Cache Oblivious Algorithms

Algorithm Engineering

Jinha Kim

Outline

- Overview of caches and ideal cache model
- Example of Cache-Aware Algorithm for Matrix Multiplication
- Cache Oblivious Algorithm for Matrix Multiplication
- Cache Oblivious Algorithm for Sorting with Distributed Sorting
- Empirical Results

Cache-Oblivious Algorithms

MATTEO FRIGO, CHARLES E. LEISERSON, HARALD PROKOP, and
SRIDHAR RAMACHANDRAN, MIT Laboratory for Computer Science

This article presents asymptotically optimal algorithms for rectangular matrix transpose, fast Fourier transform (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 $\mathcal{M}$ and cache-line length $\mathcal{B}$ where $\mathcal{M} = \Omega(\mathcal{B}^2)$, the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/\mathcal{B})$. The number of cache misses for either an n-point FFT or the sorting of n numbers is $\Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$. We also give a $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(1 + (mn + np + mp)/\mathcal{B} + mnp/\mathcal{B}\sqrt{\mathcal{M}})$ 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 offer empirical evidence that cache-oblivious algorithms perform well in practice.

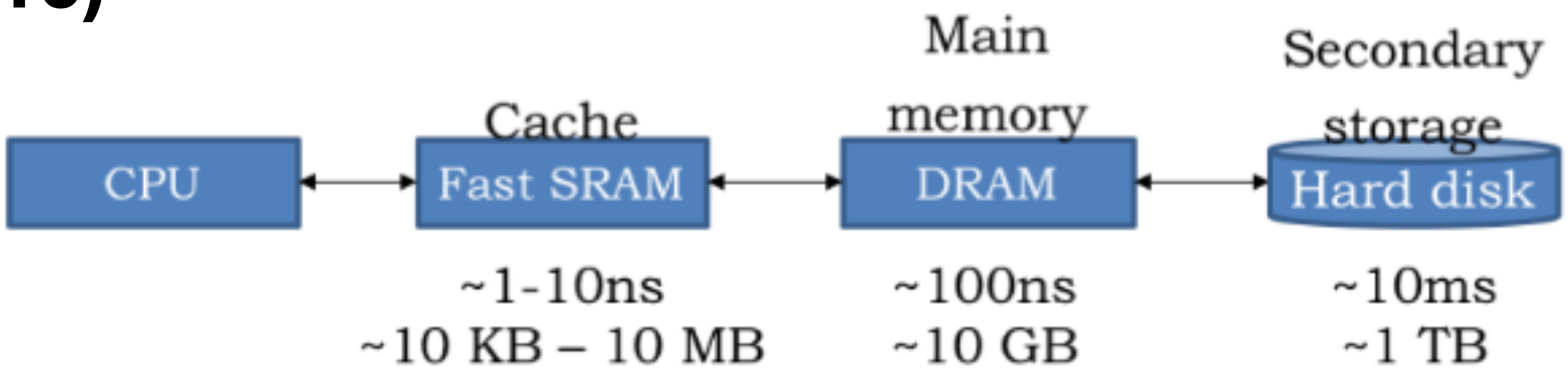
Categories and Subject Descriptors: F.2 [**Analysis of Algorithms and Problem Complexity**]: General

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Algorithm, caching, cache-oblivious, fast Fourier transform, I/O complexity, matrix multiplication, matrix transpose, sorting

ACM Reference Format:
Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. 2012. Cache-oblivious algorithms. ACM Trans. Algorithms 8, 1, Article 4 (January 2012), 22 pages.
DOI = 10.1145/2071379.2071383 http://doi.acm.org/10.1145/2071379.2071383

Memory Hierarchy in Modern Computers
(Source: 6.1910)

