

## Z-ordering and UPC

Phil Merkey

Michigan Technological University

30 June 2003

### Introduction

Morton Z-ordering is a scheme to map multi-dimensional arrays onto to a linear that enjoys a great deal of spatial locality. The scheme is attributed to Morton, a geo-physicists working the mid-sixties, who wanted to maximize the utilization of the high speed drum memory he was using. The scheme never seems to gain popular support, although it has resurfaced every few years over the last 40 years.

We have employ the Z-ordering as a natural way to block distribute the affinity of a 2-dimensional array in UPC. It has also introduced the question of determining how recursive data layouts might serve in a programming model that addresses the issue of a memory hierarchy.

### Morton Z-ordering

Ultimately, computer memory is a (linear) sequence words. In order to store a multi-dimensional array or a matrix there is an implicit scheme to associate the abstract location of our data, e.g., the  $i, j, k$  grid point in 3-D grid or the  $i, j$ th entry in the matrix, to particular entry in the sequence of memory cells provided. In typical programming languages like C or Fortran, one references elements like  $a[i][j]$  or  $M(i, j)$  with little concern for pointer or index arithmetic need to find the actual address for said element.

The Morton Z-ordering for a 2-dimensional array is one-to-one correspondence between  $\{0, 1, 2, \dots, N - 1\} \times \{0, 1, 2, \dots, N - 1\}$  onto  $\{0, 1, 2, \dots, N^2 - 1\}$ , where  $N^2$  is a power of 4. The case where  $N = 8$  is given in the following example. We map the linear sequence

$$a_0, a_1, a_2, a_3, \dots, a_{62}, a_{63}$$

to the 2-dimension array

$a_0$	$a_1$	$a_4$	$a_5$	$a_{16}$	$a_{17}$	$\cdot$	$\cdot$
$a_2$	$a_3$	$a_6$	$a_7$	$a_{18}$	$a_{19}$	$\cdot$	$\cdot$
$a_8$	$a_9$	$a_{12}$	$a_{13}$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$a_{10}$	$a_{11}$	$a_{14}$	$a_{15}$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$a_{32}$	$a_{33}$	$\cdot$	$\cdot$	$a_{48}$	$\cdot$	$\cdot$	$\cdot$
$a_{34}$	$a_{35}$	$\cdot$	$\cdot$	$\cdot$	$a_{51}$	$\cdot$	$\cdot$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$a_{60}$	$a_{61}$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$a_{62}$	$a_{63}$

The name comes from the recursive decomposition of the matrix in four submatrices. The submatrices, when listed according to their order in the linear sequence form a ‘Z’ pattern in the matrix (upper left, upper right, lower left, lower right). Note the ‘Z’ pattern in the block sub-matrices. A first  $2 \times 2$  matrix forms a ‘Z’:

$$\begin{array}{cc} a_0 & a_1 \\ a_2 & a_3 \end{array}$$

Then the  $4 \times 4$  is block matrix in the form of a ‘Z’ with respect to the  $2 \times 2$  sub-matrices:

$$\begin{array}{cc|cc} a_0 & a_1 & a_4 & a_5 \\ a_2 & a_3 & a_6 & a_7 \\ \hline a_8 & a_9 & a_{12} & a_{13} \\ a_{10} & a_{11} & a_{14} & a_{15} \end{array}$$

And so forth:

$$\begin{array}{cccc|cccc} a_0 & a_1 & a_4 & a_5 & a_{16} & a_{17} & \cdot & \cdot \\ a_2 & a_3 & a_6 & a_7 & a_{18} & a_{19} & \cdot & \cdot \\ a_8 & a_9 & a_{12} & a_{13} & \cdot & \cdot & \cdot & \cdot \\ a_{10} & a_{11} & a_{14} & a_{15} & \cdot & \cdot & \cdot & a_{31} \\ \hline a_{32} & a_{33} & \cdot & \cdot & a_{48} & \cdot & \cdot & \cdot \\ a_{34} & a_{35} & \cdot & \cdot & \cdot & a_{51} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a_{60} & a_{61} \\ \cdot & \cdot & \cdot & a_{47} & \cdot & \cdot & a_{62} & a_{63} \end{array}$$

