Negative Stride in the Column-Major Format Makes Sense and has Useful Applications

Lars Karlsson and Carl Christian Kjelgaard Mikkelsen

Department of Computing Science, Umeå University
{larsk, spock}@cs.umu.se

Abstract

Two lower triangular or two upper triangular matrices of the same size can be stored with minimal memory footprint. If both positive and negative strides are used, then both matrices can be accessed as if they were stored in regular column-major format.

1 Introduction

We say that a matrix A of size $n \times n$ is stored in the *column-major format* if element $A(i, j)$ is mapped to offset $i + js$ relative to some pointer `A`. The integer s is referred to as the *stride*. Typically, $s = n$ to ensure that a dense matrix is mapped to a contiguous block of memory. The column-major format is an example of a *full storage format* since it can be used to store a fully dense matrix (i.e., no two distinct elements are mapped to the same offset).

A *compact storage format* exploits a structural property to store the matrix using less than n^2 memory cells. For example, an upper triangular matrix can be stored compactly by mapping $A(i, j)$ to $i + j(j + 1)/2$. This maps the matrix to a contiguous block of memory of size $n(n + 1)/2$. However, since the mapping from matrix element to memory offset is different compared to the column-major format, existing codes that access the matrix elements using the column-major mapping will be incompatible with this compact storage format.

Apart from reducing the memory footprint, compact storage formats have positive implications for the performance of communication. A triangular matrix stored in a full storage format is non-contiguous in memory. To communicate such a matrix one must either send the entire buffer, explicitly pack the matrix into a contiguous buffer, or use some non-contiguous communication functionality. All these options are presently slower than simply communicating a contiguous block of memory of minimal size. In other words, a compact storage format can improve the performance of communication.

If we have *more than one structured matrix*, then one can potentially construct ways of storing these matrices that requires less memory while at the same time allowing each matrix to be accessed as if it was stored in the column-major format. We will demonstrate this principle with an example. Let A be a *lower triangular* matrix of size $n \times n$ and let B be an *upper triangular* matrix of the same size. Embed A and B into a matrix C of size $n \times (n+1)$ as illustrated below for $n = 5$:

$$C = \begin{bmatrix} a_{00} & b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ a_{10} & a_{11} & b_{11} & b_{12} & b_{13} & b_{14} \\ a_{20} & a_{21} & a_{22} & b_{22} & b_{23} & b_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & b_{33} & b_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & b_{44} \end{bmatrix}.$$

Store C in the column-major format with pointer `C` and stride n. Element $A(i,j)$ is embedded at $C(i,j)$ and is therefore stored at

$$\texttt{C} + i + jn.$$

In other words, the lower triangle of A is accessible as if A was stored in the column-major format with pointer $\texttt{A} = \texttt{C}$ and stride n. Similarly, element $B(i,j)$ is embedded at $C(i,j+1)$ and is therefore stored at

$$\texttt{C} + i + (j+1)n = (\texttt{C} + n) + i + jn.$$

In other words, the upper triangle of B is accessible as if B was stored in the column-major format with pointer $\texttt{B} = \texttt{C} + n$ and stride n.

As this example demonstrates, reducing the memory footprint while maintaining a column-major mapping is possible. In Section 2 we show how the use of both positive and negative strides allows us to extend this result to the case where *both* matrices are lower (or upper) triangular.

2 The main idea

The lower triangular case is treated in Section 2.1 and the upper triangular case in Section 2.2. A brief comparison is made in Section 2.3.

2.1 Lower triangular matrices

Let A and B be two *lower triangular* matrices of size $n \times n$. There are at least two distinct ways to embed A and B into a dense matrix C. These options are illustrated below for

$n = 5$:

$$C_1 = \begin{bmatrix} b_{44} & b_{33} & b_{22} & b_{11} & b_{00} \\ a_{00} & b_{43} & b_{32} & b_{21} & b_{10} \\ a_{10} & a_{11} & b_{42} & b_{31} & b_{20} \\ a_{20} & a_{21} & a_{22} & b_{41} & b_{30} \\ a_{30} & a_{31} & a_{32} & a_{33} & b_{40} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, \quad C_2 = \begin{bmatrix} a_{00} & b_{44} & b_{33} & b_{22} & b_{11} & b_{00} \\ a_{10} & a_{11} & b_{43} & b_{32} & b_{21} & b_{10} \\ a_{20} & a_{21} & a_{22} & b_{42} & b_{31} & b_{20} \\ a_{30} & a_{31} & a_{32} & a_{33} & b_{41} & b_{30} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & b_{40} \end{bmatrix}.$$

