Optimizing co-occurrence matrices on graphics processors using sparse representations

Francisco Igual[1], Rafael Mayo[1], Timothy D. R. Hartley[2], Umit Catalyurek[2], Antonio Ruiz[3], Manuel Ujaldón[3]

[1] Department of Computer Engineering and Computer Science
University Jaume I, Castellon, Spain
{figual, mayo}@icc.uji.es

[2] Departments of Biomedical Informatics and Electrical and Computer Engineering
The Ohio State University, Columbus, OH, USA
{hartleyt, umit}@bmi.osu.edu

[3] Computer Architecture Department
University of Malaga, Spain
{aruiz, ujaldon}@ac.uma.es

Abstract. Textural features extracted from co-occurrence matrices have shown quite effective in pattern recognition and image classification algorithms at the expense of a very high computational cost. In this work, we present a novel implementation on graphics processors (GPUs) for a fast calculation of parameters based on co-occurrence matrices. Our approach focuses on CUDA programming for exploiting the parallelism and the computational power of a GPU and derive high-performance methods which optimize the use of the video memory hierarchy. Within this framework, several sparse matrix formats are proposed and analyzed for a dynamic handling of elements to address the matrix fill-in at run-time. Experimental results are compared among different strategies and versus a counterpart implementation running on multicore CPUs.

1 Introduction

Texture analysis methods have been utilized in a wide variety of application domains, such as quality control, remote sensing, textile inspection, character recognition, document processing, medical image analysis and computer vision [1973, 1992, 2007]. Co-occurrence matrices are effective and extensively used as a tool for discriminating different textures, though they have the disadvantage of a high computational cost. Co-occurrence matrices can be computed at image-level and at pixel level, by only considering a *window* around that pixel, in order to compute image and pixel level texture features, respectively. In this paper, our focus is per-pixel co-occurrence matrix computation, because it is much more computationally demanding. In particular, our contribution is in high-performance implementation of co-occurrence matrices on graphics processors (GPUs). Clearly most of the optimization techniques we present in this work is applicable to both per-pixel and per-image co-occurrence matrix computation.

A co-occurrence matrix represents how often a pixel with the intensity value i occurs in a specific spatial relationship to a pixel with the intensity value j. The spatial relationship is defined by a displacement vector $d = (dx, dy)$ where dx and dy are the displacements in columns and rows of image, respectively. Each element (i, j) in the co-occurrence matrix is simply the sum of the number of times that the pixel with value i occurred in the specified spatial relationship to a pixel with value j in the input image (see Figure 1).

The displacement vector establishes the distance apart along a given direction having co-occuring pixel values. Often only the distances d=1 and 2 pixels are considered, combined with directions in horizontal, vertical and both diagonals. The direction in which pixels are traversed for composing the pairs when voting on co-occurrence matrices is tightly coupled to the way the matrix is stored through compressed formats when using sparse matrices. Along this paper, we focus on the $d = (1, 0)$ rowwise storage of matrix to illustrate our methods, that could be straightforwardly adapted to the $d = (0, 1)$ columnwise storage.

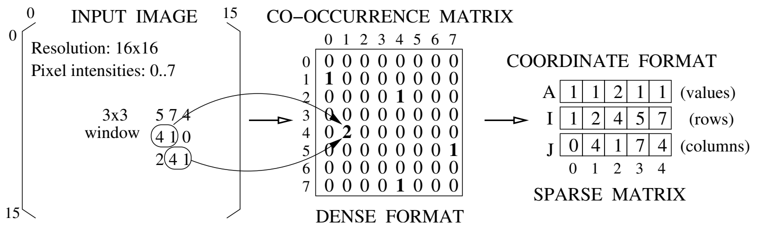

Fig. 1. A 8x8 co-occurrence matrix computed from a 3x3 window on a particular pixel of a 16x16 image, and its subsequent storage using a coordinate format.

2 Sparse formats for co-occurrence matrices

The co-occurrence matrix for a given pixel is usually computed for a square window of neighbor pixels centered on it, with the size of the output matrix given by the number of intensity levels in the image (typically 256x256 unless the matrix be discretized). This often leads to a sparse co-occurrence matrix (see Figure 1). For example, using a window size of 64x64 pixels, at least 93.75% of the 256x256 elements of a co-occurrence matrix are zeros. Optimizing the memory usage is of great interest on a GPU under CUDA programming, since only a very small ultra-fast memory is available per CPU-core or per multiprocessor of recent GPU cards.

