Cache-Oblivious Algorithms

EXTENDED ABSTRACT

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. We prove that an optimal cache-oblivious algorithm designed for two levels of memory is also optimal for multiple levels and that the assumption of optimal replacement in the ideal-cache model can be simulated efficiently by LRU replacement. We also provide preliminary empirical results on the effectiveness of cache-oblivious algorithms in practice.

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

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.

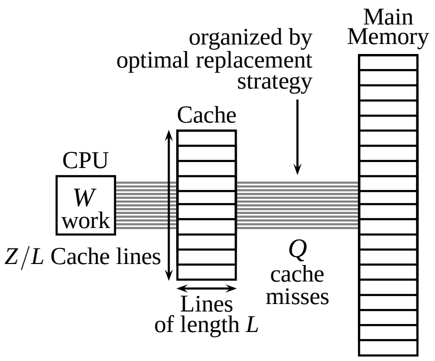

Figure 1: The ideal-cache model

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 which 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*** [20, 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 furthest in the future [7], and thus it exploits temporal locality perfectly.

Unlike various other hierarchical-memory models [1, 2, 5, 8] in which algorithms are analyzed in terms of a single measure, the ideal-cache model uses two measures. An algorithm with an input of size n is measured by 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 simply as $Q(n)$ 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 optimal[1] cache-oblivious algorithms.

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 [19, 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$(A_{ik}, B_{kj}, C_{ij}, s)$
```

The ORD-MULT(A,B,C,s) subroutine 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, yielding 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 to be the largest value 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), 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 of which are optimal in both work and cache misses.

The ideal-cache model makes the perhaps-questionable assumptions that there are only two levels in the memory hierarchy, that memory is managed automatically by an optimal cache-replacement strategy, and that the cache is fully associative. We address these assumptions in Section 6, showing that to a certain extent, these assumptions entail no loss of generality. Section 7 discusses related work, and Section 8 offers some concluding remarks, including some preliminary empirical results.

2. Matrix multiplication

This section describes and analyzes a cache-oblivious 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) 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 [31] for multiplying $n \times n$ matrices, which uses $\Theta(n^{\lg 7})$ work[2], incurs $\Theta(1 + n^2/L + n^{\lg 7}/L\sqrt{Z})$ cache misses.

In [9] with others, two of the present 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 REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}, \qquad (2)$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2, \qquad (3)$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}. \qquad (4)$$

In case (2), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B. In case (3), we have $n \geq \max\{m, p\}$. Both matrices are

[1]For simplicity in this paper, we use the term "optimal" as a synonym for "asymptotically optimal," since all our analyses are asymptotic.

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

split, and the two halves are multiplied. In case (4), 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 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 REC-MULT algorithm, we assume that the three matrices are stored in row-major order, as shown in Figure 2(a). Intuitively, REC-MULT 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* REC-MULT *algorithm uses* $\Theta(mnp)$ *work and incurs* $\Theta(m+n+p+(mn+np+mp)/L+mnp/L\sqrt{Z})$ *cache misses when multiplying an* $m \times n$ *matrix by an* $n \times p$ *matrix.*

Proof. It can be shown by induction that the work of REC-MULT is $\Theta(mnp)$. To analyze the cache misses, let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$, where $\max\{m',n',p'\} \leq \alpha\sqrt{Z}$, all fit completely in the cache. We distinguish four 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 can be described by the recurrence

$$Q(m,n,p) \leq \qquad (5)$$
$$\begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}]\ , \\ 2Q(m/2,n,p)+O(1) & \text{ow. if } m \geq n \text{ and } m \geq p\ , \\ 2Q(m,n/2,p)+O(1) & \text{ow. if } n > m \text{ and } n \geq p\ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise}\ . \end{cases}$$

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 ($n \leq \alpha\sqrt{Z}$ and $m,p > \alpha\sqrt{Z}$) 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 REC-MULT algorithm always divides n or p by 2 according to cases (3) and (4). At some point

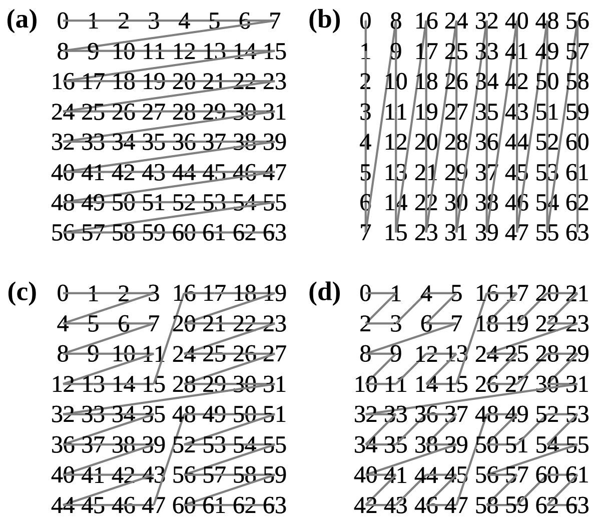

Figure 2: Layout of a 16×16 matrix in **(a)** row major, **(b)** column major, **(c)** 4×4-blocked, and **(d)** bit-interleaved layouts.

