source compiler converts these pragmas into calls to the StarSs runtime library. As the application advances the StarSs runtime executes the tasks in parallel as dictated by the data dependencies present in the original program.

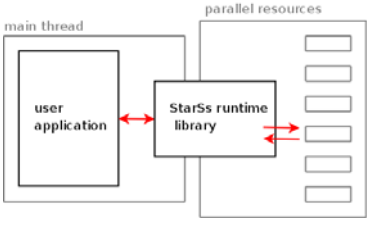


Fig. 4. Overview of the structure of a StarSs application

The main thread of a StarSs application executes the sequential code and switches to the StarSs libraries when it encounters a call to a function marked with a pragma. The StarSs runtime does not immediately execute this function or task. Instead it analyzes the task arguments to find the true dependencies that define task precedence. StarSs avoids output dependencies and anti-dependencies by renaming arguments. The main thread returns control to the user application and the StarSs runtime records the task in the Task Dependency Graph (TDG). Simultaneously the runtime schedules *ready* tasks (or tasks without outstanding dependencies in the TDG) to the resources or *workers*. Workers remove finished tasks from the TDG and update the state of dependent tasks. Ultimately this pruning of dependencies turns dependent tasks into ready tasks that again become scheduling candidates.

In StarSs dependence analysis, renaming, scheduling and updates to the TDG take place concurrently at run-time. These facilities convert a sequential stream of tasks generated by a single thread into a TDG into a parallel execution of tasks on multiple resources. This model has a strong resemblance to dynamic scheduling in superscalar processors. The pipeline there decodes instructions in order but schedules them to multiple units in parallel to the extent allowed by the data dependencies.

5 Implementation Using Tile-Level Halos

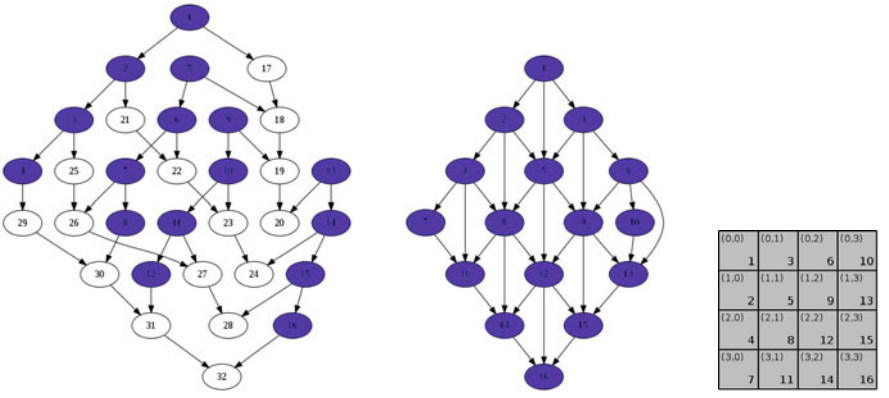
The block algorithms of Section 3 can be implemented in a straightforward manner in StarSs. The main function is a sequential implementation of the respective scan order. For the wavefront scan the code steps through the blocks in the the diagonals parallel to the minor diagonal of the blocked integral histogram. At run-time this generates the sequence of tasks $t_{0,0}, t_{1,0}, t_{0,1}, t_{2,0}, t_{1,1}, t_{0,2}, \ldots$ It suffices to ensure that the data dependencies as exposed via the task arguments completely define the task precedence. To that end we allocate physical buffers for the halos as in Figure 3(b). For example, a separate location in main memory duplicates the elements of halo $H^h_{i,j}$ from block $H_{i-1,j}$. We refer to such a buffer with the name of the halo it contains. Our implementation explicitly models the data flow between the tiles of the integral histogram. StarSs then detects the data dependencies between the tasks of the integral histogram. Although this data representation is slightly redundant, it is a small price to pay in the light of automatic parallelization.

