# Cache-Oblivious Algorithms

EXTENDED ABSTRACT SUBMITTED FOR PUBLICATION.

Matteo Frigo Charles E. Leiserson Harald Prokop Sridhar Ramachandran
*MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139*
{athena,cel,prokop,sridhar}@supertech.lcs.mit.edu

## Abstract

This paper presents asymptotically optimal algorithms for rectangular matrix transpose, FFT, and sorting on computers with multiple levels of caching. Unlike previous optimal algorithms, these algorithms are ***cache oblivious***: no variables dependent on hardware parameters, such as cache size and cache-line length, need to be tuned to achieve optimality. Nevertheless, these algorithms use an optimal amount of work and move data optimally among multiple levels of cache. 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 $\Theta(1 + (n/L)(1 + \log_Z n))$. We also give an $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache faults.

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 also 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, but without any tuning.

## 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 paper, 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 first introduce the $(Z, L)$ ***ideal-cache model*** to study the cache complexity of algorithms. This model, which is illustrated in Figure 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 paper that the cache is ***tall***:

$$Z = \Omega(L^2) , \tag{1}$$

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 delivered to the processor. Otherwise, a ***cache miss*** occurs, and the line is fetched into the cache. The ideal cache is ***fully associative*** [18, 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

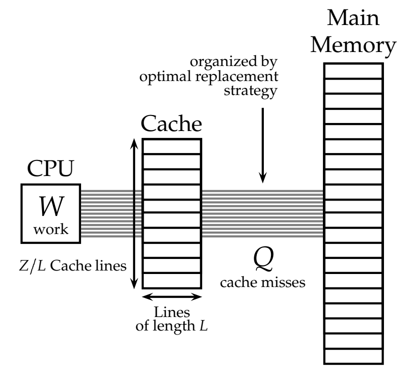

**Figure 1:** The ideal-cache model

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0270. Matteo Frigo was supported in part by a Digital Equipment Corporation fellowship.

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 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(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 [17, 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($A, B, C, n$)
1  for $i \leftarrow 1$ to $n/s$
2      do for $j \leftarrow 1$ to $n/s$
3             do for $k \leftarrow 1$ to $n/s$
4                    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. (This algorithm assumes for simplicity that $s$ evenly divides $n$, but 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 $s$ so 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), we can see that $s = \Theta(\sqrt{Z})$. Thus, each of the calls to ORD-MULT runs with at most $Z/L = \Theta(s^2/L)$ cache misses needed to bring the three matrices into the cache. Consequently, the cache complexity of the entire algorithm is $\Theta(1 + n^2/L + (n/\sqrt{Z})^3(Z/L)) = \Theta(1 + n^2/L + n^3/L\sqrt{Z})$, since the algorithm has to 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 which are 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 6 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, 28]. Section 7 discusses related work, and Section 8 offers some concluding remarks.

## 2 Matrix multiplication

This section describes an algorithm for multiplying an $m \times n$ by an $n \times p$ matrix cache-obliviously using $\Theta(mnp)$ work and incurring $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses. These results require the tall-cache assumption (1) for matrices stored with in a row-major layout format, but the assumption can be relaxed for certain other layouts. We also discuss Strassen's algorithm [25] for multiplying $n \times n$ matrices, which uses $\Theta(n^{\lg 7})$ work[1] and incurs $\Theta(1 + n^2/L +$

[1]We use the notation lg to denote $\log_2$.

$n^{\lg 7}/L\sqrt{Z})$ cache misses.

In [8], two of the authors analyzed an optimal divide-and-conquer algorithm for $n \times n$ matrix multiplication that contained no tuning parameters, but we did not study cache-obliviousness *per se*. That algorithm can be extended to multiply rectangular matrices. To multiply a $m \times n$ matrix $A$ and a $n \times p$ matrix $B$, the algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$(\mathbf{a}) \quad AB = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} ,$$

$$(\mathbf{b}) \quad AB = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 ,$$

$$(\mathbf{c}) \quad AB = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} .$$

In case (a), we have $m \ge \max\{n, p\}$. Matrix $A$ is split horizontally, and both halves are multiplied by matrix $B$. In case (b), we have $n \ge \max\{m, p\}$. Both matrices are split, and the two halves are multiplied. In case (c), we have $p \ge \max\{m, n\}$. 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 algorithm described in [8]. The base case occurs when $m = n = p = 1$, in which case the two elements are multiplied and added into the result matrix.

It can be shown by induction that the work of this algorithm is $O(mnp)$. 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(a). We further assume that any row in each of the matrices does not fit in 1 cache line, that is, $\min\{m, n, p\} \ge L$. [The final version of this paper will contain the analysis for the general case.]

The following recurrence describes the cache complexity:

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

where $\alpha$ is a constant chosen sufficiently small to allow the three submatrices (and whatever small number of temporary variables there may be) to fit in the cache. The base case arises as soon as all three matrices fit in cache. Using reasoning similar to that for analyzing ORD-MULT within

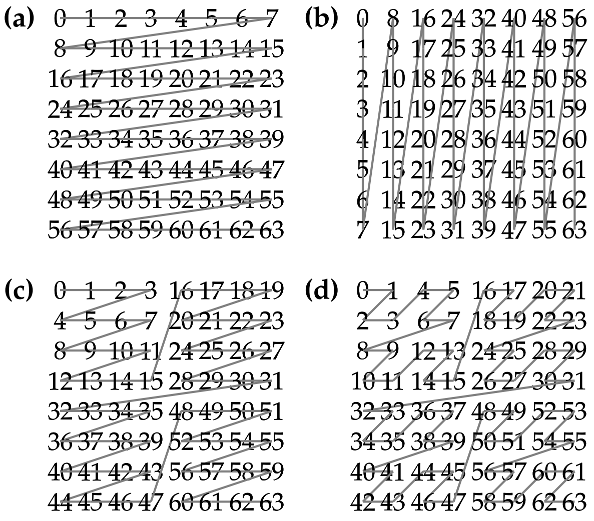

**Figure 2:** Layout of a $16 \times 16$ matrix in **(a)** row major, **(b)** column major, **(c)** $4 \times 4$-blocked, and **(d)** bit-interleaved layouts.