The matrix C_1 is stored in column-major format with pointer `C1` and stride $n + 1$. The matrix A has pointer `A1` $=$ `C1` $+ 1$ and stride $n + 1$, so that $A(i, j)$ is stored at

$$\texttt{A1} + i + j(n + 1).$$

The matrix B has pointer `B1` $=$ `C1` $+ (n + 1)(n - 1)$ and stride $-(n + 2)$, so that $B(i, j)$ is stored at

$$\texttt{B1} + i + j(-n - 2).$$

The matrix C_2 is stored in column-major format with pointer `C2` and stride n. In this case A has pointer `A2` $=$ `C` and stride n, so that $A(i, j)$ is stored at

$$\texttt{A2} + i + jn.$$

Similarly, B has pointer `B2` $=$ `C` $+ n^2$ and stride $-(n + 1)$, so that $B(i, j)$ is stored at

$$\texttt{B2} + i + j(-n - 1).$$

At this point we have no reasons to prefer embedding C_1 over embedding C_2. After treating the upper triangular case, we will explain why we prefer embedding C_1.

2.2 Upper triangular matrices

If D and E are two upper triangular matrices of size $n \times n$, then there are at least two distinct ways to embed them into a dense matrix F. These options are illustrated below for $n = 5$:

$$F_1 = \begin{bmatrix} e_{00} & e_{01} & e_{02} & e_{03} & e_{04} \\ d_{04} & e_{11} & e_{12} & e_{13} & e_{14} \\ d_{14} & d_{03} & e_{22} & e_{23} & e_{24} \\ d_{24} & d_{13} & d_{02} & e_{33} & e_{34} \\ d_{34} & d_{23} & d_{12} & d_{01} & e_{44} \\ d_{44} & d_{33} & d_{22} & d_{11} & d_{00} \end{bmatrix}, \quad F_2 = \begin{bmatrix} d_{04} & e_{00} & e_{01} & e_{02} & e_{03} & e_{04} \\ d_{14} & d_{03} & e_{11} & e_{12} & e_{13} & e_{14} \\ d_{24} & d_{13} & d_{02} & e_{22} & e_{23} & e_{24} \\ d_{34} & d_{23} & d_{12} & d_{01} & e_{33} & e_{34} \\ d_{44} & d_{33} & d_{22} & d_{11} & d_{00} & e_{44} \end{bmatrix}.$$

The matrix F_1 is stored in column-major format with pointer `F1` and stride $n + 1$. The matrix D has pointer `D1` $=$ `F1` $+ n(n + 1) - 1$ and stride $-(n + 2)$, so that $D(i, j)$ is stored at

$$\texttt{D1} + i + j(-n - 2).$$

The matrix E has pointer $\texttt{E1} = \texttt{F1}$ and stride $n+1$, so that $E(i,j)$ is stored at

$$\texttt{E1} + i + j(n+1).$$

The matrix F_2 is stored in column-major format with stride n. In this case, the matrix D has pointer $\texttt{D2} = \texttt{F2} + n^2 - 1$ and stride $-(n+1)$, so that $D(i,j)$ is stored at

$$\texttt{D2} + i + j(-n-1).$$

Similarly, E has pointer $\texttt{E2} = \texttt{F2} + n$ and stride n, so that $E(i,j)$ is stored at

$$\texttt{E2} + i + jn.$$

2.3 Comparison of the upper and lower triangular case

At this time we have no strong reasons for preferring embedding C_1 (F_1) over C_2 (F_2). However, embeddings C_1 and F_1 have an aesthetically pleasing property, which can perhaps serve as a mnemonic device. For matrix C_1 the first and the last entry is occupied by the last entry of one of the two lower triangular matrices. For matrix F_1 the first and the last entry is occupied by the first entry of one of the two upper triangular matrices.

3 Conclusion

We have shown how to store any pair of triangular matrices of the same size with minimal memory footprint and still access the relevant portions of the matrices as if they were stored in the column-major format. The idea can be extended in various ways (e.g., to any number of matrices, to matrices of different sizes, and to the row-major format) but not always with minimal memory footprint. Applications include reducing the memory footprint and increasing the rate of communication over a network or between host and accelerator.