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

$$Q(m,n,p) \leq \qquad (6)$$
$$\begin{cases} \Theta(1+n+np/L+m) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}]\ , \\ 2Q(m,n/2,p)+O(1) & \text{otherwise if } n \geq p\ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise}\ ; \end{cases}$$

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

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 REC-MULT algorithm always divides m by 2 according to case (2). At some point in the recursion, m falls into the range $\alpha\sqrt{Z}/2 \leq m \leq \alpha\sqrt{Z}$, and the whole problem fits in cache. The number cache misses can be described by the recurrence

$$Q(m,n) \leq \qquad (7)$$
$$\begin{cases} \Theta(1+m) & \text{if } m \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}]\ , \\ 2Q(m/2,n,p)+O(1) & \text{otherwise}\ ; \end{cases}$$

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

Case IV: $m,n,p \leq \alpha\sqrt{Z}$. From the choice of α, 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) 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(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 $O(\sqrt{L}) \times O(\sqrt{L})$ are cache-obliviously stored on $O(1)$ cache lines. The advantages of bit-interleaved and related layouts have been studied in [11, 12, 16]. One of the practical disadvantages of bit-interleaved layouts is that index calculations on conventional microprocessors can be costly, a deficiency we hope that processor architects will remedy.

For square matrices, the cache complexity $Q(n) = \Theta(n + n^2/L + n^3/L\sqrt{Z})$ of the REC-MULT 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 [21]. 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 [31] or one of its variants [37], 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

$$Q(n) \le \begin{cases} \Theta(1 + n + n^2/L) & \text{if } n^2 \le \alpha Z \ , \\ 7Q(n/2) + O(n^2/L) & \text{otherwise ;} \end{cases} \tag{8}$$

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

3. Matrix transposition and FFT

This section describes a recursive cache-oblivious algorithm for transposing an $m \times n$ matrix which 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 [36] 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 $O(1 + (n/L)(1 + \log_Z n))$ 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 $m \gg Z/L$ and $n \gg Z/L$, which is suboptimal.

Optimal work and cache complexities can be obtained with a divide-and-conquer strategy, however. If $n \ge m$, the REC-TRANSPOSE algorithm partitions

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

and recursively executes REC-TRANSPOSE(A_1, B_1) and REC-TRANSPOSE(A_2, B_2). Otherwise, it divides matrix A horizontally and matrix B vertically and likewise performs two transpositions recursively. The next two lemmas provide upper and lower bounds on the performance of this algorithm.