BLOCK-MULT, the matrices are held on $\Theta((mn + np + mp)/L)$ cache lines, assuming a tall cache. Thus, the only cache misses that occur during the remainder of the recursion are the $\Theta((mn + np + mp)/L)$ cache misses that occur when the matrices are brought into the cache. The recursive case arises when the matrices do not fit in cache, in which case 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 $Q(m, n, p) = O(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$, which is the same as the cache complexity of the cache-aware BLOCK-MULT algorithm for square matrices. Intuitively, the cache-oblivious divide-and-conquer algorithm uses cache effectively, because once a subproblem fits into the cache, no more cache misses occur for smaller subproblems.

We require the tall-cache assumption (1) in this analysis because the matrices are stored in row-major order. Tall caches are also needed if matrices are stored in column-major order (Figure 2(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(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(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 [14] and [9, 10]. 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 $Q(n) = \Theta(1 + n^2/L + n^3/L\sqrt{Z})$ of the cache-oblivious matrix multiplication algorithm matches the lower bound by Hong and Kung [19]. 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 [25] or one of its variants [31], both the work and cache complexity can be reduced. Indeed, Strassen's algorithm, which is cache oblivious, can be shown to have cache complexity $O(1 + n^2/L + n^{\lg 7}/L\sqrt{Z})$.

## 3 Matrix transposition and FFT

This section describes a cache-oblivious algorithm for transposing a $m \times n$ matrix that uses $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses, which is optimal. Using matrix transposition as a subroutine, we convert a variant [30] of the "six-step" fast Fourier transform (FFT) algorithm [6] into an optimal cache-oblivious algorithm. This FFT algorithm uses $O(n \lg n)$ work and incurs $O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ cache misses.

The problem of matrix transposition is defined as follows. Given an $m \times n$ matrix stored in a row-major layout, compute and store $A^T$ into an $n \times m$ matrix $B$ also stored in a row-major layout. The straightforward algorithm for transposition that employs doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $mn \gg Z$, which is suboptimal.

Optimal work and cache complexities can be obtained with a divide-and-conquer strategy, however. If $n \geq m$, we partition

$$A = (A_1 \; A_2) \; , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} .$$

Then, we recursively execute $\textsc{Transpose}(A_1, B_1)$ and $\textsc{Transpose}(A_2, B_2)$. If $m > n$, we divide matrix $A$ horizontally and matrix $B$ vertically and likewise perform two transpositions recursively. The next two lemmas provide upper and lower bounds on the performance of this algorithm.

**Lemma 1** *The cache-oblivious matrix-transpose algorithm involves $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses for an $m \times n$ matrix.*

*Proof.* See Appendix A. □

**Theorem 2** *The cache-oblivious matrix-transpose algorithm is asymptotically optimal.*

*Proof.* For an $m \times n$ matrix, the matrix-transposition algorithm must write to $mn$ distinct elements, which occupy at least $\lceil mn/L \rceil = \Omega(1 + mn/L)$ cache lines. □

As an example of application of the cache-oblivious transposition algorithm, in the rest of this section we describe and analyze a cache-oblivious algorithm for computing the discrete Fourier transform of a complex array of $n$ elements, where $n$ is an exact power of 2. The basic algorithm is the well-known "six-step" variant [6, 30] of the Cooley-Tukey FFT algorithm [11]. Using the cache-oblivious transposition algorithm, however, the FFT becomes cache-oblivious, and its performance matches the lower bound by Hong and Kung [19].

Recall that the ***discrete Fourier transform (DFT)*** of an array $X$ of $n$ complex numbers is the array $Y$ given by

$$Y[i] = \sum_{j=0}^{n-1} X[j] \omega_n^{-ij} \; , \tag{3}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive $n$th root of unity, and $0 \leq i < n$.

Many known algorithms evaluate Equation (3) in time $O(n \lg n)$ for all integers $n$ [13]. In this paper, however, we assume that $n$ is an exact power of 2, and compute Equation (3) according to the Cooley-Tukey algorithm, which works recursively as follows. In the base case where $n = O(1)$, we compute Equation (3) directly. Otherwise, for any factorization $n = n_1 n_2$ of $n$, we have

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[ \left( \sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] \omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} \; . \tag{4}$$

Observe that both the inner and the outer summation in Equation (4) is a DFT. Operationally, the computation specified by Equation (4) can be performed by computing $n_2$ transforms of size $n_1$

(the inner sum), multiplying the result by the factors $\omega_n^{-i_1 j_2}$ (called the ***twiddle factors*** [13]), and finally computing $n_1$ transforms of size $n_2$ (the outer sum).

We choose $n_1$ to be $2^{\lceil \lg n/2 \rceil}$ and $n_2$ to be $2^{\lfloor \lg n/2 \rfloor}$. The recursive step then operates as follows.

1. Pretend that input is a row-major $n_1 \times n_2$ matrix $A$. Transpose $A$ in-place, i.e., use the cache-oblivious algorithm to transpose $A$ onto an auxiliary array $B$, and copy $B$ back onto $A$. Notice that if $n_1 = 2n_2$, we can consider the matrix to be made up of records containing two elements.
2. At this stage, the inner sum corresponds to a DFT of the $n_2$ rows of the transposed matrix. Compute these $n_2$ DFT's of size $n_1$ recursively. Observe that, because of the previous transposition, we are transforming a contiguous array of elements.
3. Multiply $A$ by the twiddle factors, which can be computed on the fly with no extra cache misses.
4. Transpose $A$ in-place, so that the inputs to the next stage is arranged in contiguous locations.
5. Compute $n_1$ DFT's of the rows of the matrix, recursively.
6. Transpose $A$ in-place, so as to produce the correct output order.

It can be proven by induction that the work complexity of this FFT algorithm is $O(n \lg n)$. We now analyze its cache complexity. The algorithm always operates on contiguous data, by construction. In order to simplify the analysis of the cache complexity, assume a tall cache, in which case each transposition operation and the multiplication by the twiddle factors require at most $O(1 + n/L)$ cache misses. Thus, the cache complexity satisfies the recurrence

$$Q(n) \le \begin{cases} O(1+n/L), & \text{if } n \le \alpha Z \ , \\ n_1 Q(n_2) + n_2 Q(n_1) & \text{otherwise} \ , \\ \quad +O(1+n/L) & \end{cases} \tag{5}$$

for a sufficiently small constant $\alpha$ chosen such that a subproblem of size $\alpha Z$ fits in cache. This recurrence has solution

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) \ ,$$

which is asymptotically optimal for a Cooley-Tukey algorithm, matching the lower bound by Hong and Kung [19] when $n$ is an exact power of 2. As with matrix multiplication, no tight lower bounds for cache complexity are known for the general problem of computing the DFT.

