# *Cache-Oblivious Algorithms*

by
Harald Prokop

*Submitted to the*
*Department of Electrical Engineering and Computer Science*
*on May 21, 1999 in partial fulfillment of the*
*requirements for the degree of Master of Science.*

## *Abstract*

This thesis presents "cache-oblivious" algorithms that use asymptotically optimal amounts of work, and move data asymptotically optimally among multiple levels of cache. An algorithm is ***cache oblivious*** if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses.

We show that the ordinary algorithms for matrix transposition, matrix multiplication, sorting, and Jacobi-style multipass filtering are not cache optimal. We present algorithms for rectangular matrix transposition, FFT, sorting, and multipass filters, which are asymptotically optimal on computers with multiple levels of caches. For a cache with size $Z$ and cache-line length $L$, where $Z = \Omega(L^2)$, the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/L)$. The number of cache misses for either an $n$-point FFT or the sorting of $n$ numbers is $\Theta(1 + (n/L)(1 + \log_Z n))$. The cache complexity of computing $n$ time steps of a Jacobi-style multipass filter on an array of size $n$ is $\Theta(1 + n/L + n^2/ZL)$. We also give an $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses.

We introduce an "ideal-cache" model to analyze our algorithms, and we prove that an optimal cache-oblivious algorithm designed for two levels of memory is also optimal for multiple levels. We further prove that any optimal cache-oblivious algorithm is also optimal in the previously studied HMM and SUMH models. Algorithms developed for these earlier models are perforce ***cache-aware***: their behavior varies as a function of hardware-dependent parameters which must be tuned to attain optimality. Our cache-oblivious algorithms achieve the same asymptotic optimality on all these models, but without any tuning.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

## SECTION 1

# *Introduction*

Resource-oblivious algorithms that nevertheless use resources efficiently offer advantages of simplicity and portability over resource-aware algorithms whose resource usage must be programmed explicitly. In this thesis, we study cache resources, specifically, the hierarchy of memories in modern computers. We exhibit several "cache-oblivious" algorithms that use cache as effectively as "cache-aware" algorithms.

Before discussing the notion of cache obliviousness, we introduce the $(\boldsymbol{Z}, \boldsymbol{L})$ ***ideal-cache model*** to study the cache complexity of algorithms. This model, which is illustrated in Figure 1-1, consists of a computer with a two-level memory hierarchy consisting of an ideal (data) cache of $Z$ words and an arbitrarily large main memory. Because the actual size of words in a computer is typically a small, fixed size (4 bytes, 8 bytes, etc.), we shall assume that word size is constant; the particular constant does not affect our asymptotic analyses. The cache is partitioned into ***cache lines***, each consisting of $L$ consecutive words that are always moved together between cache and main memory. Cache designers typically use $L > 1$, banking on spatial locality to amortize the overhead of moving the cache line. We shall generally assume in this thesis that the cache is ***tall***:

$$Z = \Omega(L^2) , \tag{1.1}$$

which is usually true in practice.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a ***cache hit*** occurs, and the word is