Lemma 2 *The* REC-TRANSPOSE *algorithm involves $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses for an $m \times n$ matrix.*

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

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

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

Case II: $m \le \alpha L < n$ or $n \le \alpha L < m$. Suppose first that $m \le \alpha L < n$. The REC-TRANSPOSE algorithm divides the greater dimension n by 2 and performs divide and conquer. At some point in the recursion, n falls into the range $\alpha L/2 \le n \le \alpha L$, and the whole problem fits in cache. Because the layout is row-major, at this point the input array has n rows and m columns, and it is laid out in contiguous locations, 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 $n \ge \alpha L/2$, the total cache complexity for this base case is $O(1 + m)$. These observations yield the recurrence

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

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

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

Case III: $m,n > \alpha L$. As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$. The whole problem fits into cache and can be solved with at most $O(m+n+mn/L)$ cache misses. The cache complexity thus satisfies the recurrence

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

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

Theorem 3 *The* REC-TRANSPOSE *algorithm has optimal cache complexity.*

Proof. For an $m \times n$ matrix, the 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 an application of this 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, 36] of the Cooley-Tukey FFT algorithm [13]. Using the cache-oblivious transposition algorithm, however, the FFT becomes cache-oblivious, and its performance matches the lower bound by Hong and Kung [21].

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

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} \;, \tag{9}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive nth root of unity, and $0 \le i < n$. Many algorithms evaluate Equation (9) in $O(n \lg n)$ time for all integers n [15]. In this paper, however, we assume that n is an exact power of 2, and we compute Equation (9) according to the Cooley-Tukey algorithm, which works recursively as follows. In the base case where $n = O(1)$, we compute Equation (9) directly. Otherwise, for any factorization $n = n_1 n_2$ of n, we have

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} \;. \tag{10}$$

Observe that both the inner and outer summations in Equation (10) are DFT's. Operationally, the computation specified by Equation (10) 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*** [15]), 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 REC-TRANSPOSE 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 are 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. Thus, by the tall-cache assumption (1), the transposition operations and the twiddle-factor multiplication require at most $O(1+n/L)$ cache misses. Thus, the cache complexity satisfies the recurrence

$$Q(n) \le \begin{cases} O(1+n/L), & \text{if } n \le \alpha Z \;, \\ n_1 Q(n_2) + n_2 Q(n_1) & \text{otherwise ;} \\ \quad +O(1+n/L) & \end{cases} \tag{11}$$

where $\alpha > 0$ is a constant sufficiently small that a subproblem of size αZ fits in cache. This recurrence has solution

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

which is optimal for a Cooley-Tukey algorithm, matching the lower bound by Hong and Kung [21] when n is an exact power of 2. As with matrix multiplication, no tight lower bounds for cache complexity are known for the general DFT problem.

4. Funnelsort

Cache-oblivious algorithms, like the familiar two-way merge sort, are not optimal with respect to cache misses. The Z-way mergesort suggested by Aggarwal and Vitter [3] has optimal cache complexity, but although it apparently works well in practice [23], it is cache aware. This section describes a cache-oblivious sorting algorithm called "funnelsort." This algorithm has optimal

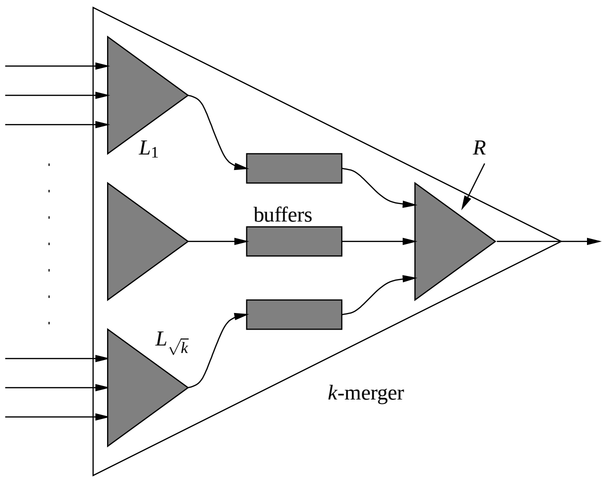

Figure 3: Illustration of a k-merger. A k-merger is built recursively out of $\sqrt{k}$ "left" $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, a series of buffers, and one "right" $\sqrt{k}$-merger R.

$O(n \lg n)$ work complexity, and optimal $O(1 + (n/L)(1 + \log_Z n))$ cache complexity.