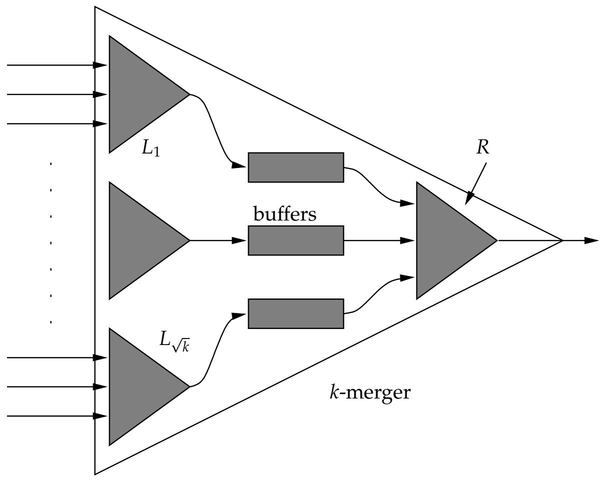

**Figure 3:** Illustration of a $k$-merger. A $k$-merger is built recursively out of $\sqrt{k}$ "left" $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, a series of buffers, and one "right" $\sqrt{k}$-merger $R$.

## 4 Funnelsort

Although it is cache oblivious, algorithms like familiar two-way merge sort are not asymptotically optimal with respect to cache misses. The $Z$-way mergesort mentioned by Aggarwal and Vitter [3] is optimal in terms of cache complexity, but it is cache aware. This section describes a cache-oblivious sorting algorithm called "funnelsort." This algorithm has an asymptotically optimal work complexity $O(n \lg n)$, and an optimal cache complexity $O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ if the cache is tall.

Funnelsort is similar to mergesort. In order to sort a (contiguous) array of $n$ elements, funnelsort performs the following two steps:

1. Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$, and sort these arrays recursively.
2. Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, which is described below.

Funnelsort differs from mergesort in the way the merge operation works. Merging is performed by a device called a ***k-merger***, which inputs $k$ sorted sequences and merges them. A $k$-merger operates by recursively merging sorted sequences that become progressively longer as the algorithm proceeds. Unlike mergesort, however, a $k$-merger stops working on a merging subproblem when the merged output sequence becomes "long enough," and it resumes working on another merging subproblem.

Since this complicated flow of control makes a $k$-merger a bit tricky to describe, we explain the operation of the $k$-merger pictorially. Figure 3 shows a representation of a $k$-merger, which has $k$ sorted sequences as inputs. Throughout its execution, the $k$-merger maintains the following invariant.

**Invariant** *The invocation of a $k$-merger outputs the first $k^3$ elements of the sorted sequence obtained by merging the $k$ input sequences.*

A $k$-merger is built recursively out of $\sqrt{k}$-mergers in the following way. The $k$ inputs are partitioned into $\sqrt{k}$ sets of $\sqrt{k}$ elements, and these sets form the input to the $\sqrt{k}$ $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$ in the left part of the figure. The outputs of these mergers are connected to the inputs of $\sqrt{k}$ ***buffers***. Each buffer is a FIFO queue that can hold $2k^{3/2}$ elements. Finally, the outputs of the buffers are connected to the $\sqrt{k}$ inputs of the $\sqrt{k}$-merger $R$ in the right part of the figure. The output of this final $\sqrt{k}$-merger becomes the output of the whole $k$-merger. The reader should notice that the intermediate buffers are overdimensioned. In fact, each buffer can hold $2k^{3/2}$ elements, which is twice the number $k^{3/2}$ of elements output by a $\sqrt{k}$-merger. This additional buffer space is necessary for the correct behavior of the algorithm, as will be explained below. The base case of the recursion is a $k$-merger with $k = 2$, which produces $k^3 = 8$ elements whenever invoked.

A $k$-merger operates recursively in the following way. In order to output $k^3$ elements, the $k$-merger invokes $R$ $k^{3/2}$ times. Before each invocation, however, the $k$-merger fills all buffers that are less than half full, i.e., all buffers that contain less than $k^{3/2}$ elements. In order to fill buffer $i$, the algorithm invokes the corresponding left merger $L_i$ once. Since $L_i$ outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after $L_i$ finishes.

It can be proven by induction that the work complexity of funnelsort is $O(n \lg n)$. The next theorem gives the cache complexity of funnelsort.

**Theorem 3** *Funnelsort sorts $n$ elements incurring at most $Q(n)$ cache misses, where*

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) .$$

*Proof.* See Appendix B. □

This upper bound matches the lower bound stated by the next theorem, proving that funnelsort is cache-optimal.

**Theorem 4** *The cache complexity of any sorting algorithm is $Q(n) = \Omega\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$.*

*Proof.* Aggarwal and Vitter [3] show that there is an $\Omega\left((n/L)\log_{Z/L}(n/Z)\right)$ bound on the number of cache misses made by any sorting algorithm on their "out-of-core" memory model, a bound that extends to the ideal-cache model. The theorem can be proved by applying the tall-cache assumption $Z = \Omega(L^2)$ and the trivial lower bounds of $Q(n) = \Omega(1)$ and $Q(n) = \Omega(n/L)$. □

## 5 Distribution sort

In this section, we describe another cache-oblivious optimal sorting algorithm based on distribution sort. Like the funnelsort algorithm from Section 4, the distribution-sorting algorithm uses $O(n \lg n)$ work to sort $n$ elements and it incurs $O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ cache misses if the cache is tall. Unlike previous cache-efficient distribution-sorting algorithms [1, 3, 21, 28, 30], which use sampling or other techniques to find the partitioning elements before the distribution step, our algorithm uses a "bucket splitting" technique to select pivots incrementally during the distribution.

Given an array $A$ (stored in contiguous locations) of length $n$, the cache-oblivious distribution sort performs sorts $A$ as follows:

1. Partition $A$ into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$. Recursively sort each subarray.
2. Distribute the sorted subarrays into $q$ buckets $B_1, \ldots, B_q$ of size $n_1, \ldots, n_q$, respectively, such that
   (a) $\max\{x \mid x \in B_i\} \le \min\{x \mid x \in B_{i+1}\}$ for all $1 \le i < q$.
   (b) $n_i \le 2\sqrt{n}$ for all $1 \le i \le q$.

   (See below for details.)
3. Recursively sort each bucket.
4. Copy the sorted buckets to array $A$.

A stack-based memory allocator is used to exploit spatial locality.