The mapping between the linear array and the 2-dimensional array is straight forward but does rely on some uncommon bit manipulations. For notation sake, when we consider an element to be an entry in the matrix,  $\mathbf{A}$ , we will denote it by  $a_{i,j}$ . The corresponding element in the linear array will be denoted by  $\mathbf{A}[\mathbf{k}]$ . Let  $i = \dots i_5 i_4 i_3 i_2 i_1 i_0$  and  $j = \dots j_5 j_4 j_3 j_2 j_1 j_0$  be the binary representation of the indices  $i$  and  $j$ . The matrix entry in position  $i, j$  maps to the  $k$  element in the linear array, where

$$k = \dots i_5 j_5 i_4 j_4 i_3 j_3 i_2 j_2 i_1 j_1 i_0 j_0.$$

The map from the 2-dimensional Z-order to the 1-dimensional sequence is clear a bijection. It is also clear that one could inter-leave the bits of  $i$ ,  $j$ , and  $k$  to obtain a map from a 3-dimensional array to 1-dimensional sequence. The recursive nature of the block structure of the matrix is the key to the relationship between the Z-ordering and quad-trees (or oct-trees in the 3-D case). Both of these topics are

beyond the scope of this note. Interested readers should refer to work on databases for geo-spatial data [2], recursive array layouts [1], or data structures for out-of-core algorithms [7,5].

### Implementation Issues

This section is concerned with developing techniques to manipulate these indices efficiently on modern architectures. This discussion is restricted to the 2-dimensional case, but the ideas would certainly generalize.

The fact that the bits of the  $j$ th column and  $i$ th row indices being stored in  $\mathbf{k}$ 's even and odd bits, respectively, gives rise to the notion of a *dilated integer*. If  $x = \sum_s x_s 2^s$  is a positive integer, then the (even) dilated integer for  $x$  is  $\overrightarrow{x} = \sum_s x_s 4^s$  and the (odd) dilated integer for  $x$  is  $\overleftarrow{x} = \sum_s x_s 2 \cdot 4^s$ .

The choice of functions and macros to form and manipulate dilated integer depends on the following list of properties. The following is taken from the Techreport by Wise [8]. We have adopted their use of left and right arrows to indicate odd and even dilation.

Let  $\mathbf{Jmask} = \overrightarrow{N-1}$  represent the mask that covers the even bits. In the 32-bit case,  $N = 2^{16}$ ,  $\mathbf{Jmask} = \overrightarrow{N-1} = 0x55555555$ . Likewise,  $\mathbf{Imask} = \overleftarrow{N-1} = 0xA5A5A5A5$  is the mask that covers the odd bits.

**Prop:** Dilation is strictly monotonic. That is, for any non-negative integers  $i$  and  $j$ ,

$$(\overrightarrow{i} == \overrightarrow{j}) \quad \text{iff} \quad (i == j) \quad \text{iff} \quad (\overleftarrow{i} == \overleftarrow{j})$$

and

$$(\overrightarrow{i} < \overrightarrow{j}) \quad \text{iff} \quad (i < j) \quad \text{iff} \quad (\overleftarrow{i} < \overleftarrow{j}).$$

**Prop:** The map from the 2-dimensional array  $a_{i,j}$  to the corresponding element in the 1-dimensional array  $\mathbf{A}[\mathbf{k}]$  is simply  $a_{i,j}$  maps to  $\mathbf{A}[\overleftarrow{i} + \overrightarrow{j}]$  or, equivalently,  $\mathbf{A}[\overleftarrow{i} \vee \overrightarrow{j}]$ .