Both the image and the integral histogram are represented in block data layout (Section 3). They consist of $w_B \times h_B$ tiles. The halos $H^v_{i,j}$ occupy an additional $h_B \times b_h \times w_B \times b_c$ bins, the halos $H^h_{i,j}$ take up $w_B \times b_w \times h_B \times b_c$ bins and the $H^d_{i,j}$ $h_B \times w_B \times b_c$ bins. Task $t_{i,j}$ reads the halos $H^h_{i,j}, H^v_{i,j}$ and $H^d_{i,j}$ and aside from $H_{i,j}$ it produces $H^h_{i+1,j}, H^v_{i,j+1}$ and $H^d_{i+1,j+1}$, which in turn are read by $t_{i+1,j}, t_{i,j+1}$ and $t_{i+1,j+1}$. These chains of halo production and consumption establish the required data dependencies. Figure 6 depicts the TDG for the integral histogram for a small image size for both scan methods. The cross-weave scan visits each tile $H_{i,j}$ twice, once during the horizontal pass and next in the vertical pass, and generates twice as many tasks as the wavefront scan.

The storage requirements can be reduced by noting that the horizontal halos can be recycled per row, the vertical halos per column and the diagonal halos per diagonal. The lifetime of $H^h_{i,j}$ ends before $H^h_{i+1,j}$ is produced because task $t_{i,j}$ executes and finishes before $t_{i+1,j}, \forall j = 0, \ldots, w_B - 1$. The task precedence for this application guarantees that the accesses to $H^h_{i,j}$ do not overlap in time for fixed j. Similar observations hold for the vertical and diagonal halos. Hence there is no reason to separate the input and output halos of a task: $H^h_{i,j}, H^v_{i,j}$ and $H^d_{i,j}$ can occupy the same memory as $H^h_{i+1,j}, H^v_{i,j+1}$ and $H^d_{i+1,j+1}$ respectively. With this reduction the horizontal halos occupy $w_B \times b_w \times b_c$ additional bins, the vertical halos $h_B \times b_h \times b_c$ bins and the diagonal halos $(w_B + h_B - 1) \times b_c$ bins. As a side-effect the task can be defined with less parameters, because the halos that are read and written are one and the same. At run-time this translates to fewer arguments per StarSs task, less dependency analysis and less runtime overhead.

6 Experiments

We implemented the cross-weave scan and the wavefront scan in StarSs according to the specification in Section 5. The programming model ensures portability



(a) (b) (c)

Fig. 5. Task Dependency Graph for the tiled integral histogram ($w_B = h_B = 4$) with (a) the cross-weave scan and (b) the wavefront scan. The tasks are numbered according to program order, which is identical to the scan order in our implementation. The cross-weave scan has two different types of tasks, represented by the two different colors of its nodes. In (c) the tiles in the block data layout (Figure 2) are matched up with the corresponding tasks of the wavefront scan. The top left corner contains the coordinates of the tile and the bottom right corner the task number.

across a variety of platforms, but because of the limitation on the page count we restrict the scope to the Cell/B.E.. We chose this particular architecture because it belongs to the family of the heterogeneous multi-core architectures, which together with GPGPU shapes the form of high-performance computing today. And secondly because for this architecture peak performance is notoriously hard to achieve despite of its impressive potential. All measurements were performed on a QS20 Blade at the Barcelona Supercomputing Center. The default image size is 640×480 and the number of bins defaults to $16, 32, 64$ and 128.

6.1 Integral Histogram for Single Images

Figure 6 summarizes the performance of the cross-weave scan and the wavefront scan for a single image. The StarSs implementation accepts the block size as an input argument, so we repeated the measurements for different values.