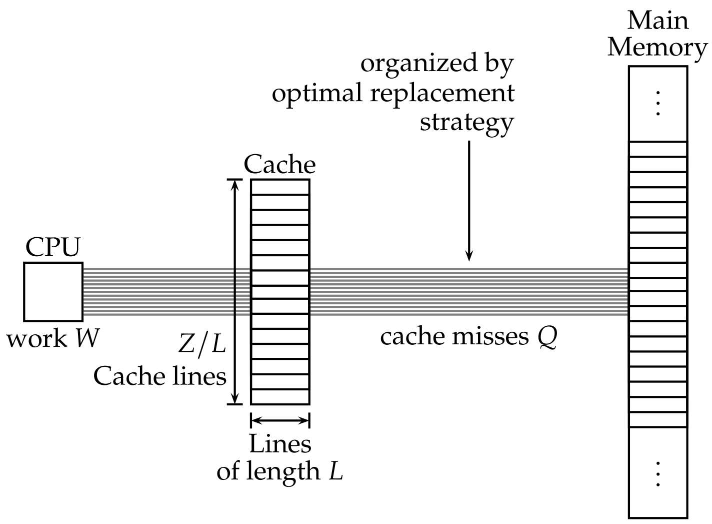



**Figure 1-1:** The ideal-cache model

delivered to the processor. Otherwise, a ***cache miss*** occurs, and the line is fetched into the cache. The ideal cache is ***fully associative*** [24, Ch. 5]: Cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line strategy of replacing the cache line whose next access is farthest in the future [7], and thus it exploits temporal locality perfectly.

An algorithm with an input of size $n$ is measured in the ideal-cache model in terms of its ***work complexity*** $W(n)$—its conventional running time in a RAM model [4]—and its ***cache complexity*** $Q(n; Z, L)$—the number of cache misses it incurs as a function of the size $Z$ and line length $L$ of the ideal cache. When $Z$ and $L$ are clear from context, we denote the cache complexity as simply $Q(n)$ to ease notation.

We define an algorithm to be ***cache aware*** if it contains parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line length. Otherwise, the algorithm is ***cache oblivious***. Historically, good performance has been obtained using cache-aware algorithms, but we shall exhibit several cache-oblivious algorithms for fundamental problems that are asymptotically as efficient as their cache-aware counterparts.

To illustrate the notion of cache awareness, consider the problem of multiplying two $n \times n$ matrices $A$ and $B$ to produce their $n \times n$ product $C$. We assume that the three matrices are stored in row-major order, as shown in Figure 2-1(a). We further assume that $n$ is "big," i.e., $n > L$, in order to simplify the analysis. The conventional way to multiply matrices on a computer with caches is to use a ***blocked*** algorithm [22, p. 45]. The idea is to view each matrix $M$ as consisting of $(n/s) \times (n/s)$ submatrices $M_{ij}$ (the blocks), each of which has size $s \times s$, where $s$ is a tuning parameter. The following algorithm implements this strategy:

BLOCK-MULT$(A, B, C, n)$
**for** $i \leftarrow 1$ **to** $n/s$
  **do for** $j \leftarrow 1$ **to** $n/s$
    **do for** $k \leftarrow 1$ **to** $n/s$
      **do** ORD-MULT$(A_{ik}, B_{kj}, C_{ij}, s)$

where ORD-MULT$(A, B, C, s)$ is a subroutine that computes $C \leftarrow C + AB$ on $s \times s$ matrices using the ordinary $O(s^3)$ algorithm (see Section 7.1). (This algorithm assumes for simplicity that $s$ evenly divides $n$. In practice, $s$ and $n$ need have no special relationship, which yields more complicated code in the same spirit.)

Depending on the cache size of the machine on which BLOCK-MULT is run, the parameter $s$ can be tuned to make the algorithm run fast, and thus BLOCK-MULT is a cache-aware algorithm. To minimize the cache complexity, we choose $s$ as large as possible such that the three $s \times s$ submatrices simultaneously fit in cache. An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines. From the tall-cache assumption (1.1), we can see that $s = \Theta(\sqrt{Z})$. Thus, each of the calls to ORD-MULT runs with at most $Z/L = \Theta(s + s^2/L)$ cache misses needed to bring the three matrices into the cache. Consequently, the cache complexity of the entire algorithm is $\Theta(n + n^2/L + (n/\sqrt{Z})^3(Z/L)) = \Theta(n + n^2/L + n^3/L\sqrt{Z})$, since the algorithm must read $n^2$ elements, which reside on $\lceil n^2/L \rceil$ cache lines.