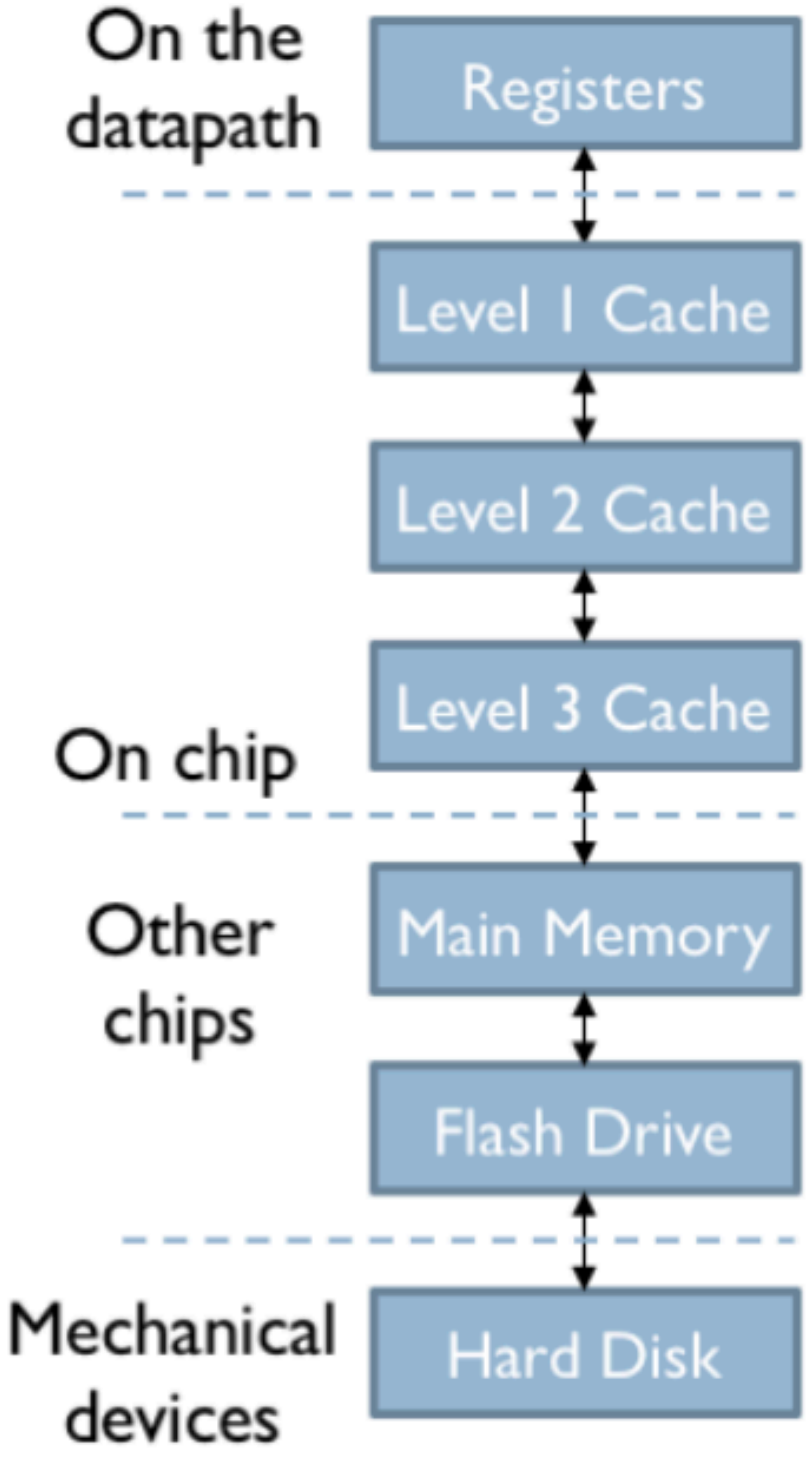


- From 6.1910, we know…
  - DRAM is 10~100x slower than SRAM
  - Disk is 100,000x slower than DRAM
  - Fetching successive words in DRAM is ~5x faster than first word
  - Fetching successive words in Disk is ~100,000x faster than first word

Caches
(Source: 6.1910)

- Cache: small interim storage component that retains data from recently accessed locations