This implementation proves to be very practical, because the block size for best performance is not uniform across different bin counts. E.g. the cross-weave scan for 16 bins performs best for blocks of 28×28 elements, whereas 128 bins require smaller blocks of 11×11. In general two factors determine the performance of a Cell/B.E. application. Computations on an SPE must be well balanced, so that code execution can seamlessly overlap the latency of the associated data transfers. This requirement relates to the granularity of tasks. If dynamic analysis (*i.e.* task creation, dependence analysis, scheduling,etc.) takes place while

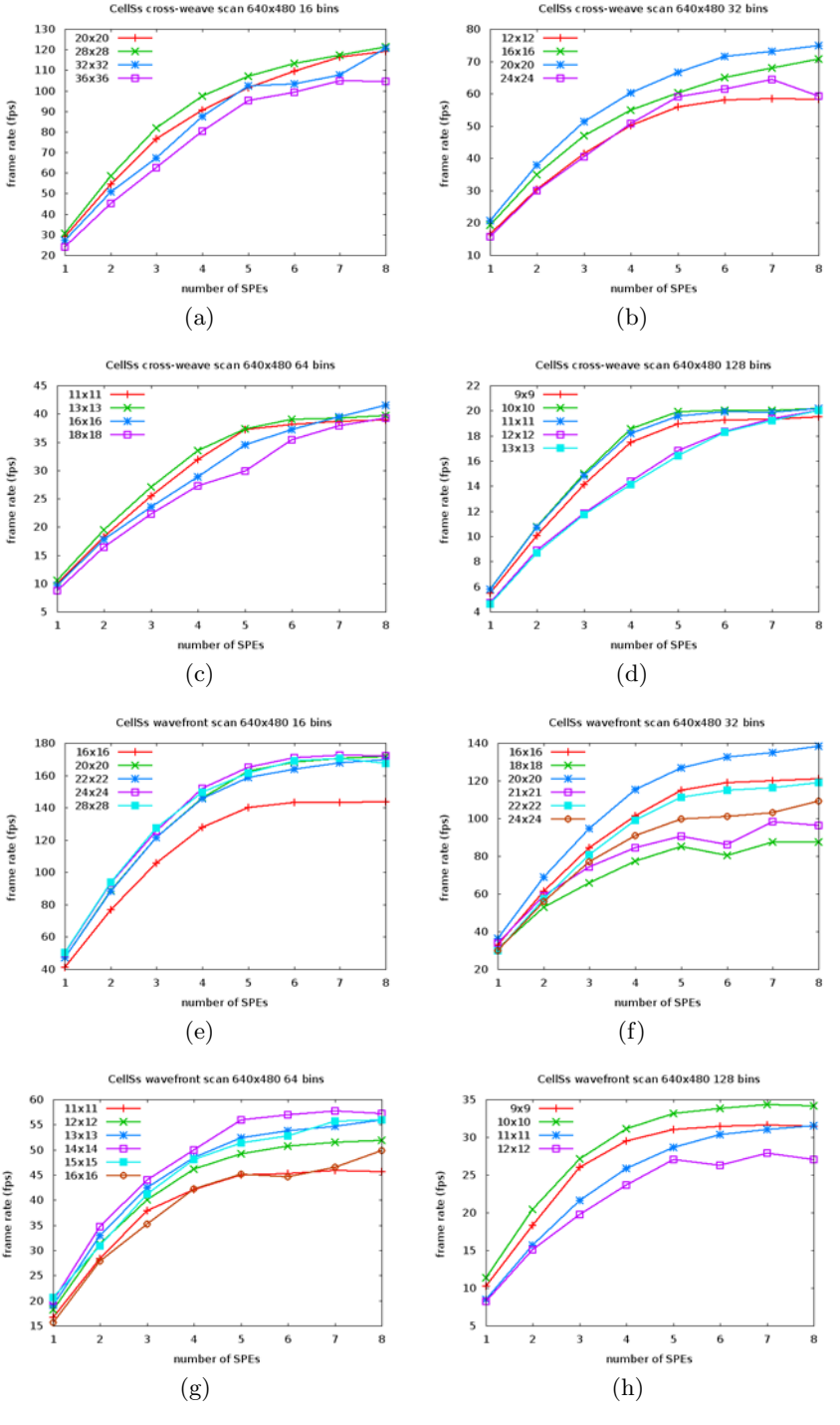


Fig. 6. Performance of the cross-weave (a,b,c,d) and wavefront scan (e,f,g,h) in CellSs on an 640x480 image for different block sizes and different numbers of bins