The same bound can be achieved using a simple cache-oblivious algorithm that requires no tuning parameters such as the $s$ in BLOCK-MULT. We present such an algorithm, which works on general rectangular matrices, in Section 2. The problems of computing a matrix transpose and of performing an FFT also succumb to remarkably simple algorithms, which are described in Section 3. Cache-oblivious sorting poses a more formidable challenge. In Sections 4 and 5, we present two sorting algorithms, one based on mergesort and the other on distribution sort, both of which are optimal. Section 6 compares an optimal recursive algorithm with an "ordinary" iterative algorithm, both of which compute a multipass filter over one-dimensional data. It also provides some brief empirical results for this problem. In Section 7, we show that the ordinary algorithms for matrix transposition, matrix multiplication, and sorting are not cache optimal.

The ideal-cache model makes the perhaps-questionable assumption that memory is managed automatically by an optimal cache replacement strategy. Although the current trend in architecture does favor automatic caching over programmer-specified data movement, Section 8 addresses this concern theoretically. We show that the assumptions of two hierarchical memory models in the literature, in which memory movement is programmed explicitly, are actually no weaker than ours. Specifically, we prove (with only minor assumptions) that optimal cache-oblivious algorithms in the ideal-cache model are also optimal in the hierarchical memory model (HMM) [1] and in the serial uniform memory hierarchy (SUMH) model [5, 42]. Section 9 discusses related work, and Section 10 offers some concluding remarks.

Many of the results in this thesis are based on a joint paper [21] coauthored by Matteo Frigo, Charles E. Leiserson, and Sridhar Ramachandran.

## SECTION 2

# *Matrix multiplication*

This section describes and analyzes an algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using $\Theta(mnp)$ work and incurring $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses. These results require the tall-cache assumption (1.1) for matrices stored in row-major layout format, but the assumption can be relaxed for certain other layouts. We also show that Strassen's algorithm [38] for multiplying $n \times n$ matrices, which uses $\Theta(n^{\log_2 7})$ work, incurs $\Theta(1 + n^2/L + n^{\log_2 7}/L\sqrt{Z})$ cache misses.

The following algorithm extends the optimal divide-and-conquer algorithm for square matrices described in [9] to rectangular matrices. To multiply an $m \times n$ matrix $A$ by an $n \times p$ matrix $B$, the algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$AB = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{2.1}$$

$$AB = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2.2}$$

$$AB = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{2.3}$$

In case (2.1), we have $m \geq \max\{n, p\}$. Matrix $A$ is split horizontally, and both halves are multiplied by matrix $B$. In case (2.2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied. In case (2.3), we have $p \geq \max\{m, n\}$. Matrix $B$ is split vertically, and each half is multiplied by $A$. For square matrices, these three cases together are equivalent to the recursive multiplication