**Distribution step** The goal of Step 2 is to distribute the sorted subarrays of $A$ into $q$ buckets $B_1, B_2, \ldots, B_q$. The algorithm maintains two invariants. First, at any time each bucket holds at most $2\sqrt{n}$ elements and any element in bucket $B_i$ is smaller than any element in bucket $B_{i+1}$. Second, every bucket has an associated pivot. Initially, only one empty bucket exists with pivot $\infty$.

The idea is to copy all elements from the subarrays into the buckets while maintaining the invariants. We keep state information for each subarray and bucket. The state of a subarray consists of the index *next* of the next element to be read from the subarray and the bucket number *bnum* where this element should be copied. By convention, $bnum = \infty$ if all elements in a subarray have been copied. The state of a bucket consists of the pivot and the number of elements currently in the bucket.

We would like to copy the element at position *next* of a subarray to bucket *bnum*. If this element is greater than the pivot of bucket *bnum*, we would increment *bnum* until we find a bucket for which the element is smaller than the pivot. Unfortunately, this basic strategy has poor caching behavior, which calls for a more complicated procedure.

The distribution step is accomplished by the recursive procedure DISTRIBUTE$(i, j, m)$ which distributes elements from the $i$th through $(i + m - 1)$th subarrays into buckets starting from $B_j$. Given the precondition that each subarray $i, i + 1, \ldots, i + m - 1$ has its $bnum \geq j$, the execution of DISTRIBUTE$(i, j, m)$ enforces the postcondition that subarrays $i, i + 1, \ldots, i + m - 1$ have their $bnum \geq j + m$. Step 2 of the distribution sort invokes DISTRIBUTE$(1, 1, \sqrt{n})$. The following is a recursive implementation of DISTRIBUTE:

```
DISTRIBUTE($i, j, m$)
 1  if $m = 1$
 2    then COPYELEMS($i, j$)
 3    else DISTRIBUTE($i, j, m/2$)
 4         DISTRIBUTE($i + m/2, j, m/2$)
 5         DISTRIBUTE($i, j + m/2, m/2$)
 6         DISTRIBUTE($i + m/2, j + m/2, m/2$)
```

In the base case, the procedure COPYELEMS$(i, j)$ copies all elements from subarray $i$ that belong to bucket $j$. If bucket $j$ has more than $2\sqrt{n}$ elements after the insertion, it can be split into two buckets of size at least $\sqrt{n}$. For the splitting operation, we use the deterministic median-finding algorithm [12, p. 189] followed by a partition. The median-finding algorithm uses $O(m)$ work and incurs $O(1 + m/L)$ cache misses to find the median of an array of size $m$. (In our case, we have $m = 2\sqrt{n} + 1$.) In addition, when a bucket splits, all subarrays whose *bnum* is greater than the *bnum* of the split bucket must have their *bnum*'s incremented. The analysis of DISTRIBUTE is given by the following lemma.

**Lemma 5** *Step 2 uses $O(n)$ work, incurs $O(1 + n/L)$ cache misses, and uses $O(n)$ stack space to distribute $n$ elements.*

*Proof.* See Appendix C. □

**Theorem 6** *Distribution sort uses $O(n \lg n)$ work and incurs $O(1 + (n/L)(1 + \log_Z n))$ cache misses to sort $n$ elements.*

*Proof.* The work done by the algorithm is given by

$$W(n) = \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{q} W(n_i) + O(n) ,$$

where each $n_i \leq 2\sqrt{n}$ and $\sum n_i = n$. The solution to this recurrence is $W(n) = O(n \lg n)$.

The space complexity of the algorithm is given by

$$S(n) \leq S(2\sqrt{n}) + O(n) ,$$

where the $O(n)$ term comes from Step 2. The solution to this recurrence is $S(n) = O(n)$.

The cache complexity of distribution sort is described by the recurrence

$$Q(n) \leq \begin{cases} O(1 + n/L) & \text{if } n \leq \alpha Z , \\ \sqrt{n}Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i) & \text{otherwise} , \\ \quad + O(1 + n/L) & \end{cases}$$

where $\alpha$ is a sufficiently small constant such that the stack space used by a sorting problem of size $\alpha Z$, including the input array, fits completely in cache. The base case $n \leq \alpha Z$ arises when both the input array $A$ and the contiguous stack space of size $S(n) = O(n)$ fit in $O(1 + n/L)$ cache lines of the cache. In this case, the algorithm incurs $O(1 + n/L)$ cache misses to touch all involved memory locations once. In the case where $n > \alpha Z$, the recursive calls in Steps 1 and 3 cause $Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i)$ cache misses and $O(1 + n/L)$ is the cache complexity of Steps 2 and 4, as shown by Lemma 5. The theorem now follows by solving the recurrence. □

## 6 Other cache models

In this section we show that cache-oblivious algorithms designed in the two-level ideal-cache model can be efficiently ported to other cache models. We show that algorithms whose complexity bounds satisfy a simple regularity condition (including all algorithms heretofore presented) can be ported to less-ideal caches incorporating least-recently-used (LRU) or first-in, first-out (FIFO) replacement policies [18, p. 378]. We argue that optimal cache-oblivious algorithms are also optimal for multilevel caches. Finally, we present simulation results proving that optimal cache-oblivious algorithms satisfying the regularity condition are also optimal (in expectation) in the previously studied SUMH [5, 28] and HMM [1] models. Thus, all the algorithmic results in this paper apply to these models, matching the best bounds previously achieved.

### 6.1 Two-level models

Many researchers, such as [3, 19, 29], employ two-level models similar to the ideal-cache model, but without an automatic replacement strategy. In these models, data must be moved explicitly between the the primary and secondary levels "by hand." We define a cache complexity bound $Q(n; Z, L)$ to be ***regular*** if

$$Q(n; Z, L) = O(Q(n; 2Z, L)) . \qquad (6)$$

We now show that optimal algorithms in the ideal-cache model whose cache complexity bounds are regular can be ported to these models to run using optimal work and incurring an optimal expected number of cache misses.

The first lemma shows that the optimal and omniscient replacement strategy used by an ideal cache can be simulated efficiently by the LRU and FIFO replacement strategies.