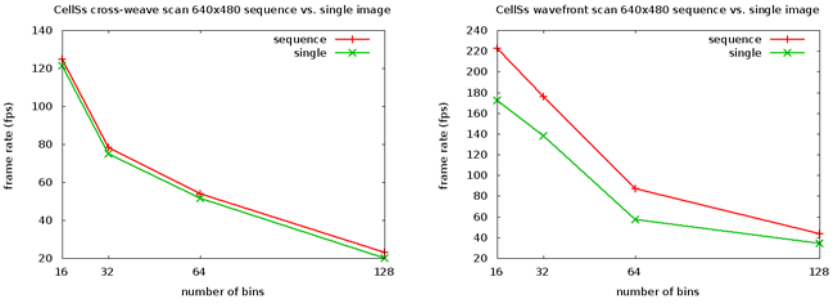


Fig. 7. Comparison of the cross-weave scan in (a) with the wavefront scan in (b) for varying numbers of histogram bins and 8 SPEs

the SPEs execute, run-time overhead influences the performance as well. Larger tiles divide the image or the integral histogram in fewer parts, which results in less tasks and less overhead at execution time. The results in Figure 6 can be interpreted and understood as balancing granularity versus run-time overhead. The wavefront scan (175 fps and 20 fps for 16 and 128 bins resp., speedup between 5 and 6 for 8 SPEs) generates less tasks than the cross-weave scan (120 fps and 35 fps for 16 and 128 bins resp., speedup between 4 and 5 for 8 SPEs) for the same block size and consistently outperforms the latter. For both propagation methods the performance improves as the tasks become larger and fewer, until the task granularity becomes prohibitive. Remark that the TDG for the wavefront scan has limited parallelism at the top and the bottom (Figure 5(b)); this algorithm inherently scales poorly. Figure 8 summarizes the performance and scalability for the wavefront scan for the optimal tile sizes (Figure 6) on different image sizes and for different bin counts.

6.2 Integral Histogram for Sequences of Images

Practical applications tend to process sequences of images but the results in Section 6.1 measure the frame rate on an isolated image. This practice fails to capture the performance of our implementation in production conditions. When incorporated into an image processing pipeline we expect performance to improve. For such a sequence the initialization and the shutdown of the StarSs libraries are amortized over multiple images, while for a single image these inherently sequential operations preclude good scalability. The wavefront scan additionally suffers from a TDG that allows limited parallelism at the beginning and the end of the execution (Figure 5(b)). The lack of ready tasks in the TDG of image i can be compensated by tasks from image $i + 1$. Figure 7 compares the performance for single images with the performance over multiple images for multiple bin counts. The tile size for each bin count corresponds to the optimal tile size from Figure 6 for the associated scan method.