**Prop:** Common index manipulation: If  $i > j$ , then

$$\begin{aligned} \overrightarrow{i-j} &= (\overrightarrow{i} - \overrightarrow{j}) \& \overrightarrow{N-1}; \\ \overleftarrow{i-j} &= (\overleftarrow{i} - \overleftarrow{j}) \& \overleftarrow{N-1}; \end{aligned}$$

and for any  $i$  and  $j$ ,

$$\begin{aligned}\overrightarrow{i+j} &= (\overrightarrow{i} + \overrightarrow{j} + \overleftarrow{N-1}) \& \overleftarrow{N-1}; \\ \overleftarrow{i+j} &= (\overleftarrow{i} + \overleftarrow{j} + \overrightarrow{N-1}) \& \overrightarrow{N-1};\end{aligned}$$

in particular,

$$\begin{aligned}\overrightarrow{i++} &= (\overrightarrow{i} + 0xAAAAAAAA) \& \overleftarrow{N-1}; \\ \overleftarrow{i++} &= (\overleftarrow{i} + 0x55555557) \& \overrightarrow{N-1};\end{aligned}$$

and,

$$\begin{aligned}\overrightarrow{i<<j} &= \overrightarrow{i} \ll (2j); \\ \overleftarrow{i<<j} &= \overleftarrow{i} \ll (2j).\end{aligned}$$

On most architectures converting between the standard and dilated forms of an integer requires a sequence of shifts, masking and OR operations.

The following function dilates a 16-bit integer into a 32-bit word.

```
//Xpand: Computes the (even) dilation of a int
int Xpand( int t )
{
    int u,v,w;
    u = ((t&0x0000FF00)<<8) | (t & 0x000000FF);
    v = ((u&0x00F000F0)<<4) | (u & 0x000F000F);
    w = ((v&0x0C0C0C0C)<<2) | (v & 0x03030303);
    return(((w&0x22222222)<<1) | (w & 0x11111111));
}
```

To verify that this subroutine implements the expansion or dilation of a positive integer observe that if consider the binary expansion of  $t$ , we have

$$t = 0000000000000000t_{15}t_{14}t_{13}t_{12}t_{11}t_{10}t_9t_8t_7t_6t_5t_4t_3t_2t_1t_0.$$

Then following the computation we have

$$\begin{aligned}u &= 00000000t_{15}t_{14}t_{13}t_{12}t_{11}t_{10}t_9t_800000000t_7t_6t_5t_4t_3t_2t_1t_0, \\ v &= 0000t_{15}t_{14}t_{13}t_{12}0000t_{11}t_{10}t_9t_80000t_7t_6t_5t_40000t_3t_2t_1t_0, \\ w &= 00t_{15}t_{14}00t_{13}t_{12}00t_{11}t_{10}00t_9t_800t_7t_600t_5t_400t_3t_200t_1t_0,\end{aligned}$$

and the subroutine returns

$$0t_{15}0t_{14}0t_{13}0t_{12}0t_{11}0t_{10}0t_90t_80t_70t_60t_50t_40t_30t_20t_10t_0.$$

Note that we could dilate a 32-bit number into the even bits of a 64-bit number by providing 64-bit versions of the masking constants and adding one more line of code.

The complementary function to extract an (even) dilated integer from a given integer, ignoring the odd bits entirely is given by

```
// Xtract: Xtract the even bits and
//          compress them to a standard int
int Xtract( int t )
{
    int u,v,w,x;
    u = (t&0x55555555);
    v = ((u&0x44444444)>>1) | (u & 0x11111111);
    w = ((v&0x30303030)>>2) | (v & 0x03030303);
    x = ((w&0x0F00F000)>>4) | (w & 0x000F000F);
    return( ((x&0x00FF0000)>>8) | (x & 0x000000FF) );
}
```

The basic set of macros we need to convert integers to and from their dilated formats and to run basic loops are

```
#define Imask 0xAAAAAAAA
#define Jmask 0x55555555
#define Zord(i,j) ((Xpand((i))<<1) | Xpand((j)))
#define ixtrct(K) (Xtract( ((K))>>1 ))
#define jxtrct(K) (Xtract( ((K)) ))
#define Ixpnd(k) (Xpand((k))<<1)
#define Jxpnd(k) (Xpand((k))
#define Iinc(I) ( ((I)+0x55555557) & Imask )
#define Jinc(J) ( ((J)+0xAAAAAAAAAB) & Jmask )
```