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 which become progressively longer as the algorithm proceeds. Unlike mergesort, however, a k-merger suspends work on a merging subproblem when the merged output sequence becomes "long enough" and resumes work on another merging subproblem.

This complicated flow of control makes a k-merger a bit tricky to describe. 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 *Each invocation of a k-merger outputs the next 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, which 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 intermediate buffers are overdimensioned, since each 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 $O(n \lg n)$. We will now analyze the cache complexity. The goal of the analysis is to show that funnelsort on n elements requires at most $Q(n)$ cache misses, where

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

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

Lemma 4 *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 $Q(n)$, the buffers in a k-merger must 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 5 *Performing r insert and remove operations on a circular queue causes in $O(1 + r/L)$ cache misses as long as two cache lines are available for the buffer.*

Proof. Associate the two cache lines with 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 next lemma bounds the cache complexity of a k-merger.

Lemma 6 *If $Z = \Omega(L^2)$, then a k-merger operates with at most*

$$Q_{\mathrm{M}}(k) = O(1 + k + k^3/L + k^3 \log_Z k/L)$$

cache misses.

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

Case I: $k < \alpha\sqrt{Z}$. By Lemma 4, 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 ith input queue. Since $k < \alpha\sqrt{Z}$ and the tall-cache assumption (1) implies that $L = O(\sqrt{Z})$, there are at least $Z/L = \Omega(k)$ cache lines available for the input buffers. Lemma 5 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, Lemma 4 implies that the cache complexity of writing the output queue is $O(1 + k^3/L)$. Finally, the algorithm incurs $O(1 + k^2/L)$ cache misses for touching its internal data structures. The total cache complexity is therefore $Q_{\mathrm{M}}(k) = O(1 + k + k^3/L)$.

Case I: $k > \alpha\sqrt{Z}$. We prove by induction on k that whenever $k > \alpha\sqrt{Z}$, we have

$$Q_{\mathrm{M}}(k) \le ck^3 \log_Z k/L - A(k) , \qquad (12)$$

where $A(k) = k(1 + 2c\log_Z k/L) = o(k^3)$. This particular value of $A(k)$ will be justified at the end of 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 only to 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 $Q_{\mathrm{M}}(k) = O(1 + k + k^3/L)$. Because $k^2 > \alpha\sqrt{Z} = \Omega(L)$ and $k = \Omega(1)$, the last term dominates, which implies $Q_{\mathrm{M}}(k) = O(k^3/L)$. Consequently, a big enough value of c can be found that satisfies Inequality (12).

For the inductive case, suppose that $k > \alpha\sqrt{Z}$. 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 $Q_{\mathrm{M}}(\sqrt{k})$ 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

$$Q_{\mathrm{M}}(k) \le \left(2k^{3/2} + 2\sqrt{k}\right) Q_{\mathrm{M}}(\sqrt{k}) + k^2 .$$

Application of the inductive hypothesis and the choice $A(k) = k(1 + 2c\log_Z k/L)$ yields Inequality (12) as follows:

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

Theorem 7 *To sort n elements, funnelsort incurs $O(1 + (n/L)(1 + \log_Z n))$ cache misses.*

Proof. If $n < \alpha Z$ for a small enough constant α, 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

$$Q(n) = n^{1/3} Q(n^{2/3}) + Q_{\mathrm{M}}(n^{1/3}) .$$

By Lemma 6, we have $Q_{\mathrm{M}}(n^{1/3}) = O(1 + n^{1/3} + n/L + n\log_Z n/L)$.