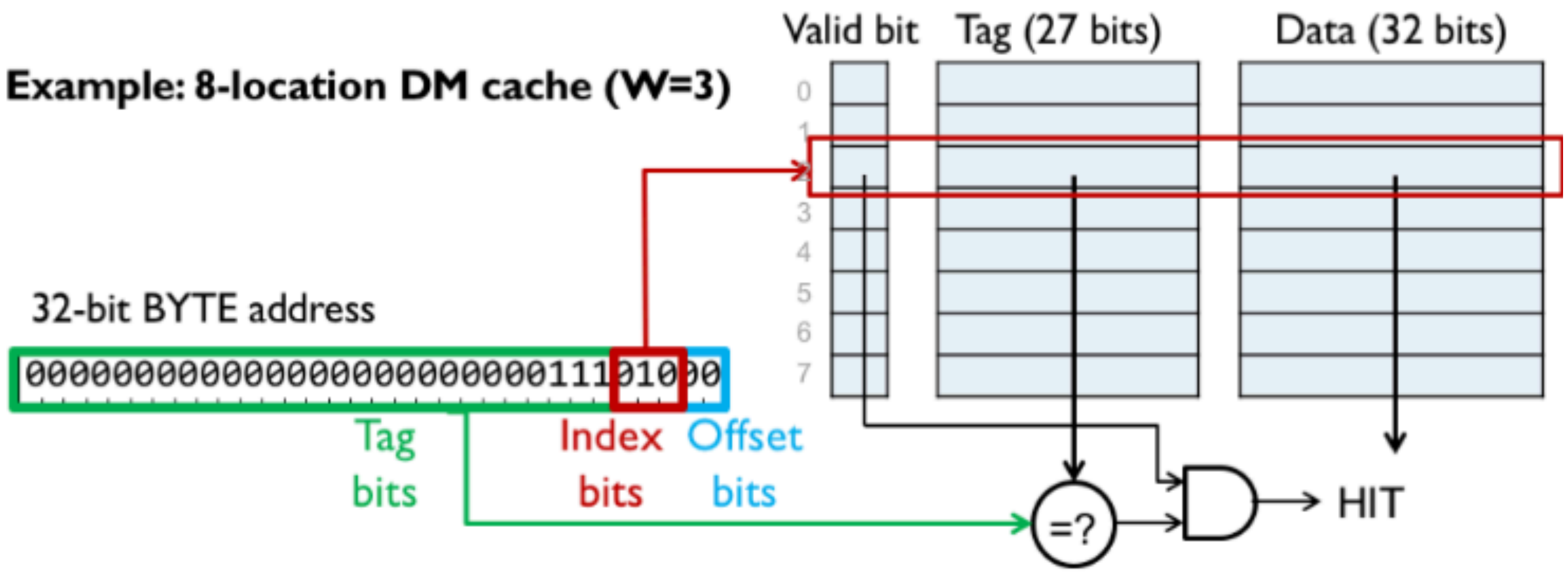
- Relies on locality principle (it is likely for accesses that happen at a similar time to also access places that are close in memory)