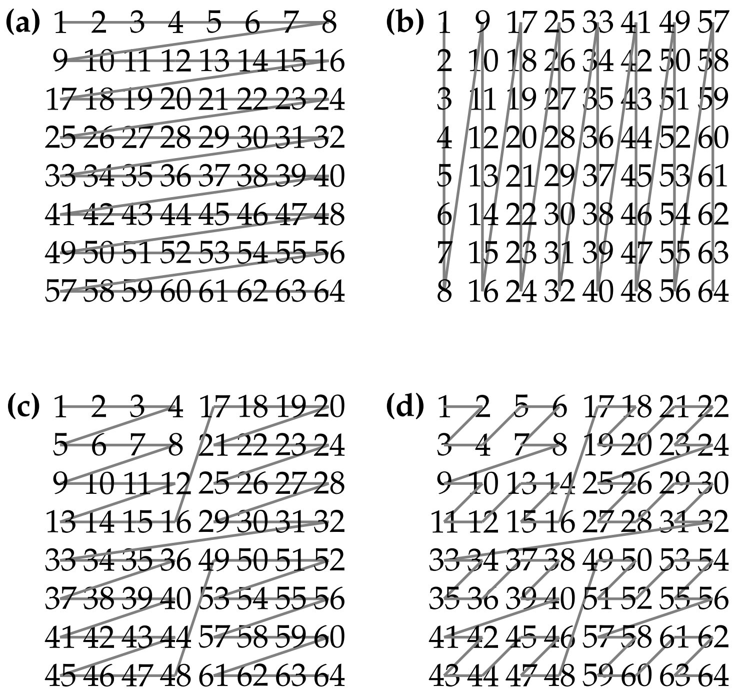



**Figure 2-1:** Layout of a $16 \times 16$ matrix in **(a)** row major, **(b)** column major, **(c)** $4 \times 4$-blocked, and **(d)** bit-interleaved layouts.

algorithm described in [9]. The base case occurs when $m = n = p = 1$, in which case the two elements are multiplied and added into the result matrix.

Although this straightforward divide-and-conquer algorithm contains no tuning parameters, it uses cache optimally. To analyze the algorithm, we assume that the three matrices are stored in row-major order, as shown in Figure 2-1(a). Intuitively, the cache-oblivious divide-and-conquer algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.

**Theorem 1** *The cache-oblivious matrix multiplication algorithm uses* $\Theta(mnp)$ *work and incurs* $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ *cache misses when multiplying an* $m \times n$ *by an* $n \times p$ *matrix.*

*Proof.* It can be shown by induction that the work of this algorithm is $\Theta(mnp)$. To analyze the cache misses, let $\alpha$ be a constant sufficiently small that three submatrices of size $m' \times n'$, $n' \times p'$, and $m' \times p'$, where $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$, all fit completely in the cache. We distinguish the following four cases cases depending on the initial size of the matrices.

**Case I:** $m, n, p > \alpha\sqrt{Z}$.

This case is the most intuitive. The matrices do not fit in cache, since all dimensions are "big enough." The cache complexity of matrix multiplication can be described by the recurrence

$$Q(m,n,p) \leq \begin{cases} \Theta((mn+np+mp)/L) & \text{if } (mn+np+mp) \leq \alpha Z \,, \\ 2Q(m/2,n,p)+O(1) & \text{otherwise and if } m \geq n \text{ and } m \geq p \,, \\ 2Q(m,n/2,p)+O(1) & \text{otherwise and if } n > m \text{ and } n \geq p \,, \\ 2Q(m,n,p/2)+O(1) & \text{otherwise} \,. \end{cases} \tag{2.4}$$

The base case arises as soon as all three submatrices fit in cache. The total number of lines used by the three submatrices is $\Theta((mn+np+mp)/L)$. The only cache misses that occur during the remainder of the recursion are the $\Theta((mn+np+mp)/L)$ cache misses required to bring the matrices into cache. In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, which depend on the dimensions of the matrices, plus $O(1)$ cache misses for the overhead of manipulating submatrices. The solution to this recurrence is $Q(m,n,p) = \Theta(mnp/L\sqrt{Z})$.

**Case II:** ($m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$) OR ($m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$) OR ($p \leq \alpha\sqrt{Z}$ and $m, n > \alpha\sqrt{Z}$).

Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$. The proofs for the other cases are only small variations of this proof. The multiplication algorithm always divides $n$ or $p$ by 2 according to cases (2.2) and (2.3). At some point in the recursion, both are small enough that the whole problem fits into cache. The number of cache misses can be described by the recurrence

$$Q(m,n,p) \leq \begin{cases} \Theta(1+n+np/L+m) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \,, \\ 2Q(m,n/2,p)+O(1) & \text{otherwise and if } n \geq p \,, \\ 2Q(m,n,p/2)+O(1) & \text{otherwise} \,. \end{cases}$$

The solution to this recurrence is $\Theta(np/L + mnp/L\sqrt{Z})$.

**Case III:** ($n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$) OR ($m, p \leq \alpha\sqrt{Z}$ and $n > \alpha\sqrt{Z}$) OR ($m, n \leq \alpha\sqrt{Z}$ and $p > \alpha\sqrt{Z}$).