**Lemma 7** *Consider an algorithm that causes $Q^*(n; Z, L)$ cache misses on a problem of size n using a $(Z, L)$ ideal cache. Then, the same algorithm incurs $Q(n; Z, L) \le 2Q^*(n; Z/2, L)$ cache misses on a $(Z, L)$ cache that uses either LRU or FIFO replacement.*

*Proof.* Sleator and Tarjan [24] have shown that the cache misses on a $(Z, L)$ cache using LRU replacement is $(Z/(Z - Z^* + 1))$-competitive with optimal replacement on a $(Z^*, L)$ ideal if both caches start with an empty cache. It follows that the number of misses on a $(Z, L)$ LRU-cache is at most twice the number of misses on a $(Z/2, L)$ ideal-cache. The same argument holds for FIFO caches. □

**Corollary 8** *For algorithms with regular cache complexity bounds, the asymptotic number of cache misses is the same for LRU, FIFO, and optimal replacement.* □

Since previous two-level models do not support automatic replacement, to port a cache-oblivious algorithms to them, we implement a LRU (or FIFO) replacement strategy in software.

**Lemma 9** *A $(Z, L)$ LRU-cache (or FIFO-cache) can be maintained using $O(Z)$ primary memory locations such that every access to a cache line in primary memory takes $O(1)$ expected time.*

*Proof.* Given the address of the memory location to be accessed, we use a 2-universal hash function [20, p. 216] to maintain a hash table of cache lines present in the primary memory. The $Z/L$ entries in the hash table point to linked lists in a heap of memory containing $Z/L$ records corresponding to the cache lines. The 2-universal hash function guarantees that the expected size of a chain is $O(1)$. All records in the heap are organized as a doubly linked list in the LRU order (or singly linked for FIFO). Thus, the LRU (FIFO) replacement policy can be implemented in $O(1)$ expected time using $O(Z/L)$ records of $O(L)$ words each. □

**Theorem 10** *An optimal cache-oblivious algorithm with a regular cache-complexity bound can be implemented optimally in expectation in two-level models with explicit memory management.* □

Consequently, our cache-oblivious algorithms for matrix multiplication, matrix transpose, FFT, and sorting are optimal in two-level models.

### 6.2 Multilevel ideal caches

We now show that optimal cache-oblivious algorithms also perform optimally in computers with multiple levels of ideal caches. Moreover, Theorem 10 extends to multilevel models with explicit memory management.

The $\langle (Z_1, L_1), (Z_2, L_2), \ldots, (Z_r, L_r) \rangle$ ***ideal-cache model*** consists of an arbitrarily large main memory and a hierarchy of $r$ caches, each of which is managed by an optimal replacement strategy. The model assumes that the caches satisfy the ***inclusion property*** [18, p. 723], which says that for $i = 1, 2, \ldots, r-1$, the values stored in cache $i$ are also stored in cache $i+1$. The performance of an algorithm running on an input of size $n$ is measured by its work complexity $W(n)$ and its cache complexities $Q_i(n;Z_i,L_i)$ for each level $i=1,2,\ldots,r$.

**Theorem 11** *An optimal cache-oblivious algorithm in the ideal-cache model incurs an asymptotically optimal number of cache misses on each level of a multilevel cache with optimal replacement.*

*Proof.* The theorem follows directly from the definition of cache obliviousness and the optimality of the algorithm in the two-level ideal-cache model. □

**Theorem 12** *An optimal cache-oblivious algorithm with a regular cache-complexity bound incurs an asymptotically optimal number of cache misses on each level of a multilevel cache with LRU, FIFO, or optimal replacement.*

*Proof.* Follows from Corollary 8 and Theorem 12. □

## 6.3 The SUMH model

In 1990 Alpern et al. [5] presented the uniform memory hierarchy model (UMH), a parameterized model for a memory hierarchy. In the $\mathrm{UMH}_{\alpha,\rho,b(l)}$ model, for integer constants $\alpha,\rho>1$, the size of the $i$th memory level is $Z_i=\alpha\rho^{2i}$ and the line length is $L_i=\rho^i$. A transfer of one $\rho^l$-length line between the caches on level $l$ and $l+1$ takes $\rho^l/b(l)$ time. The bandwidth function $b(l)$ must be nonincreasing and the processor accesses the cache on level 1 in constant time per access. An algorithm given for the UMH model must include a schedule that, given for a particular set of input variables, tells exactly when each block is moved along which of the buses between caches. Work and cache misses are folded into one cost measure $T(n)$. Alpern et al. prove that an algorithm that performs the optimal number of I/O's at all levels of the hierarchy does not necessarily run in optimal time in the UMH model, since scheduling bottlenecks can occur when all buses are active. In the more restrictive SUMH model [28], however, only one bus is active at a time. Consequently, we can prove that optimal cache-oblivious algorithms run in optimal expected time in the SUMH model.

**Lemma 13** *A cache-oblivious algorithm with $W(n)$ work and $Q(n;Z,L)$ cache misses on a $(Z,L)$-ideal cache can be executed in the $SUMH_{\alpha,\rho,b(l)}$ model in expected time*

$$T(n)=O\left(W(n)+\sum_{i=1}^{r-1}\frac{\rho^i}{b(i)}Q(n;\Theta(Z_i),L_i)\right),$$

*where $Z_i=\alpha\rho^{2i}$, $L_i=\rho^i$, and $Z_r$ is big enough to hold all elements used during the execution of the algorithm.*

*Proof.* Use the memory at the $i$th level as a cache of size $Z_i=\alpha\rho^{2i}$ with line length $L_i=\rho^i$ and manage it with software LRU described in Lemma 9. The $r$th level is the main memory, which is direct mapped and not organized by the software LRU mechanism. An LRU-cache of size $\Theta(Z_i)$ can be simulated by the $i$th level, since it has size $Z_i$. Thus, the number of cache misses at level $i$ is $2Q(n;\Theta(Z_i),L_i)$, and each takes $\rho^i/b(i)$ time. Since only one memory movement happens at any point in time, and there are $O(W(n))$ accesses to level 1, the lemma follows by summing the individual costs. □

**Lemma 14** *Consider a cache-oblivious algorithm whose work on a problem of size $n$ is lower-bounded by $W^*(n)$ and whose cache complexity is lower-bounded by $Q^*(n;Z,L)$ on an $(Z,L)$ ideal-cache. Then, no matter how data movement is implemented in $SUMH_{\alpha,\rho,b(l)}$, the time taken on a problem of size $n$ is at least*

$$T(n)=\Omega\left(W^*(n)+\sum_{i=1}^{r}\frac{\rho^i}{b(i)}Q^*(n,\Theta(Z_j),L_i)\right),$$

*where $Z_i=\alpha\rho^{2i}$, $L_i=\rho^i$ and $Z_r$ is big enough to hold all elements used during the execution of the algorithm.*