A number of storage formats are available as sparse representations [1991]. For our purposes, we have to select a format fulfilling two major premises: (1) To be simple enough to be adapted to the way the GPU computes, and (2) to be compact enough to store as many co-occurrence matrices as possible. This way, a number of threads may run in parallel on the GPU with each thread storing its results on a local output. In our framework, the overall computational load is decomposed by partitioning the window to process into a number of threads, say N. Each thread concurrently updates its own local co-occurrence matrix and at the end of the computation a reduction process takes place to build the global co-occurrence matrix from the local ones in $log(N)$ stages as shown in Figure 2.

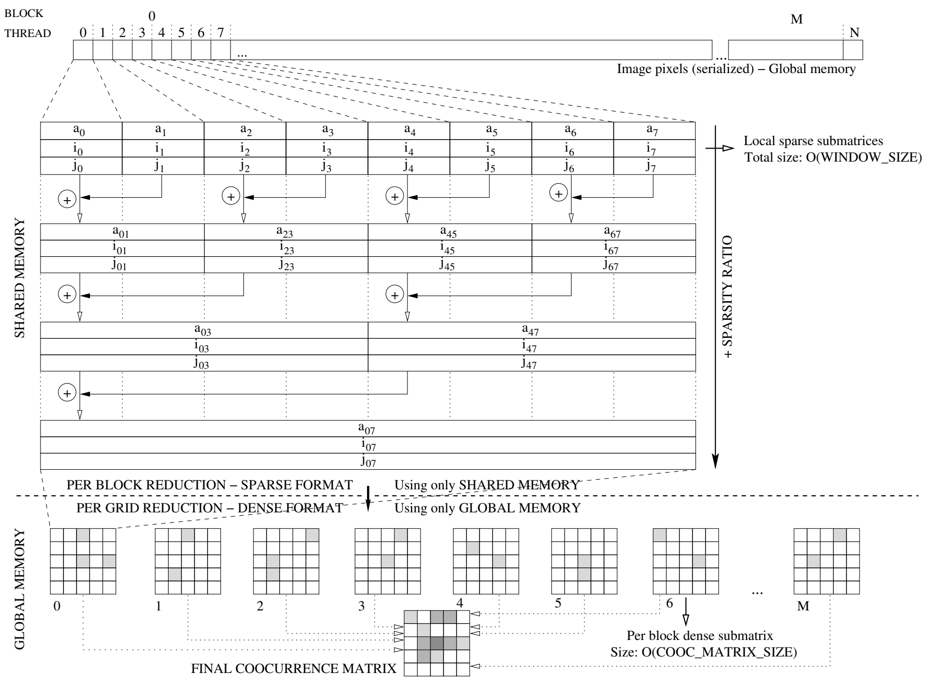
BLOCK
THREAD
0
M
N
Image pixels (serialized) – Global memory
Local sparse submatrices
Total size: O(WINDOW_SIZE)
SHARED MEMORY
+ SPARSITY RATIO
PER BLOCK REDUCTION – SPARSE FORMAT
Using only SHARED MEMORY
PER GRID REDUCTION – DENSE FORMAT
Using only GLOBAL MEMORY
GLOBAL MEMORY
Per block dense submatrix
Size: O(COOC_MATRIX_SIZE)
FINAL COOCURRENCE MATRIX

Fig. 2. The way co-occurrence matrices are computed during our GPU implementation using CUDA.

Among available sparse matrix representations, the most commonly used ones are Compressed Row/Column Storage (CRS/CCS) formats [1994]. These are a good choice when the algorithm does not create new nonzero elements at run-time (fill-in), which does not suite well with our application. Therefore, for the initial implementation we have chosen to use a coordinate format, where nonzeros are stored on an array in row-wise order and are accompanied by its pair of coordinates (i,j) in other two additional vectors (see Figure 1). In addition, we maintain an index which emulates a linked list to keep track of the first free position within the data vector. Other sparse formats are currently under development and will be analyzed in the final version of the paper as well.

Table 1. Execution times (in msecs.) for the computation of a 256x256 co-occurrence matrix on a single pixel under different window sizes.

Window size	CPU on dense matrices	GPU on dense matrices stored in global mem.	GPU on sparse matrices stored in both memories (shared + global)	Maximum sparsity rate (% nonzeros)	Sparse speed up on GPU	GPU speed up vs. CPU
4x4	1.36	7.61	0.06 + 0.04 = 0.10	0.0244%	76.10x	13.60x
8x8	2.82	7.62	0.06 + 0.10 = 0.16	0.0976%	47.62x	17.62x
16x16	2.82	7.58	0.28 + 0.11 = 0.39	0.3906%	19.43x	7.23x
32x32	3.04	7.63	0.29 + 0.45 = 0.74	1.5625%	10.31x	4.10x
64x64	3.08	7.76	0.84 + 0.90 = 1.74	6.2500%	4.45x	1.77x
128x128	2.94	8.54	5.89 + 1.81 = 7.70	25%	1.10x	0.38x
256x256	2.96	9.19	42.90 + 3.59 = 46.49	100%	0.19x	0.32x

Table 2. Execution times (in msecs.) for the computation of a co-occurrence matrix on a single pixel under different discretization levels. The window size is 16x16 pixels.

Size of co-oc. matrix	CPU on dense matrices	GPU on dense matrices stored in global mem.	GPU on sparse matrices stored in both memories (shared + global)	Maximum sparsity rate (% nonzeros)	Sparse speed up on GPU	GPU speed up vs. CPU
16x16	2.82	0.23	0.19 + 0.02 = 0.21	100%	1.09x	13.42x
32x32	2.82	0.31	0.25 + 0.02 = 0.27	25%	1.14x	10.44x
64x64	2.82	0.67	0.26 + 0.02 = 0.28	6.25%	2.39x	10.07x
128x128	2.82	2.09	0.29 + 0.04 = 0.33	1.56%	6.33x	8.54x
256x256	2.82	7.58	0.28 + 0.11 = 0.39	0.39%	19.43x	7.23x

3 Empirical results

Preliminary results on a state-of-the-art computer equipped with a Nvidia GeForce 8800GTX GPU and a Intel Core 2 Duo CPU are given in Table 1 for different window sizes and in Table 2 varying the discretization level of the output matrix. Factor gains increase on the GPU with the sparsity rate, reaching up to 50x when the matrix contains only 0.1% of nonzero values. On the contrary, numbers were worse in the CPU using sparse matrices, and therefore are not represented.

Acknowledgments

This work was partially supported by Fundación Caixa-Castellón/Bancaixa and the University Jaume I (P1-1B2007-32) of Spain, the Junta de Andalucía of Spain under Project TIC-02109, the Ministry of Education of Spain under Project TIC2003-06623, and the US National Science Foundation (CNS-0643969, CNS-0403342, CNS-0615155, CCF-0342615), the NIH NIBIB BISTI (P20EB000591) and the US Department of Energy (ED-FC02-06ER25775).

References

[1994] Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., van der Vorst, H.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* 2nd Edition. SIAM, 1994.

[1973] Haralick, R.M., Shanmugan, K. and Dinstein, I.: *Textural features for image classification.* IEEE Transactions on Systems, Man, and Cybernetics, volume 3, pages 610-621, 1973.

[1992] Jain, A.K. and Bhattacharjee, S.: *Text Segmentation Using Gabor Filters for Automatic Document Processing.* Machine Vision and Applications. Special Issue on document image analysis techniques. Volume 5, number 3, pages 169-184, 1992.

[2007] Ruiz, A., Sertel, O., Ujaldón, M., Catalyurek, U., Saltz, J., Gurcan, M.: *Stroma Classification for Neuroblastoma on Graphics Processors.* Intl. Journal of Data Mining and Bioinformatics, Ed. Inderscience, 2008 (to appear).

[1991] Zlatev, Z.: *Computational Methods for General Sparse Matrices.* Mathematics and its Applications Series, volume 65. Kluwer Academic Publisher. Holland, 1991.