If we are working on a grid and need access to the grid points surrounding the current point, the follow macros are useful.

```
#define Nth(k) (((k)&Jmask) | (((k)&Imask)-2) & Imask))
#define Sth(k) (((k)&Jmask) | (((k)|Jmask)+2) & Imask))
#define Est(k) (((k)&Imask) | (((k)|Imask)+1) & Jmask))
#define Wst(k) (((k)&Imask) | (((k)&Jmask)-1) & Jmask))
```

It might appear that using Z-ordering would be expensive because of the functions Xpand and Xtract. The 32-bit versions of Xpand and Xtract require 16 and 17

operations, and the 64-bit version would require 20 and 21 operations, respectively. However, there are a number of reasons why the impact of the operations should be nominal.

First and foremost, with the difference between arithmetic operations and memory operations approaching two order of magnitude, trading lot of ALU operations to improve locality and memory efficiency makes sense. Also note that the functions are straight line code and could be inlined thus avoiding the function call overhead. In addition, the code enjoys lots of ILP which should be exploited by superscaler architectures. Furthermore, we will argue below that we can often avoid doing the expansion and extraction by working with the dilated integers directly.

We also observe that on the Cray MTA, these subroutines would be replaced with the assemble language instruction "extract and pack" and on the Cray X1 they would be replaced with macros that use two bit-matrix-multiply operations and a mask. Consider a 64-bit word as if it were an  $8 \times 8$  bit matrix

$$\mathbf{t} = \begin{bmatrix} t_{63} & t_{62} & t_{61} & t_{60} & t_{59} & t_{58} & t_{57} & t_{56} \\ t_{55} & t_{54} & t_{53} & t_{52} & t_{51} & t_{50} & t_{49} & t_{48} \\ t_{47} & t_{46} & t_{45} & t_{44} & t_{43} & t_{42} & t_{41} & t_{40} \\ t_{39} & t_{38} & t_{37} & t_{36} & t_{35} & t_{34} & t_{33} & t_{32} \\ t_{31} & t_{30} & t_{29} & t_{28} & t_{27} & t_{26} & t_{25} & t_{24} \\ t_{23} & t_{22} & t_{21} & t_{20} & t_{19} & t_{18} & t_{17} & t_{16} \\ t_{15} & t_{14} & t_{13} & t_{12} & t_{11} & t_{10} & t_9 & t_8 \\ t_7 & t_6 & t_5 & t_4 & t_3 & t_2 & t_1 & t_0 \end{bmatrix}.$$

The bit-matrix-multiply operation is a matrix multiply on these  $8 \times 8$  matrices where the arithmetic is performed in  $GF(2)$ . The following combination of bit-matrix multiplies and a masking operation yields the dilation of a 32-bit word.

Set

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_{31} & t_{30} & t_{29} & t_{28} & t_{27} & t_{26} & t_{25} & t_{24} \\ t_{23} & t_{22} & t_{21} & t_{20} & t_{19} & t_{18} & t_{17} & t_{16} \\ t_{15} & t_{14} & t_{13} & t_{12} & t_{11} & t_{10} & t_9 & t_8 \\ t_7 & t_6 & t_5 & t_4 & t_3 & t_2 & t_1 & t_0 \end{bmatrix} = \mathbf{u}.$$

Then set

$$\text{FOOFFOOFFOOFFOOF} \ \& \ \mathbf{u} = \begin{bmatrix} t_{31} & t_{30} & t_{29} & t_{28} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & t_{27} & t_{26} & t_{25} & t_{24} \\ t_{23} & t_{22} & t_{21} & t_{20} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & t_{19} & t_{18} & t_{17} & t_{16} \\ t_{15} & t_{14} & t_{13} & t_{12} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & t_{11} & t_{10} & t_9 & t_8 \\ t_7 & t_6 & t_5 & t_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & t_3 & t_2 & t_1 & t_0 \end{bmatrix} = \mathbf{v}$$