*Proof.* The optimal scheduling of the data movements does not need to obey the inclusion property, and thus the number of $i$th-level cache misses is at least as large as for an ideal cache of size $\sum_{j=1}^{i}Z_i=O(Z_i)$. Since $Q^*(n,Z,L)$ lower-bounds the cache misses on a cache of size $Z$, at least $Q^*(n,\Theta(Z_i),L_i)$ data movements occur at level $i$, each of which takes $\rho^i/b(i)$ time. Since only one movement can occur at a time, the total cost is the maximum of the work and the sum of the costs at all the levels, which is within a factor of 2 of their sum. □

**Theorem 15** *A cache-oblivious algorithm that is optimal in the ideal-cache model and whose cache-complexity is regular can be executed optimal expected time in the $SUMH_{\alpha,\rho,b(l)}$ model.*

*Proof.* The theorem follows directly from regularity and Lemmas 13 and 14. □

### 6.4 The HMM model

Aggarwal, Alpern, Chandra and Snir [1] proposed the hierarchical memory model (HMM) in which an access to location $x$ takes $f(x)$ time. The authors assume that $f$ is a monotonically nondecreasing function, usually of the form $\lceil \log x \rceil$ or $\lceil x^{\alpha} \rceil$. The final paper will show that optimal cache-oblivious algorithms run in optimal expected time in the HMM model.

## 7 Related work

In this section, we discuss the origin of the notion of cache-obliviousness. We also give an overview of other hierarchical memory models.

Our research group at MIT noticed as far back as 1994 that divide-and-conquer matrix multiplication was a cache-optimal algorithm that required no tuning, but we did not adopt the term "cache-oblivious" until 1997. This matrix-multiplication algorithm, as well as a cache-oblivious algorithm for LU-decomposition without pivoting, eventually appeared in [8]. Shortly after leaving our research group, Toledo [26] independently proposed a cache-oblivious algorithm for LU-decomposition, but with pivoting. For $n \times n$ matrices, Toledo's algorithm uses $\Theta(n^3)$ work and incurs $\Theta(1 + n^2/L + n^3/L\sqrt{Z})$ cache misses. More recently, our group has produced an FFT library called FFTW [16], which in its most recent incarnation [15], employs a register-allocation and scheduling algorithm inspired by our cache-oblivious FFT algorithm. The general idea that divide-and-conquer enhances memory locality has been known for a long time [23].

Previous theoretical work on understanding hierarchical memories and the I/O-complexity of algorithms has been studied in cache-aware models lacking an automatic replacement strategy. Hong and Kung [19] use the red-blue pebble game to prove lower bounds on the I/O-complexity of matrix multiplication, FFT, and other problems. The red-blue pebble game models temporal locality using two levels of memory. The model was extended by Savage [22] for deeper memory hierarchies. Aggarwal and Vitter [3] introduced spatial locality and investigated a two-level memory in which a block of $P$ contiguous items can be transferred in one step. They obtained tight bounds for matrix multiplication, FFT, sorting, and other problems. The hierarchical memory model (HMM) by Aggarwal et al. [1] treats memory as a linear array, where the cost of an access to element at location $x$ is given by a cost function $f(x)$. The BT model [2] extends HMM to support block transfers. The UMH model by Alpern et al. [5] is a multilevel model that allows I/O at different levels to proceed in parallel. Vitter and Shriver introduce parallelism, and they give algorithms for matrix multiplication, FFT, sorting, and other problems in both a two-level model [29] and several parallel hierarchical memory models [30]. Vitter [27] provides a comprehensive survey of external-memory algorithms.

## 8 Conclusion

[All is well that ends]

## Acknowledgments

Thanks to Bobby Blumofe, now of the University of Texas at Austin, who sparked early discussions at MIT about what we now call cache obliviousness. Thanks to David Wise of Indiana University, Sid Chatterjee of University of North Carolina, Chris Joerg of Compaq Cambridge Research Center, and Bin Song of MIT for helpful discussions.

## Appendix

## A Analysis of matrix transposition

**Lemma 1** *The cache-oblivious matrix-transpose algorithm involves $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses for an $m \times n$ matrix.*

*Proof.* It is clear that the algorithm does $O(mn)$ work. For the cache analysis, let $Q(m,n)$ be the cache complexity of transposing a $m \times n$ matrix. We assume that the matrices are stored in row-major order, the column-major case having a similar analysis.

Let $\alpha$ be a constant sufficiently small such that two submatrices of size $m \times n$ and $n \times m$, where $\max\{m,n\} \leq \alpha L$, fit completely in the cache even if each row is stored in a different cache line. We distinguish the following three cases cases.

**Case I:** $\max\{m,n\} \leq \alpha L$

Both the matrices fit in $O(1) + 2mn/L$ lines. From the choice of $\alpha$, the number of lines required is at most $Z/L$. Therefore $Q(m,n) = O(1 + mn/L)$.

**Case II:** $m \leq \alpha L < n$ OR $n \leq \alpha L < m$

For this case, assume first that $m \leq \alpha L < n$. The transposition algorithm divides the greater dimension $n$ by 2 and performs divide and conquer. At some point in the recursion, $n$ is in the range $\alpha L/2 \leq n \leq \alpha L$, and the whole problem fits in cache. Because the layout is row-major, at this point the input array has $n$ rows, $m$ columns, and it is laid out in contiguous locations, thus requiring at most $O(1 + nm/L)$ cache misses to be read. The output array consists of $nm$ elements in $m$ rows, where in the worst case every row lies on a different cache line. Consequently, we incur at most $O(m + nm/L)$ for writing the output array. Since $n \geq \alpha L/2$, the total cache complexity for this base case is $O(1 + m)$.

These observations yield the recurrence

$$Q(m,n) \leq \begin{cases} O(1+m) & \text{if } n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m, n/2) + O(1) & \text{otherwise} \ , \end{cases}$$

whose solution is $Q(m,n) = O(1 + mn/L)$.

The case $n \leq \alpha L < m$ is analogous.

**Case III:** $m, n > \alpha L$

Like in Case II, at some point in the recursion both $n$ and $m$ are in the range $[\alpha L/2, \alpha L]$. The whole problem fits into cache and it can be solved with at most $O(m + n + mn/L)$ cache misses.