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

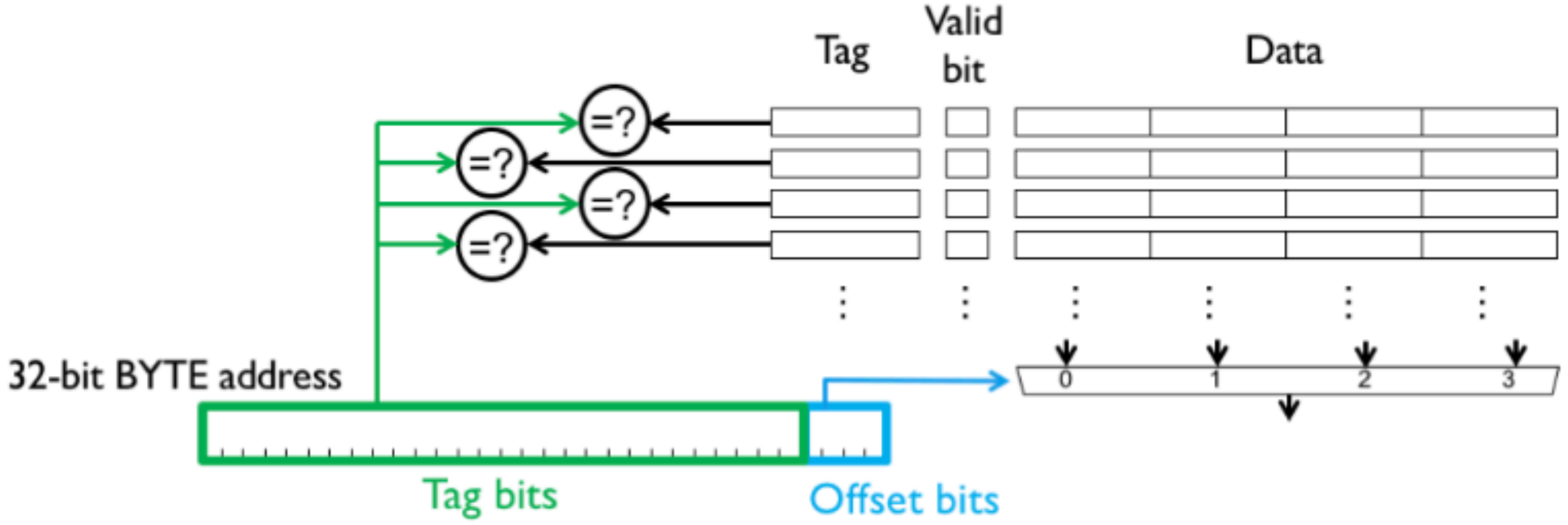
Direct Mapped Cache

(Source: 6.1910)



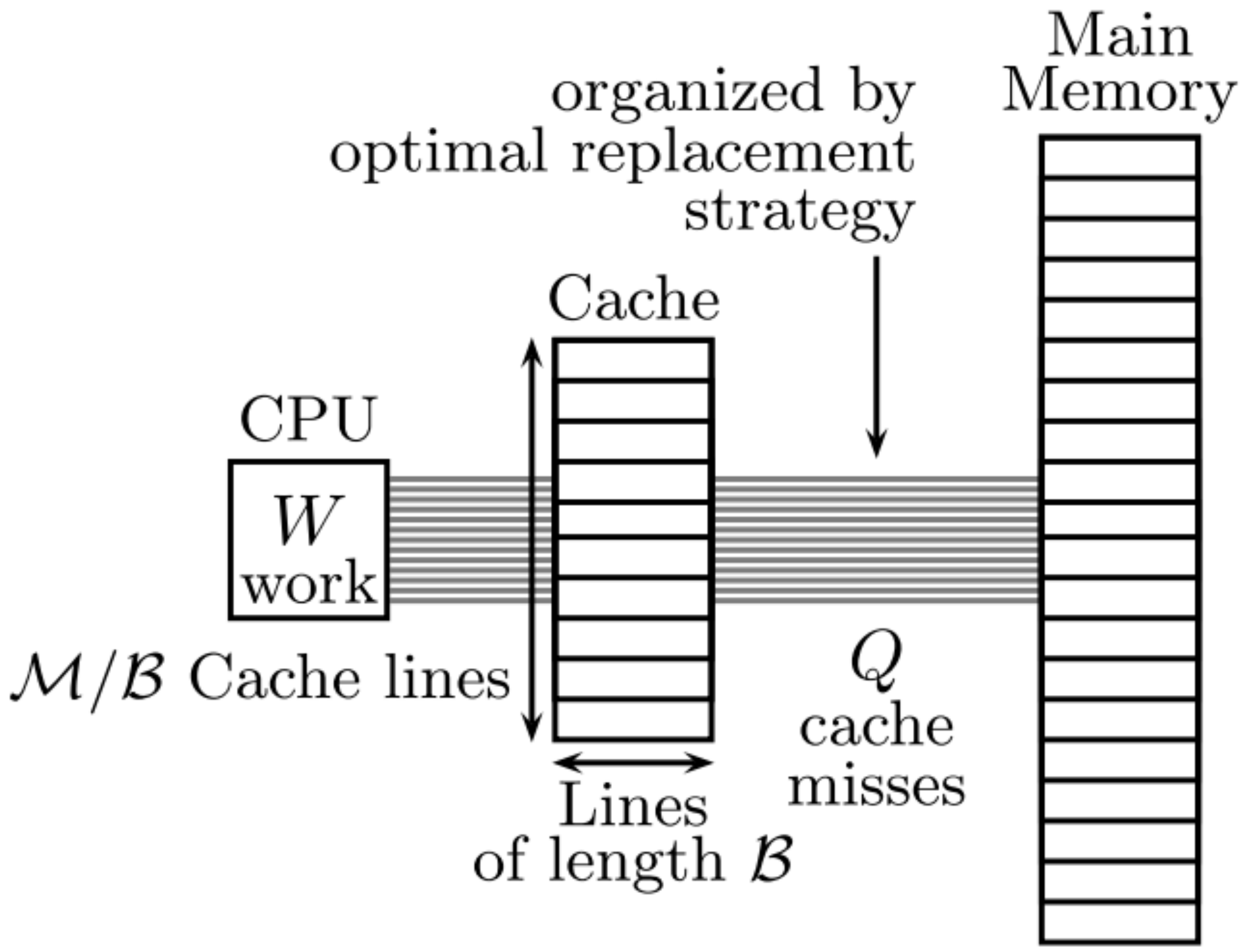
- Each memory location maps to a specific line