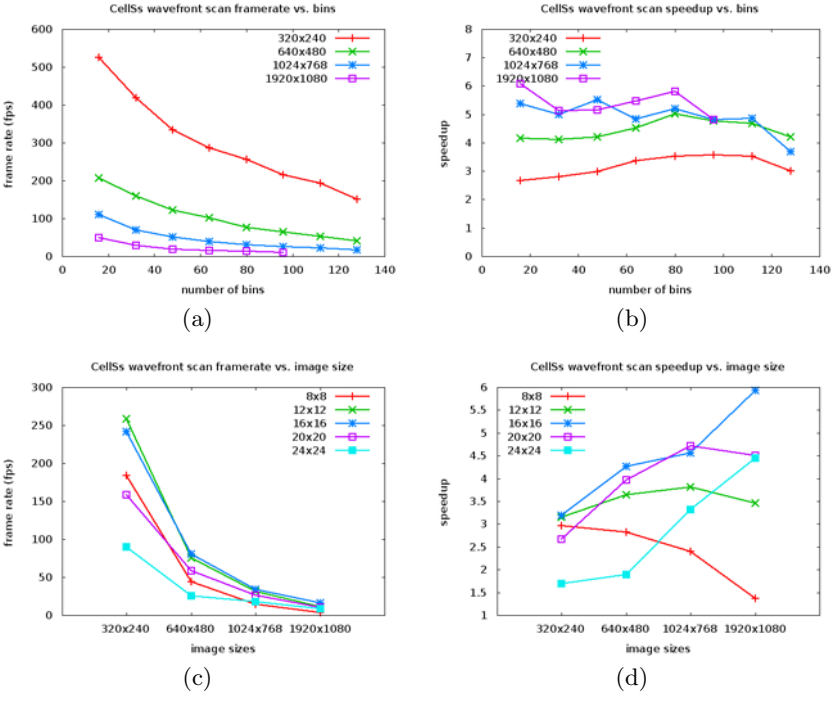


Fig. 8. Performance and speedup of the wavefront scan for different image sizes and bin counts, for the optimal tile sizes from Figure 6

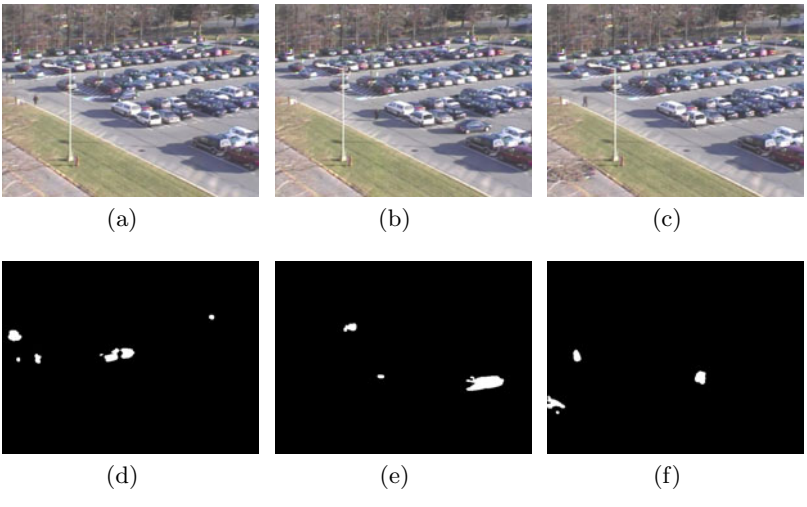


Fig. 9. Otsu-thresholded images using the integral histogram

7 Conclusions and Future Work

We described two parallel implementations of the integral histogram based on a block data layout, preserving the characteristic propagation of histograms in the original formulation of the algorithm. Each block corresponds to one (wavefront scan) or two (cross-weave scan) tasks. Inside the tiles we process the elements in row-major order, with halos passing histograms between tiles. This formulation has the advantage that it cleanly models the data flow and scales well for images of different sizes. As a result the implementation in StarSs was straightforward and easy. The cross-weave scan reaches 120 fps for histograms of 16 bins for images with dimensions 640×480. The wavefront scan has a more symmetric, but also a severely more restricted TDG. The reduction in tasks results in 220 fps for the same image size and bin count. However, it must be stressed that the code is portable and is being evaluated on all platforms supported by StarSs, including SMP and GPU. We are also in the process of developing applications based on the integral histogram. For example, Figure 9 illustrates the results from Otsu-thresholding using our implementation of the integral histogram on the motion energy filtering output produced by the flux tensor algorithm [15–17] using a sequence of surveillance images [18]. In order to improve the absolute performance for larger image sizes such as 1920×1080, image tiles could be distributed to multiple Cell/B.E. processors in a cluster configuration, or the processing of different images could be overlapped in time.