The cache complexity thus satisfies the recurrence

$$Q(m,n) \leq \begin{cases} O(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2, n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m, n/2) + O(1) & \text{otherwise,} \end{cases}$$

whose solution is $Q(m,n) = O(1 + mn/L)$. ☐

## B Analysis of funnel sort

In this appendix, we analyze the cache complexity of funnelsort. The goal of the analysis is to show that funnelsort on $n$ elements requires at most $Q(n)$ cache misses, where

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) \ ,$$

provided that $Z = \Omega(L^2)$. [Note to the program committee: we believe that this hypothesis can be weakened to $Z = \Omega(L^{1+\epsilon})$ for all $\epsilon > 0$. If correct, this result will appear in the final paper.]

In order to prove this result, we need three auxiliary lemmas. The first lemma bounds the space required by a $k$-merger.

**Lemma 16** *A $k$-merger can be laid out in $O(k^2)$ contiguous memory locations.*

*Proof.* A $k$-merger requires $O(k^2)$ memory locations for the buffers, plus the space required by the $\sqrt{k}$-mergers. The space $S(k)$ thus satisfies the recurrence

$$S(k) \leq (\sqrt{k} + 1) S(\sqrt{k}) + O(k^2) \ ,$$

whose solution is $S(k) = O(k^2)$. ☐

In order to achieve the bound on $Q(n)$, it is important that the buffers in a $k$-merger be maintained as circular queues of size $k$. This requirement guarantees that we can manage the queue cache-efficiently, in the sense stated by the next lemma.

**Lemma 17** *Performing $r$ insert and remove operations on a circular queue causes in $O(1 + r/L)$ cache misses if two cache lines are available for the buffer.*

*Proof.* Associate the two cache lines to the head and tail of the circular queue. If a new cache line is read during a insert (delete) operation, the next $L - 1$ insert (delete) operations do not cause a cache miss. The result follows. ☐

The next lemma bounds the number of cache misses $Q_M$ incurred by a $k$-merger.

**Lemma 18** *If $Z = \Omega(L^2)$, then a $k$-merger operates with at most $Q_M(k)$ cache misses, where*

$$Q_M(k) = O\left(1 + k + k^3/L + k^3 \log_Z k/L\right) .$$

*Proof.* There are two cases: either $k < \alpha\sqrt{Z}$ or $k > \alpha\sqrt{Z}$, where $\alpha$ is a sufficiently small constant.

Assume first that $k < \alpha\sqrt{Z}$. By Lemma 16, the data structure associated with the $k$-merger requires at most $O(k^2) = O(Z)$ contiguous memory locations, and therefore it fits into cache. The $k$-merger has $k$ input queues, from which it loads $O(k^3)$ elements. Let $r_i$ be the number of elements extracted from the $i$-th input queue. Since $k < \alpha\sqrt{Z}$ and $L = O(\sqrt{Z})$, there are at least $Z/L = \Omega(k)$ cache lines available for the input buffers. Lemma 17 applies, whence the total number of cache misses for accessing the input queues is

$$\sum_{i=1}^{k} O(1 + r_i/L) = O(k + k^3/L) .$$

Similarly by Lemma 16, the cache complexity of writing the output queue is at most $O(1 + k^3/L)$. Finally, the algorithm incurs at most $O(1 + k^2/L)$ cache misses for touching its internal data structures. The total cache complexity is therefore $Q_M(k) = O\left(1 + k + k^3/L\right)$, completing the proof of the first case.

Assume now that $k > \alpha\sqrt{Z}$. In this second case, we prove by induction on $k$ that, whenever $k > \alpha\sqrt{Z}$, we have

$$Q_M(k) \leq ck^3 \log_Z k/L - A(k) , \qquad (7)$$

where $A(k) = k(1 + 2c\log_Z k/L)$ is a $o(k^3)$ term. This particular value of $A(k)$ will be justified later in the analysis.

The base case of the induction consists of values of $k$ such that $\alpha Z^{1/4} < k < \alpha\sqrt{Z}$. (It is not sufficient to just consider $k = \Theta(\sqrt{Z})$, since $k$ can become as small as $\Theta(Z^{1/4})$ in the recursive calls.) The analysis of the first case applies, yielding $Q_M(k) = O\left(1 + k + k^3/L\right)$. Because $k^2 > \alpha\sqrt{Z} = \Omega(L)$ and $k = \Omega(1)$, the last term dominates, and $Q_M(k) = O\left(k^3/L\right)$ holds. Consequently, a big enough value of $c$ can be found that satisfies Inequality (7).

For the inductive case, let $k > \alpha\sqrt{Z}$. The $k$-merger invokes the $\sqrt{k}$-mergers recursively. Since $\alpha Z^{1/4} < \sqrt{k} < k$, the inductive hypothesis can be used to bound the number $Q_M(\sqrt{k})$ of cache misses incurred by the submergers. The "right" merger $R$ is invoked exactly $k^{3/2}$ times. The total number $l$ of invocations of "left" mergers is bounded by $l < k^{3/2} + 2\sqrt{k}$. To see why, consider that every invocation of a left merger puts $k^{3/2}$ elements into some buffer. Since $k^3$ elements are output and the buffer space is $2k^2$, the bound $l < k^{3/2} + 2\sqrt{k}$ follows.

Before invoking $R$, the algorithm must check every buffer to see whether it is empty. One such check requires at most $\sqrt{k}$ cache misses, since there are $\sqrt{k}$ buffers. This check is repeated exactly $k^{3/2}$ times, leading to at most $k^2$ cache misses for all checks.

These considerations lead to the recurrence

$$Q_M(k) \leq \left(2k^{3/2} + 2\sqrt{k}\right) Q_M(\sqrt{k}) + k^2 .$$

Application of the inductive hypothesis yields the desired bound Inequality (7), as follows.

$$\begin{aligned} Q_M(k) &\leq \left(2k^{3/2} + 2\sqrt{k}\right) Q_M(\sqrt{k}) + k^2 \\ &\leq 2\left(k^{3/2} + \sqrt{k}\right)\left[\frac{ck^{3/2}\log_Z k}{2L} - A(\sqrt{k})\right] + k^2 \\ &\leq ck^3 \log_Z k/L + k^2\left(1 + c\log_Z k/L\right) \\ &\qquad - \left(2k^{3/2} + 2\sqrt{k}\right) A(\sqrt{k}) . \end{aligned}$$

If $A(k) = k(1 + 2c\log_Z k/L)$ (for example) Inequality (7) follows. ☐

**Theorem 3** *If $Z = \Omega(L^2)$, then funnelsort sorts $n$ elements with at most $Q(n)$ cache misses, where*

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) .$$

*Proof.* If $n < \alpha Z$ for a small enough constant $\alpha$, then the algorithm fits into cache. To see why, observe that only one $k$-merger is active at any time. The biggest $k$-merger is the top-level $n^{1/3}$-merger, which requires $O(n^{2/3}) < O(n)$ space. The algorithm thus can operate in $O(1 + n/L)$ cache misses.