and

$$\mathbf{v} \cdot \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & t_{31} & 0 & t_{30} & 0 & t_{29} & 0 & t_{28} \\ 0 & t_{27} & 0 & t_{26} & 0 & t_{25} & 0 & t_{24} \\ 0 & t_{23} & 0 & t_{22} & 0 & t_{21} & 0 & t_{20} \\ 0 & t_{19} & 0 & t_{18} & 0 & t_{17} & 0 & t_{16} \\ 0 & t_{15} & 0 & t_{14} & 0 & t_{13} & 0 & t_{12} \\ 0 & t_{11} & 0 & t_{10} & 0 & t_9 & 0 & t_8 \\ 0 & t_7 & 0 & t_6 & 0 & t_5 & 0 & t_4 \\ 0 & t_3 & 0 & t_2 & 0 & t_1 & 0 & t_0 \end{bmatrix}$$

as required. Thus, if the bit-matrix-multiply is called like a function, say  $\text{BMM}(\mathbf{x}, \mathbf{y})$ , then

$$\text{Jxpcnd}(\mathbf{t}) = \text{BMM}(\text{FOOFFOOFFOOFFOOF} \ \& \ \text{BMM}(\text{0808040402020101}, \mathbf{t}), \\ \text{4010040140100401}).$$

Similarly,

$$\text{Ixpcnd}(\mathbf{t}) = \text{BMM}(\text{FOOFFOOFFOOFFOOF} \ \& \ \text{BMM}(\text{0808040402020101}, \mathbf{t}), \\ \text{8020080280200802}).$$

Taking appropriate inverses yields the extract operations.

$$\text{jxtrct}(\mathbf{T}) = \text{BMM}(\text{00000000C030C03}, \\ \text{FOOFFOOFFOOFFOOF} \ \& \ \text{BMM}(\mathbf{T}, \text{0088004400220011})).$$

and

$$\text{ixtrct}(\mathbf{T}) = \text{BMM}(\text{00000000C030C03}, \\ \text{FOOFFOOFFOOFFOOF} \ \& \ \text{BMM}(\mathbf{T}, \text{8800440022001100})).$$

## Examples

The first example is the naive version of matrix multiply. In standard C, one would expect to see matrix multiply described by the following code fragments.

```
int A[N][N], B[N][N], C[N][N];
.....
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    C[i][j] = 0;
    for(k=0; k<N; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

If we assume that each of the matrices are Z-ordered and we wish to arrange the computation by rows and then columns, as in the original loop, then the code fragments become:

```
int A[N*N], B[N*N], C[N*N];
int IdN, JdN; // dilated forms of loop limits
.....
IdN = Ixpan(N);
JdN = Jxpan(N);
for( I=0; I<IdN; I = Iinc(I) ){
  for( J=0; J<JdN; J = Jinc(J) ){
    C[ I+J ] = 0;
    for(K=0; K<JdN; K = Jinc(K) ){
      C[ I+J ] += A[ I+K ] * B[ (K<<1) + J ];
    }
  }
}
```

This shows that we can mimic codes that are row or column oriented without significant overhead.

We can update the C matrix in any order we wish. In particular, using the

Z-order allows for an interesting trick.

```

int A[N*N], B[N*N], C[N*N];
int IdN, JdN; // dilated forms of loop limits
.....
IdN = Ixpond(N);
JdN = Jxpond(N);
for( z=0; z<N*N; z++ ){
    I = z & Imask;
    J = z & Jmask;
    C[ z ] = 0;
    for(K=0; K<JdN; K = Jinc(K) ){
        C[ z ] += A[ I+K ] * B[ (K<<1) + J ];
    }
}
}

```

More importantly, we can arrange the loops to increase locality.