By the tall-cache assumption (1), we have $n/L = \Omega(n^{1/3})$. Moreover, we also have $n^{1/3} = \Omega(1)$ and $\lg n = \Omega(\lg Z)$. Consequently, $Q_{\mathrm{M}}(n^{1/3}) = O(n\log_Z n/L)$ holds, and the recurrence simplifies to

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

The result follows by induction on n. $\square$

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

Theorem 8 *The cache complexity of any sorting algorithm is $Q(n) = \Omega(1 + (n/L)(1 + \log_Z n))$.*

Proof. Aggarwal and Vitter [3] show that there is an $\Omega((n/L)\log_{Z/L}(n/Z))$ 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 $Q(n) = \Omega(n/L)$. $\square$

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 $O(1 + (n/L)(1 + \log_Z n))$ cache misses. Unlike previous cache-efficient distribution-sorting algorithms [1, 3, 25, 34, 36], 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 step.

Given an array A (stored in contiguous locations) of length n, the cache-oblivious distribution sort operates 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
    1. $\max\{x \mid x \in B_i\} \le \min\{x \mid x \in B_{i+1}\}$ for $i = 1, 2, \ldots, q-1$.
    2. $n_i \le 2\sqrt{n}$ for $i = 1, 2, \ldots, q$.

    (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.

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 ∞.

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 ith 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 $bnum \ge j$, the execution of DISTRIBUTE(i, j, m) enforces the postcondition that subarrays $i, i+1, \ldots, i+m-1$ have their $bnum \ge j+m$. 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 [14, p. 189] followed by a partition.

Lemma 9 *The median of n elements can be found cache-obliviously using $O(n)$ work and incurring $O(1 + n/L)$ cache misses.*

Proof. See [14, 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.

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

where α is a sufficiently small constant. The result follows. $\square$

In our case, we have buckets of size $2\sqrt{n} + 1$. 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 10 *The distribution step involves $O(n)$ 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 COPYELEMS 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 α be a sufficiently small constant such that the stack space used fits into cache.

Case I, $n \le \alpha Z$: 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 first prove that $R(c,m) = O(L + c^2/L + m/L)$, ignoring the cost splitting of buckets, which we shall account for separately. We argue that $R(c,m)$ satisfies the recurrence

$$R(c,m) \le \begin{cases} O(L+m/L) & \text{if } c \le \alpha L \ , \\ \sum_{i=1}^{4} R(c/2, m_i) & \text{otherwise} \ ; \end{cases} \qquad (13)$$

where $\sum_{i=1}^{4} m_i = m$, whose solution is $R(c,m) = O(L + c^2/L + m/L)$. The recursive case $c > \alpha L$ follows immediately from the algorithm. The base case $c \le \alpha L$ can be justified as follows. 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. The initial access to each subarray and bucket causes $O(c) = O(L)$ cache misses. Copying the m elements to and from contiguous locations causes $O(1+m/L)$ cache misses.

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 9) and partitioning of $\sqrt{n}$ contiguous elements. An 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 $O(L + n/L + \sqrt{n}(1+\sqrt{n}/L)) = O(n/L)$. □

The analysis of distribution sort is given in the next theorem. The work and cache complexity match lower bounds specified in Theorem 8.

Theorem 11 *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 $n_i \le 2\sqrt{n}$ 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

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

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

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

where α is a sufficiently small constant such that the stack space used by a sorting problem of size αZ, including the input array, fits completely in cache. The base case $n \le \alpha Z$ 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 10. The theorem follows by solving the recurrence. □

6. Theoretical justifications for the ideal-cache model

How reasonable is the ideal-cache model for algorithm design? The model incorporates four major assumptions that deserve scrutiny:

- optimal replacement,
- exactly two levels of memory,
- automatic replacement,
- full associativity.

Designing algorithms in the ideal-cache model is easier than in models lacking these properties, but are these assumptions too strong? In this section we show that cache-oblivious algorithms designed in the ideal-cache model can be efficiently simulated by weaker models.

The first assumption that we shall eliminate is that of optimal replacement. Our strategy for the simulation is to use an LRU (least-recently used) replacement strategy [20, p. 378] in place of the optimal and omniscient replacement strategy. We start by proving a lemma that bounds the effectiveness of the LRU simulation. We then show that algorithms whose complexity bounds satisfy a simple regularity condition (including all algorithms heretofore presented) can be ported to caches incorporating an LRU replacement policy.