Fully Associative Cache
(Source: 6.1910)



- Each memory location can map to any specific line (must compare tags of all entries to find a matching one)
- Most of the times find a balance between direct mapped and fully associative of n-way associative

Ideal Cache Model



- Two-level memory hierarchy with an ideal data cache of M words and arbitrarily large main memory. Each line can store B consecutive words
- The cache is fully associative (blocks can be stored anywhere) and uses optimal *offline* strategy for eviction. It replaces cache block farthest away in future.
- Additionally, assume that Cache is tall $M = \Omega(B^2)$

Measures for Ideal Cache Model

- Algorithm of input size n is measured by $W(n)$, conventional running time
- Cache complexity of $Q(n; M, B)$ which is number of cache misses algorithm incurs as function of size M and line length B of ideal cache
- Algorithm is **cache aware** if it contains parameters that can be tuned to optimize cache complexity (for a particular M, B)
- If not, it is **cache oblivious**

Historical Background / Relevant Works

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 Blumofe et al. [1996]. Shortly after leaving our research group, Toledo [1997] independently proposed a cache-oblivious algorithm for LU-decomposition with pivoting. For $n \times n$ matrices, Toledo's algorithm uses $\Theta(n^3)$

ARTICLE | FREE ACCESS

I/O complexity: The red-blue pebble game

Authors: Hong Jia-Wei, H. T. Kung Authors Info & Claims

STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing • Pages 326 - 333
https://doi.org/10.1145/800076.802486

Published: 11 May 1981 Publication History Check for updates

Proved lower bounds on I/O complexity of matrix multiplication, FFT.

Model used temporal locality using two levels of memory but no caches.

Later inspired the ideal cache model

Cache-Aware Matrix Multiplication

- A, B, C are n x n and stored in row major order
- Assume n is big and n > B
- Ord-Mult computes block multiplications in $O(s^3)$and then does $C_{ij} \leftarrow C_{ij} + A_{ik}B_{kj}$

ALGORITHM: TILED-MULT(A, B, C, n)

```
for i ← 1 to n/s
    do for j ← 1 to n/s
        do for k ← 1 to n/s
            do ORD-MULT(A_ik, B_kj, C_ij, s)
```

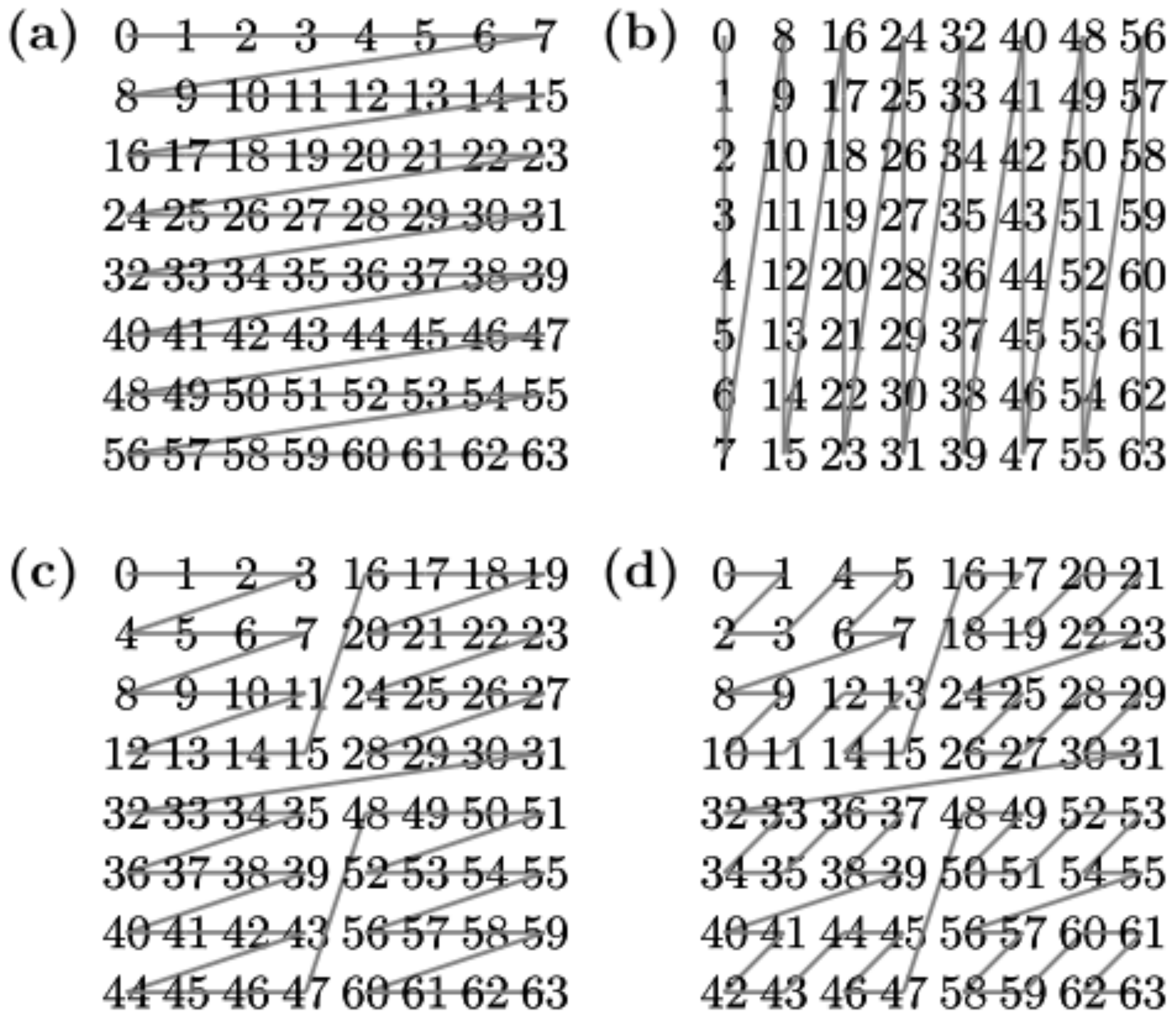


Fig. 2. Layout of a 16×16 matrix in (a) row major, (b) column major, (c) 4×4-tiled, and (d) bit-interleaved layouts.

Cache-Aware Matrix Multiplication

Cache Complexity Analysis

- Choose s greatest so that 3 s x s matrixes can fit in cache.
- s x s submatrix can fit in $\Theta(s + \frac{s^2}{B})$ cache lines
- Each Ord-Mult call has at most $\Theta(\frac{s^2}{B})$ misses. This is $\Theta(M/B)$.
- Final cache complexity is

$$\Theta(1 + n^2/B + (n/\sqrt{M})^3(M/B))$$
$$= \Theta(1 + n^2/B + n^3/B\sqrt{M})$$

ALGORITHM: TILED-MULT(A, B, C, n)

```
for i ← 1 to n/s
    do for j ← 1 to n/s
        do for k ← 1 to n/s
            do ORD-MULT(A_ik, B_kj, C_ij, s)
```

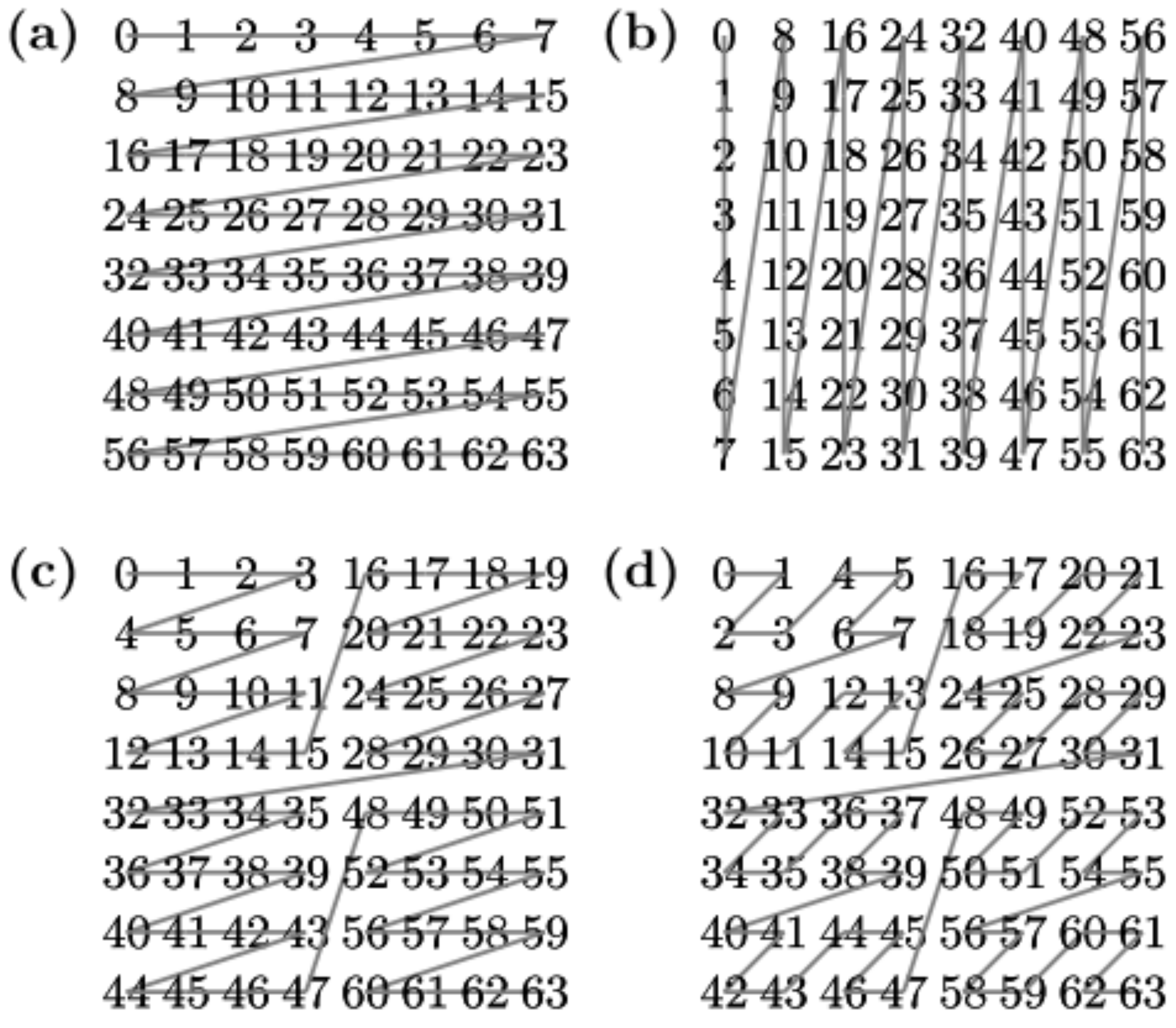


Fig. 2. Layout of a 16×16 matrix in (a) row major, (b) column major, (c) 4×4-tiled, and (d) bit-interleaved layouts.