```

int A[N*N], B[N*N], C[N*N];
int IdN, JdN; // dilated forms of loop limits
.....
IdN = Ixpond(N);
JdN = Jxpond(N);
memset(C,0,N*N);
for(zblk=0; zblk<N*N; zblk += BLKSIZE){
    for(K=0; K<JdN; K = Jinc(K) ){
        for( z0=0; z<BLKSIZE; z++ ){
            z = zblk + z0;
            I = z & Imask;
            J = z & Jmask;
            C[ z ] += A[ I+K ] * B[ (K<<1) + J ];
        }
    }
}
}

```

The second example is actually the motivation for looking at Z-ordering for UPC. Many scientific applications can be characterized as “grid” based application. Consider solving the heat equation on a 2-D grid Let  $u[i][j]$  be the temperature

at the point  $i, j$  grid point. The defining code fragments for this application are:

```

float u[N][N], v[N][N];
....
// Put the boundary condition
//   in the 0th and (N-1)st rows and columns
....
while( timeloop ){
  for(i=1; i<N-1; i++){
    for(j=1; j<N-1; j++){
      v[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1])/4.0
    }
    for(i=1; i<N-1; i++){
      for(j=1; j<N-1; j++){
        u[i][j] = v[i][j];
      }
    }
    ....
    // Compute stop condition
    ....
  }
}

```

If we store the grid points using the Z-ordering and update the values in the Z-order we naturally exploit the spatial locality. Note that this code uses the `North`, `South`, `East` and `West` macros defined above, which hide a little bit of overhead. We also need to check that we don't over-write the boundary conditions that are stored along the outside of the grid.

```

#define EDGE(K) ( (K&Imask)==0 || (K&Jmask)==0 \
                 || (K&Imask)==IdN1 || (K&Jmask)==JdN1 )

```

where `IdN1` and `JdN1` are the I and J dilated versions of `N-1`.

```

float u[N*N], v[N*N];
....
// Put the boundary condition
//   in the 0th and (N-1)st rows and columns
....
while( timeloop ){
  for(k=1; k<N*N; k++){
    if( EDGE(k) )
      continue;
    else
      v[k] = (u[Nth(k)] + u[Sth(k)] + u[Eth(k)] + u[Wth(k)])/4.0
  }
  for(k=1; k<N*N; k++){
    u[k] = v[k];
  }
  ....
  // Compute stop condition
  ....
}

```

The UPC code for this requires almost no modification. With `THREADS` equal to a power of 4, we simply block the array so that each thread has affinity to  $N*N/THREADS$  elements and obvious loop minimizes the number of non-affinity references.

```

shared [N*N/THREADS] float u[N*N], v[N*N];
....
// Put the boundary condition
//   in the 0th and (N-1)st rows and columns
....
while( timeloop ){
  upc\_forall( k=1; k<N*N; k++; \&u[k] ){
    if( EDGE(k) )
      continue;
    else
      v[k] = (u[Nth(k)] + u[Sth(k)] + u[Eth(k)] + u[Wth(k)])/4.0
  }
  upc_barrier;
  upc\_for( k=1; k<N*N; k++; \&u[k] ){
    u[k] = v[k];
  }
  upc_barrier;
  ....
  // Compute stop condition
  ....
}

```

## References

- [1] S. Chatterjee, A. Lebeck and P. Patnala, "Recursive Array Layouts and Fast Matrix Multiplication," Submitted to IEEE TPDS.
- [2] G. Hjaltason, H. Samet, "Improved Bulk-Loading Algorithms for Quadtrees," Proc of 7th Sym on GIS (ACM GIS' 99), pp 110-115.
- [3] G. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Tech Report, IBM, Ottawa, Canada, 1966.
- [4] P. Oosterom and T. Vrijlbrief, "The Spatial Location Code," Presented at SDH'96 (7th International Symposium on Spatial Data Handling).
- [5] J. Salmon and M. Warren, "Parallel, out-of-core methods for N-body simulation," Proc. 8th SIAM Conf on Parallel Processing for Scientific Computing, 1997
- [6] J. Thiyagalingam and P. Kelly, "Is Morton layout competitive for large two-dimensional arrays?" EuroPar 2000.
- [7] T. Tu, D. Hallaron and J. Lopez, "ETREE—A Database-Oriented Method for Generating Large Octree Meshes,"
- [8] David Wise, Jeremy Frens, "Morton-order Matrices Deserve Compilers' Support," Indiana Univ, Technical Report 533, 1999.