If $N > \alpha Z$, we have the recurrence

$$Q(n) = n^{1/3} Q(n^{2/3}) + Q_M(n^{1/3}) .$$

By Lemma 18, we have $Q_M(n^{1/3}) = O\left(1 + n^{1/3} + n/L + n\log_Z n/L\right)$.

With the hypothesis $Z = \Omega(L^2)$, we have $n/L = \Omega(n^{1/3})$. Moreover, we also have $n^{1/3} = \Omega(1)$ and $\lg n = \Omega(\lg Z)$. Consequently, $Q_M(n^{1/3}) = O\left(n\log_Z n/L\right)$ holds, and the recurrence simplifies to

$$Q(n) = n^{1/3} Q(n^{2/3}) + O\left(n\log_Z n/L\right) .$$

The result follows by induction on $n$. ☐

## C Analysis of Distribution Sort

This appendix contains the proof of Lemma 5, which is used in Section 5.

**Lemma 19** *The median of $n$ elements can be found cache-obliviously using $O(n)$ work and incurring $O(1+n/L)$ cache misses.*

*Proof.* See [12, p. 189] for the linear-time median finding algorithm and the work analysis. The cache complexity is given by the same recurrence as the work complexity with a different base case.

$$Q(m) = \begin{cases} O(1+m/L) & \text{if } m \leq \alpha Z\ , \\ Q(\lceil m/5 \rceil) + Q(7m/10+6) & \text{otherwise}\ , \\ \qquad + O(1+m/L) & \end{cases}$$

where $\alpha$ is a sufficiently small constant. The result follows. □

**Lemma 5** *The distribute step uses $O(n)$ work, incurs $O(1+n/L)$ cache misses, and uses $O(n)$ stack space to distribute $n$ elements.*

*Proof.* In order to simplify the analysis of the work used by DISTRIBUTE, assume that COPY-ELEMS uses $O(1)$ work for procedural overhead. We will account for the work due to copying elements and splitting of buckets separately. The work of DISTRIBUTE is described by the recurrence

$$T(c) = 4T(c/2) + O(1)\ .$$

It follows that $T(c) = O(c^2)$, where $c = \sqrt{n}$ initially. The work due to copying elements is also $O(n)$.

The total number of bucket splits is at most $\sqrt{n}$. To see why, observe that there are at most $\sqrt{n}$ buckets at the end of the distribution step, since each bucket contains at least $\sqrt{n}$ elements. Each split operation involves $O(\sqrt{n})$ work and so the net contribution to the work is $O(n)$. Thus, the total work used by DISTRIBUTE is $W(n) = O(T(\sqrt{n})) + O(n) + O(n) = O(n)$.

For the cache analysis, we distinguish two cases. Let $\alpha$ be a sufficiently small constant such that the stack space used fits into cache.

**Case I:** $n \leq \alpha Z$

The input and the auxiliary space of size $O(n)$ fit into cache using $O(1+n/L)$ cache lines. Consequently, the cache complexity is $O(1+n/L)$.

**Case II:** $n > \alpha Z$

Let $R(c,m)$ denote the cache misses incurred by an invocation of DISTRIBUTE$(a,b,c)$ that copies $m$ elements from subarrays to buckets. We again account for the splitting of buckets separately. We first prove that $R$ satisfies the following recurrence:

$$R(c,m) \leq \begin{cases} O(L+m/L) & \text{if } c \leq \alpha L\ , \\ \sum_{1\leq i\leq 4} R(c/2, m_i) & \text{otherwise}\ , \end{cases} \qquad (8)$$

where $\sum_{1\leq i \leq 4} m_i = m$.

First, consider the base case $c \leq \alpha L$. An invocation of DISTRIBUTE$(a,b,c)$ operates with $c$ subarrays and $c$ buckets. Since there are $\Omega(L)$ cache lines, the cache can hold all the auxiliary storage involved and the currently accessed element in each subarray and bucket. In this case there are $O(L+m/L)$ cache misses. $O(c) = O(L)$ cache misses are due to the initial access to each subarray and bucket. $O(1+m/L)$ is the cache complexity of copying the $m$ elements from contiguous to contiguous locations. This completes the proof of the base case. The recursive case, when $c > \alpha L$, follows immediately from the algorithm. The solution for Equation (8) is $R(c,m) = O(L + c^2/L + m/L)$.

We still need to account for the cache misses caused by the splitting of buckets. Each split causes $O(1+\sqrt{n}/L)$ cache misses due to median finding (Lemma 19) and partitioning of $\sqrt{n}$ contiguous elements. Additional $O(1+\sqrt{n}/L)$ misses are incurred by restoring the cache. As proven in the work analysis, there are at most $\sqrt{n}$ split operations.

By adding $R(\sqrt{n}, n)$ to the split complexity, we conclude that the total cache complexity of the distribution step is $O(L + n/L + \sqrt{n}(1+\sqrt{n}/L)) = O(n/L)$.

□

## References

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, May 1987.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, California, 12–14 Oct. 1987. IEEE.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[5] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 600–608, Oct. 1990.

[6] D. H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, May 1990.

[7] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[9] S. Chatterjee, V. V. Jain, A. R. Lebeck, and S. Mundhra. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[10] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Parallel Algorithms and Architectures*, June 1999.

[11] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, Apr. 1965.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[13] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19:259–299, Apr. 1990.

[14] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, Las Vegas, NV, June 1997.

[15] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, May 1999.

[16] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Seattle, Washington, May 1998.

[17] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., 2nd edition, 1996.

[19] J.-W. Hong and H. T. Kung. I/O complexity: the red-blue pebbling game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, Milwaukee, 1981.

[20] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[21] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, 1993.

[22] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In D.-Z. Du and M. Li, editors, *Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 270–281. Springer Verlag, 1995.

[23] R. C. Singleton. An algorithm for computing the mixed radix fast Fourier transform.

*IEEE Transactions on Audio and Electroacoustics*, AU-17(2):93–103, June 1969.

[24] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, Feb. 1985.

[25] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[26] S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, Oct. 1997.

[27] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[28] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17(1–2):107–114, January and February 1993.

[29] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.

[30] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, August and September 1994.

[31] S. Winograd. On the algebraic complexity of functions. *Actes du Congrès International des Mathématiciens*, 3:283–288, 1970.