Lemma 12 *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 $Q(n;Z,L) \leq 2Q^*(n;Z/2,L)$ cache misses on a (Z,L) cache that uses LRU replacement.*

Proof. Sleator and Tarjan [30] have shown that the cache misses on a (Z,L) cache using LRU replacement are $(Z/L)/((Z-Z^*)/L+1)$-competitive with optimal replacement on a (Z^*,L) ideal cache if both caches start empty. 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. □

Corollary 13 *For any algorithm whose cache-complexity bound $Q(n;Z,L)$ in the ideal-cache model satisfies the regularity condition*

$$Q(n;Z,L) = O(Q(n;2Z,L)) \; , \tag{14}$$

the number of cache misses with LRU replacement is $\Theta(Q(n;Z,L))$.

Proof. Follows directly from (14) and Lemma 12. □

The second assumption we shall eliminate is the assumption of only two levels of memory. Although models incorporating multiple levels of caches may be necessary to analyze some algorithms, for cache-oblivious algorithms, analysis in the two-level ideal-cache model suffices. Specifically, optimal cache-oblivious algorithms also perform optimally in computers with multiple levels of LRU caches. We assume that the caches satisfy the ***inclusion property*** [20, p. 723], which says that the values stored in cache i are also stored in cache $i+1$ (where cache 1 is the cache closest to the processor). We also assume that if two elements belong to the same cache line at level i, then they belong to the same line at level $i+1$. Moreover, we assume that cache $i+1$ has strictly more cache lines than cache i. These assumptions ensure that cache $i+1$ includes the contents of cache i plus at least one more cache line.

The multilevel LRU cache operates as follows. A hit on an element in cache i is served by cache i and is not seen by higher-level caches. We consider a line in cache $i+1$ to be ***marked*** if any element stored on the line belongs to cache i. When cache i misses on an access, it recursively fetches the needed line from cache $i+1$, replacing the least-recently accessed unmarked cache line. The replaced cache line is then brought to the front of cache $(i+1)$'s LRU list. Because marked cache lines are never replaced, the multilevel cache maintains the inclusion property. The next lemma, whose proof is omitted, asserts that even though a cache in a multilevel model does not see accesses that hit at lower levels, it nevertheless behaves like the first-level cache of a simple two-level model, which sees all the memory accesses.

Lemma 14 *A (Z_i,L_i)-cache at a given level i of a multilevel LRU model always contains the same cache lines as a simple (Z_i,L_i)-cache managed by LRU that serves the same sequence of memory accesses.* □

Lemma 15 *An optimal cache-oblivious algorithm whose cache complexity satisifies the regularity condition (14) incurs an optimal number of cache misses on each level[3] of a multilevel cache with LRU replacement.*

Proof. Let cache i in the multilevel LRU model be a (Z_i,L_i) cache. Lemma 14 says that the cache holds exactly the same elements as a (Z_i,L_i) cache in a two-level LRU model. From Corollary 13, the cache complexity of a cache-oblivious algorithm working on a (Z_i,L_i) LRU cache lower-bounds that of any cache-aware algorithm for a (Z_i,L_i) ideal cache. A (Z_i,L_i) level in a multilevel cache incurs at least as many cache misses as a (Z_i,L_i) ideal cache when the same algorithm is executed. □

Finally, we remove the two assumptions of automatic replacement and full associativity. Specifically, we shall show that a fully associative LRU cache can be maintained in ordinary memory with no asymptotic loss in expected performance.

Lemma 16 *A (Z,L) LRU-cache can be maintained using $O(Z)$ memory locations such that every access to a cache line in memory takes $O(1)$ expected time.*

Proof. Given the address of the memory location to be accessed, we use a 2-universal hash function [24, p. 216] to maintain a hash table of cache lines present in the memory. The Z/L entries in the hash table point to linked lists in a heap of memory that contains 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. Thus, the LRU policy can be implemented in $O(1)$ expected time using $O(Z/L)$ records of $O(L)$ words each. □

[3]Alpern, Carter and Feig [5] show that optimality on each level of memory in the UMH model does not necessarily imply global optimality. The UMH model incorporates a single cost measure that combines the costs of work and cache faults at each of the levels of memory. By analyzing the levels independently, our multilevel ideal-cache model remains agnostic about the various schemes by which work and cache faults might be combined.