In each of these cases, one of the matrices fits into cache, and the others do not. Here, we shall present the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$. The other cases can be proven similarly. The multiplication algorithm always divides $m$ by 2 according to case (2.1). At some point in the recursion, $m$ is in the range $\alpha\sqrt{Z}/2 \le m \le \alpha\sqrt{Z}$, and the whole problem fits in cache. The number cache misses can be described by the recurrence

$$Q(m,n) \le \begin{cases} \Theta(1+m) & \text{if } m \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p) + O(1) & \text{otherwise} \ ; \end{cases}$$

whose solution is $Q(m,n,p) = \Theta(m + mnp/L\sqrt{Z})$.

**Case IV:** $m, n, p \le \alpha\sqrt{Z}$.

From the choice of $\alpha$, all three matrices fit into cache. The matrices are stored on $\Theta(1 + mn/L + np/L + mp/L)$ cache lines. Therefore, we have $Q(m,n,p) = \Theta(1 + (mn + np + mp)/L)$. □

We require the tall-cache assumption (1.1) in these analyses, because the matrices are stored in row-major order. Tall caches are also needed if matrices are stored in column-major order (Figure 2-1(b)), but the assumption that $Z = \Omega(L^2)$ can be relaxed for certain other matrix layouts. The $s \times s$-blocked layout (Figure 2-1(c)), for some tuning parameter $s$, can be used to achieve the same bounds with the weaker assumption that the cache holds at least some sufficiently large constant number of lines. The cache-oblivious bit-interleaved layout (Figure 2-1(d)) has the same advantage as the blocked layout, but no tuning parameter need be set, since submatrices of size $\Theta(\sqrt{L} \times \sqrt{L})$ are cache-obliviously stored on one cache line. The advantages of bit-interleaved and related layouts have been studied in [18] and [12, 13]. One of the practical disadvantages of bit-interleaved layouts is that index calculations on conventional microprocessors can be costly.

For square matrices, the cache complexity $Q(n) = \Theta(n + n^2/L + n^3/L\sqrt{Z})$ of the cache-oblivious matrix multiplication algorithm is the same as the cache complexity of the cache-aware BLOCK-MULT algorithm and also matches the lower bound by Hong and Kung [25]. This lower bound holds for all algorithms that execute the $\Theta(n^3)$ operations given by the definition of matrix multiplication

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ .$$

No tight lower bounds for the general problem of matrix multiplication are known.

By using an asymptotically faster algorithm, such as Strassen's algorithm [38] or one of its variants [45], both the work and cache complexity can be reduced. When multiplying $n \times n$ matrices, Strassen's algorithm, which is cache oblivious, requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and a constant number of matrix additions, yielding the recurrence

$$Q(n) \leq \begin{cases} \Theta(1 + n + n^2/L) & \text{if } n^2 \leq \alpha Z \text{ ,} \\ 7Q(n/2) + O(n^2/L) & \text{otherwise ;} \end{cases} \tag{2.5}$$

where $\alpha$ is a sufficiently small constant. The solution to this recurrence is $\Theta(n + n^2/L + n^{\log_2 7}/L\sqrt{Z})$.

## *Summary*

In this section we have used the ideal-cache model to analyze two algorithms for matrix multiplication. We have described an efficient cache-oblivious algorithm for rectangular matrix multiplication and analyzed the cache complexity of Strassen's algorithm.

## 7.1 *Matrix multiplication*

In this section, we analyze the straightforward iterative algorithm for matrix multiplication. We prove that it causes $\Omega(n^3)$ cache misses when the $n \times n$ matrices are stored in row-major order and do not fit in cache. We further show that even if the matrices are stored in the order in which they are used and do not fit in cache, the number of cache misses is at least $\Omega(n^3/L)$, compared to $\Theta(n^3/L\sqrt{Z})$ for an the cache-optimal $\Theta(n^3)$-work algorithm presented in Section 2.