Acknowledgements. The authors acknowledge the support of the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625), the European Commission in the context of the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. This research was partially supported by grants to KP from the the U.S. Air Force Research Laboratory (AFRL) under agreements FA8750-10-1-0182 and FA8750-11-1-0073. 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. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The FPSS video sequences were provided by Dr. Alex Chan at the U.S. Army Research Laboratory.

References

1. Aldavert, D., de Mantaras, R.L., Ramisa, A., Toledo, R.: Fast and robust object segmentation with the integral linear classifier. In: IEEE Conf. Computer Vision and Pattern Recognition, pp. 1046–1053 (2010)
2. Wei, Y., Tao, L.: Efficient histogram-based sliding window. In: IEEE Conf. Computer Vision and Pattern Recognition, pp. 3003–3010 (2010)
3. Blake, G., Dreslinski, R.G., Mudge, T.: A survey of multicore processors. IEEE Signal Processing Magazine 26(6), 26–37 (2009)

4. Lin, D., Huang, X., Nguyen, Q., Blackburn, J., Rodrigues, C., Huang, T., Do, M.N., Patel, S.J., Hwu, W.-M.W.: The parallelization of video processing. IEEE Signal Processing Magazine 26(6), 103–112 (2009)
5. Shams, R., Sadeghi, P., Kennedy, R., Hartley, R.: A survey of medical image registration on multicore and the GPU. IEEE Signal Processing Magazine 27(2), 50–60 (2010)
6. Palaniappan, K., Bunyak, F., Kumar, P., Ersoy, I., Jaeger, S., Ganguli, K., Haridas, A., Fraser, J., Rao, R., Seetharaman, G.: Efficient feature extraction and likelihood fusion for vehicle tracking in low frame rate airborne video. In: 13th Int. Conf. Information Fusion (2010)
7. Mehta, S., Misra, A., Singhal, A., Kumar, P., Mittal, A., Palaniappan, K.: Parallel implementation of video surveillance algorithms on GPU architectures using CUDA. In: 17th IEEE Int. Conf. Advanced Computing and Communications, ADCOM (2009)
8. 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)
9. 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)
10. 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)
11. Porikli, F.: Integral histogram: A fast way to extract histograms in Cartesian spaces. In: IEEE Conf. Computer Vision and Pattern Recognition, pp. 829–836 (2005)
12. Podlozhnyuk, V.: Image convolution with CUDA. Technical report, NVIDIA Corp., Santa Clara, CA (2007)
13. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. Int. J. High Perform. Comput. Appl. 23(3), 284–299 (2009)
14. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM J. Res. Dev. 51(5), 593–604 (2007)
15. Palaniappan, K., Ersoy, I., Nath, S.K.: Moving object segmentation using the flux tensor for biological video microscopy. In: Ip, H.H.-S., Au, O.C., Leung, H., Sun, M.-T., Ma, W.-Y., Hu, S.-M. (eds.) PCM 2007. LNCS, vol. 4810, pp. 483–493. Springer, Heidelberg (2007)
16. Bunyak, F., Palaniappan, K., Nath, S.K., Seetharaman, G.: Flux tensor constrained geodesic active contours with sensor fusion for persistent object tracking. J. Multimedia 2(4), 20–33 (2007)
17. Bunyak, F., Palaniappan, K., Nath, S.K., Seetharaman, G.: Geodesic active contour based fusion of visible and infrared video for persistent object tracking. In: 8th IEEE Workshop Applications of Computer Vision (WACV 2007), Austin, TX, pp. 35–42 (February 2007)
18. Chan, A.L.: A description on the second dataset of the U.S. Army Research Laboratory Force Protection Surveillance System. Technical Report ARL-MR-0670, Army Research Laboratory, Adelphi, MD (2007)