Theorem 17 *An optimal cache-oblivious algorithm whose cache-complexity bound satisfies the regularity condition (14) can be implemented optimally in expectation in multilevel models with explicit memory management.*

Proof. Combine Lemma 15 and Lemma 16. □

Corollary 18 *The recursive cache-oblivious algorithms for matrix multiplication, matrix transpose, FFT, and sorting are optimal in multilevel models with explicit memory management.*

Proof. Their complexity bounds satisfy the regularity condition (14). □

It can also be shown [26] that cache-oblivous algorithms satisfying (14) are also optimal (in expectation) in the previously studied SUMH [5, 34] and HMM [1] models. Thus, all the algorithmic results in this paper apply to these models, matching the best bounds previously achieved.

Other simulation results can be shown. For example, by using the copying technique of [22], cache-oblivious algorithms for matrix multiplication and other problems can be designed that are provably optimal on direct-mapped caches.

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 [9]. Shortly after leaving our research group, Toledo [32] independently proposed a cache-oblivious algorithm for LU-decomposition 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 [18], which in its most recent incarnation [17], 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 [29].

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, although [10, 28] are recent exceptions. Hong and Kung [21] 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 [27] 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 [35] and several parallel hierarchical memory models [36]. Vitter [33] provides a comprehensive survey of external-memory algorithms.

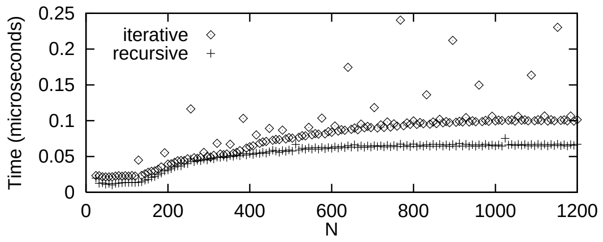

Figure 4: Average time to transpose an $N \times N$ matrix, divided by N^2.

8. Conclusion

The theoretical work presented in this paper opens two important avenues for future research. The first is to determine the range of practicality of cache-oblivious algorithms, or indeed, of any algorithms developed in the ideal-cache model. The second is to resolve, from a complexity-theoretic point of view, the relative strengths of cache-oblivious and cache-aware algorithms. To conclude, we discuss each of these avenues in turn.

Figure 4 compares per-element time to transpose a matrix using the naive iterative algorithm employing a doubly nested loop with the recursive cache-oblivious REC-TRANSPOSE algorithm from Section 3. The two algorithms were evaluated on a 450 megahertz AMD K6III processor with a 32-kilobyte 2-way set-associative L1 cache, a 64-kilobyte 4-way set-associative L2 cache, and a 1-megabyte L3 cache of unknown associativity, all with 32-byte cache lines. The code for REC-TRANSPOSE was the same as presented in Section 3, ex-

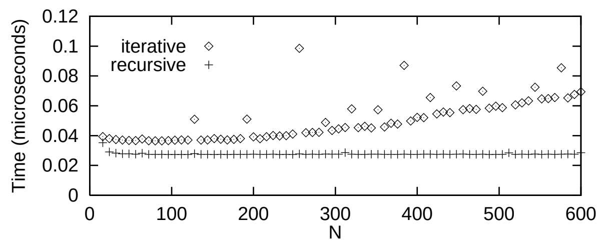

Figure 5: Average time taken to multiply two $N \times N$ matrices, divided by N^3.

cept that the divide-and-conquer structure was modified to produce exact powers of 2 as submatrix sizes wherever possible. In addition, the base cases were "coarsened" by inlining the recursion near the leaves to increase their size and overcome the overhead of procedure calls. (A good research problem is to determine an effective compiler strategy for coarsening base cases automatically.)