The simplest way to compute the product of two matrices is to evaluate the formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

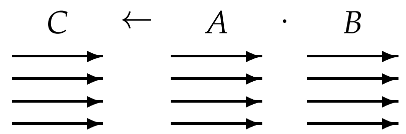
directly, as in the following program:

ORD-MULT$(A, B, C, n)$
**for** $i \leftarrow 1$ **to** $n$
  **do** **for** $j \leftarrow 1$ **to** $n$
    **do** $c_{ij} \leftarrow 0$
      **for** $k \leftarrow 1$ **to** $n$
        **do** $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$

**Theorem 20** *The* ORD-MULT *algorithm for matrix multiplication uses $\Theta(n^3)$ work when multiplying $n \times n$ matrices that do not fit in cache. It incurs $\Omega(n^3)$ cache misses, when the matrices are stored in row-major order. Even if the matrices are stored in the order in which they are used,* ORD-MULT *incurs $\Omega(n^3/L)$ cache misses, which is a factor of $\Theta(\sqrt{Z})$ from optimal.*

*Proof.* Analyzing the work of ORD-MULT$(A, B)$ is straightforward. Since there are three nested loops, each of which performs $n$ iterations, the work is $\Theta(n^3)$. Since the algorithm cannot access more than $O(1)$ elements in constant time, $O(n^3)$ is also an upper bound on the number of cache misses for this algorithm.

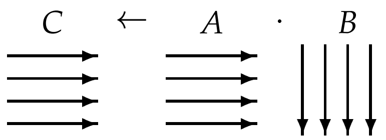
First, we assume that both matrices are stored in row-major order (Figure 2-1(a)):




The number of cache misses can be lower-bounded by counting only the misses caused by reading matrix $B$. In the $i$th iteration of the outer loop, the elements of the $i$th row of $C$ are computed. While $i$ is fixed the inner two loops iterate over all $n^2$ values of $k$ and $j$, reading all elements of matrix $B$ column by column.

Assuming that the matrix does not fit in cache, e.g. $n \gg Z/L$, the LRU replacement strategy overwrites lines of matrix $B$ before they can be reused. Therefore, the number of misses on matrix $B$ to compute one element of matrix $C$ is $\Omega(n)$. Since $C$ contains $n^2$ elements, the algorithm causes $\Omega(n^3)$ cache misses. It follows from Corollary 19 that even with an optimal replacement strategy, ORD-MULT incurs $\Omega(n^3)$ cache misses.

ORD-MULT$(A, B)$ does not exhibit good cache behavior. Accesses to the same element, or at least to the same cache line, are far apart. The spatial locality of the memory accesses can be improved by changing the memory layout of the matrices: The previous analysis showed that the accesses to matrix $B$ alone cause $\Theta(n^3)$ cache misses. The problem is that matrix $B$ is stored in row-major order, but accessed columnwise. Assume that the memory layout for $B$ is changed from row-major to column-major order (Figure 2-1(b)):




Now, both matrices are accessed in the order in which they are stored. As long as the cache can provide a single line for each of $A$, $B$, and $C$, for each cache miss, the following $L-1$ accesses are cache hits. Hence, the number of cache misses is $\Theta(n^3/L)$, which is a factor of $\Theta(L)$ improvement over the previous algorithm. This improvement comes with the disadvantage that the matrices are not stored uniformly and it is still a factor of $\Theta(\sqrt{Z})$ away from the cache-optimal algorithms shown in Sections 1 and 2. □

## *7.2 Matrix transposition*

In this section, we argue that the iterative algorithm for matrix transposition causes $\Omega(n^2)$ cache misses on a $n \times n$ matrix, when the matrix is stored in row- or column-major order (Figure 2-1(a,b)). This is a factor of $\Theta(L)$ more cache misses than the cache-optimal algorithm presented in Section 3.

The ordinary algorithm for matrix transposition walks through the matrix row by row and swaps elements: