Data Structures for Images on Java

Hans Braxmeier

Department of Applied Information Processing and Department of Stochastics
University of Ulm
Helmholtzstr. 18, D-89069 Ulm, Germany
hans.braxmeier@mathematik.uni-ulm.de
http://www.mathematik.uni-ulm.de/sai/braxmeier/

Abstract. Many data structures for images are limited to dimension two. Furthermore, common transformations for images are usually not supported (efficiently). In this paper, efficient Java data structures for images are discussed and their performance is compared.

Keywords: image processing, data structures, Java
Classification: first year of PhD studies

1 Introduction

For medical applications, e.g. in computer tomography, data are often given as three-dimensional images. Therefore, the image data structures supporting only two-dimensional images are not sufficient. For this reason data structures for images should be independent of the dimension or at least for dimension three. Furthermore, all commonly image transformations should be possible without great effort. Such image transformations are:

- **Subrange**: yields a rectangle subset of the image. This is important in processing parts of an image.
- **Reflection**: reflects the image at the origin. This is, e.g., necessary for the computation of the autocorrelation of an image.
- **Reduction of Dimensions**: yields an intersection of the image with a coordinate plane. This is used for analyzing sections of images, e.g., for medical purposes.

After an explanation and comparison of four possible image data structures in Section 2, the results of performance tests are compared in Section 3. In Section 4 related work is discussed and a summary of the results is given in Section 5.

2 Data Structures for Images

In this section, four data structures for images are explained. Their structure as well as the advantages and shortcomings of these data structures are discussed with respect to image representation.

In the following examples, assume the minimal values of all coordinates to be zero.

2.1 Multi-Dimensional Arrays in C

No matter how many dimensions are declared, a multi-dimensional array in C is one single array. The access to the element with indices x and y of a two-dimensional array is shown in Figure 1.

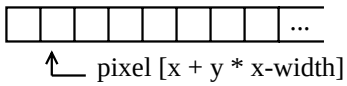


Fig 1. Two-dimensional arrays in C

Due to the addressing mechanism, the image transformations, presented in Section 1, cannot be implemented without running through each element of the array. But this addressing mechanism can easily be extended to higher dimensions.

2.2 Multi-Dimensional Arrays in Java

Multi-dimensional arrays in Java are arrays with references to arrays; the individual array components may themselves be references to arrays of different lengths as shown in Figure 2.

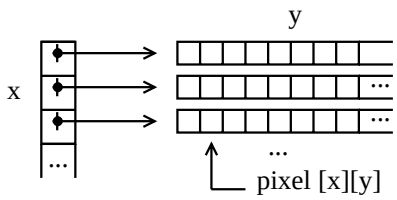


Fig 2. Two-dimensional arrays in Java

The main disadvantage of this adressing mechanism is the multiple indirection depending on the dimension of the image, which entails loss of performance. Also the data structure would be different in structure for different dimensions.

Just as multi-dimensional arrays in C, this data structure does not (directly) support the mentioned transformations, except reflecting the image at the y-axis (reordering the pointers of the vertical array in Figure 2).

2.3 Index Arrays

For every dimension, this addressing mechanism uses an extra array with indices. The sizes of these arrays correspond to the number of elements of each dimension. A pixel can be addressed by summing up the indices of all index arrays as shown in Figure 3.

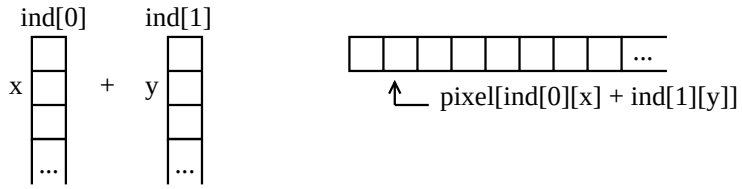


Fig 3. Two-dimensional indexed arrays

This index data structure for images is independent of the dimension of the image. The more important aspect of this adressing mechanism is that the discussed transformations can be implemented without running through the elements. However, in contrast to the the multi-dimensional array in C and Java, this data structure needs additional space for the index arrays.

2.4 Strides and Offset

As shown in Figure 4, an array with strides and offset is also based on a one-dimensional array.

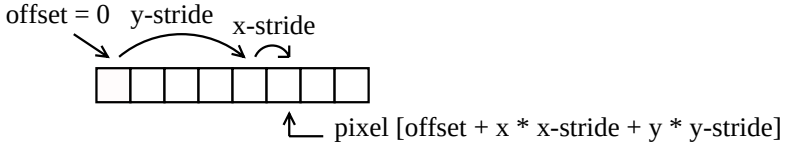


Fig 4. Two-dimensional arrays with Strides and Offset

The index of the origin is given by the offset. Strides allow to process a whole image in a very fast and efficient way: if the index of a pixel is known, only one addition is necessary to calculate the index of the successing pixel, independent of the dimension. For the direct access one addition and one multiplication is necessary for each dimension. Index arrays need only as many additions as dimensions.

Similarly to the index arrays, this data structure is independent of the dimension. All the transformations can be implemented without running through the elements, and the overhead is minimal compared to multi-dimensional arrays in C or Java and less than the overhead of index arrays.

3 Performance Measurements

In this section the performance of the presented data structures is evaluated. The attention is payed to index arrays and strided arrays. In both cases, different methods are implemented to retrieve and modify a pixel:

- Direct access to the used array
- Method access where the coordinates are given as integers
- Method access where the coordinates are given as an integer array

Every method uses an XOR operation between two images of the same size, which ensures that all pixels of both images are touched. The implementations are tested on images of different sizes: 2 MB, 20 MB and 200 MB.

All experiments are conducted on a PentiumIII@800MHz, 64KB L1, 512KB L2, 640MB, SuSE Linux8.0, Kernel2.4.18. and on a SunEnterprise450, 4xUltra SPARC-II@400MHz, 1152 MB, Solaris 2.8. The programs are tested with the JavaHotSpot1.3.1 JVM and the g++2.95.3 compiler on the SPARC and, additionally with the IBMJava1.3.0 JVM/JIT on the PC. Measurements are given in user times to compare the performance of the implementations as shown in Figures 5, 6 and 7.

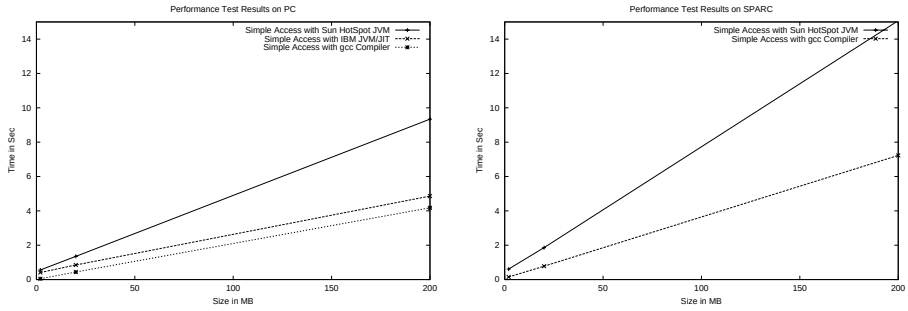


Fig 5. Simple Arrays

4 Related Work

Hoscheck [1] implements the index data structure in Java. His main focus is fixed on rectangular dens and sparse multi-dimensional matrices. Accessing elements is done by set-and-get methods which causes loss of performance since method calls are expensive. For image processing, the sparse case is not relevant.

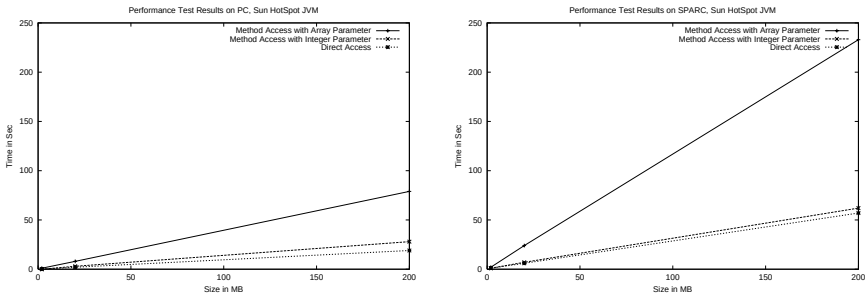


Fig 6. Index Arrays

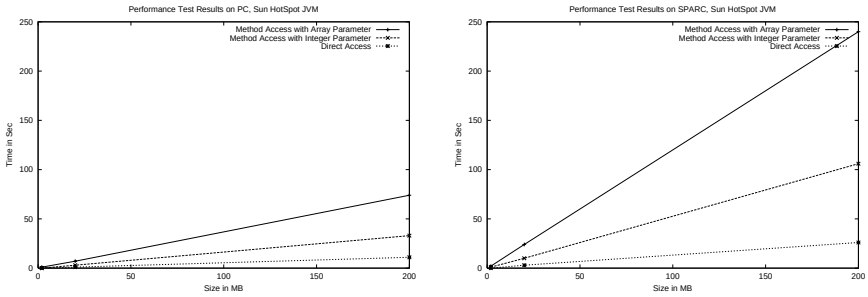


Fig 7. Arrays with Strides and Offset

5 Summary and Conclusion

In this paper, different image operations and four data structures for images were analyzed. The performance of every implementation was tested.

Figure 5 shows that for both the gcc compiler and the IBM JVM/JIT, the time difference is nearly constant for every image size. The small difference is only a result of the compilation through the IBM JIT. Figure 5 shows also that it is advisable to use the IBM JVM/JIT on PCs instead of the Sun HotSpot JVM. As it can be seen in Figures 6 and 7 it is obviously that direct access using arrays with strides and offset is the fastest implementation.

Acknowledgment

I would like to thank Johannes Mayer for the helpful discussions and ideas.

References

[1] W. Hoschek: *Uniform, Versatile and Efficient Dense and Sparse Multi-Dimensional Arrays.* Preprint, CERN, 2000 available at http://cern.web.cern.ch/CERN/Divisions/EP/HL/publications/