Although these results must be considered preliminary, Figure 4 strongly indicates that the recursive algorithm outperforms the iterative algorithm throughout the range of matrix sizes. Moreover, the iterative algorithm behaves erratically, apparently due to so-called "conflict" misses [20, p. 390], where limited cache associativity interacts with the regular addressing of the matrix to cause systematic interference. Blocking the iterative algorithm should help with conflict misses [22], but it would make the algorithm cache aware. For large matrices, the recursive algorithm executes in less than 70% of the time used by the iterative algorithm, even though the transpose problem exhibits no temporal locality.

Figure 5 makes a similar comparison between the naive iterative matrix-multiplication algorithm, which uses three nested loops, with the $O(n^3)$-work recursive REC-MULT algorithm described in Section 2. This problem exhibits a high degree of temporal locality, which REC-MULT exploits effectively. As the figure shows, the average time used per integer multiplication in the recursive algorithm is almost constant, which for large matrices, is less than 50% of the time used by the iterative variant. A similar study for Jacobi multipass filters can be found in [26].

Several researchers [12, 16] have also observed that recursive algorithms exhibit performance advantages over iterative algorithms for computers with caches. A comprehensive empirical study has yet to be done, however. Do cache-oblivious algorithms perform nearly as well as cache-aware algorithms in practice, where constant factors matter? Does the ideal-cache model capture the substantial caching concerns for an algorithms designer?

An anecdotal affirmative answer to these questions is exhibited by the popular FFTW library [17, 18], which uses a recursive strategy to exploit caches in Fourier transform calculations. FFTW's code generator produces straight-line "codelets," which are coarsened base cases for the FFT algorithm. Because these codelets are cache oblivious, a C compiler can perform its register allocation efficiently, and yet the codelets can be generated without knowing the number of registers on the target architecture.

To close, we mention two theoretical avenues that should be explored to determine the complexity-theoretic relationship between cache-oblivious algorithms and cache-aware algorithms.

Separation: *Is there a gap in asymptotic complexity between cache-aware and cache-oblivious algorithms?* It appears that cache-aware algorithms should be able to use caches better than cache-oblivious algorithms, since they have more knowledge about the system on which they are running. Do there exist problems for which this advantage is asymptotically significant, for example an $\Omega(\lg Z)$ advantage? Bilardi and Peserico [8] have recently taken some steps in proving a separation.

Simulation: *Is there a limit as to how much better a cache-aware algorithm can be than a cache-oblivious algorithm for the same problem?* That is, given a class of optimal cache-aware algorithms to solve a single problem, can we construct a good cache-oblivious algorithm that solves the same problem with only, for example, $O(\lg Z)$ loss of efficiency? Perhaps simulation techniques can be used to convert a class of efficient cache-aware algorithms into a comparably efficient cache-oblivious algorithm.

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 Gianfranco Bilardi of University of Padova, Sid Chatterjee of University of North Carolina, Chris Joerg of Compaq CRL, Martin Rinard of MIT, Bin Song of MIT, Sivan Toledo of Tel Aviv University, and David Wise of Indiana University for helpful discussions. Thanks also to our anonymous reviewers.

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 (STOC)*, 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 (FOCS)*, 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 (FOCS)*, 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] G. Bilardi and E. Peserico. Efficient portability across memory hierarchies. Unpublished manuscript, 1999.

[9] 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.

[10] L. Carter and K. S. Gatlin. Towards an optimal bit-reversal permutation program. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 544–555. IEEE Computer Society Press, 1998.

[11] 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.

[12] 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 Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1999.

[13] 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.

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

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

[16] 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 (PPoPP)*, pages 206–216, Las Vegas, NV, June 1997.

[17] 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.

[18] 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.

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

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

[21] 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 (STOC)*, pages 326–333, Milwaukee, 1981.

[22] M. S. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algortihms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, Santa Clara, CA, Apr. 1991. ACM SIGPLAN Notices 26:4.

[23] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 370–377, 1997.

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

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

[26] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[27] 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.

[28] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. Unpublished manuscript, 1999.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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