Cache-Oblivious Matrix Multiplication

- Cache oblivious algorithm *Rec-Mult* for multiplying matrix A which is $m \times n$ and matrix B which is $n \times p$. Let final matrix M be $m \times p$
- If m, n, p = 1: $C \leftarrow C + AB$ (scalar multiply add)
- If $m \geq \max\{n, p\}$, $\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$.
- If $n \geq \max\{m, p\}$, $C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$.
- If $p \geq \max\{m, n\}$, $\begin{pmatrix} C_1 & C_2 \end{pmatrix} = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$.
- (Each of the cases recurse in a subproblem) $C \leftarrow C + A_1 B_1$ then $C \leftarrow C + A_2 B_2$ to avoid storing intermediate states

Cache-Oblivious Matrix Multiplication

Cache Complexity Analysis

- Assume that matrices A, B, M are stored in row-major order.
- Make tall-Cache assumption $M = \Omega(B^2)$
- Has no tuning parameters (cache-oblivious) but can prove uses cache optimally

THEOREM 2.1. *The* REC-MULT *algorithm uses* $\Theta(mnp)$ *work and incurs* $\Theta(m + n + p + (mn + np + mp)/\mathcal{B} + mnp/\mathcal{B}\sqrt{\mathcal{M}})$ *cache misses when multiplying an* $m \times n$ *matrix by an* $n \times p$ *matrix.*

Proof of Theorem 2.1

Cache Complexity Analysis

THEOREM 2.1. *The* REC-MULT *algorithm uses* $\Theta(mnp)$ *work and incurs* $\Theta(m + n + p + (mn + np + mp)/\mathcal{B} + mnp/\mathcal{B}\sqrt{\mathcal{M}})$ *cache misses when multiplying an* $m \times n$ *matrix by an* $n \times p$ *matrix.*

- Proof of work is simple. For complexity, let $\alpha' > 0$ be the largest constant sufficiently small s.t. for submatrices $m' \times n', n' \times p', m' \times p'$ where $\max\{m', n', p'\} \leq \alpha\sqrt{M}$, they all fit in the cache.

- Case I: $m', n', p' > \alpha\sqrt{M}$ (matrices do not fit in cache)

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

Technically, algorithm recurses more but for cache miss analysis, can terminate immediately once they all fit in cache

Proof of Theorem 2.1

Cache Complexity Analysis

THEOREM 2.1. *The* REC-MULT *algorithm uses* $\Theta(mnp)$ *work and incurs* $\Theta(m + n + p + (mn + np + mp)/\mathcal{B} + mnp/\mathcal{B}\sqrt{\mathcal{M}})$ *cache misses when multiplying an* $m \times n$ *matrix by an* $n \times p$ *matrix.*

- Case I: $m', n', p' > \alpha\sqrt{M}$ (matrices do not fit in cache)

$$Q(m, n, p) \le \begin{cases} \Theta((mn + np + mp)/\mathcal{B}) & \text{if } m, n, p \in [\alpha\sqrt{\mathcal{M}}/2, \alpha\sqrt{\mathcal{M}}] \ , \\ 2Q(m/2, n, p) + O(1) & \text{otherwise if } m \ge n \text{ and } m \ge p \ , \\ 2Q(m, n/2, p) + O(1) & \text{otherwise. if } n > m \text{ and } n \ge p \ , \\ 2Q(m, n, p/2) + O(1) & \text{otherwise} \ . \end{cases}$$

Technically, algorithm recurses more but for cache miss analysis, can terminate immediately once they all fit in cache

This recurrence evaluates to $Q(m, n, p) = \Theta(mnp/B\sqrt{M})$ and similarly can do three other cases for m', n', p' to prove the theorem

O(1) for keeping track of subarrays

For square matrices, we get the bound $Q(n) = \Theta(n + n^2/\mathcal{B} + n^3/\mathcal{B}\sqrt{\mathcal{M}})$ which is the same as the cache oblivious algorithm (which we saw)

Cache Oblivious Distribution Sort Algorithm

- Cache Oblivious algorithm that does $O(n \log n)$ work to sort n elements and $O(1 + (n/B)(1 + \log_M n))$ cache misses

(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) $\max\{x \mid x \in B_i\} \leq \min\{x \mid x \in B_{i+1}\}$ for $i = 1, 2, \ldots, q-1$.
    (b) $n_i \leq 2\sqrt{n}$ for $i = 1, 2, \ldots, q$.
    (See below for details.)
(3) Recursively sort each bucket.
(4) Copy the sorted buckets to array A.

Guarantee all elements in B_{i+1} is greater than or equal to all elements in B_i (can define a pivot / upper bound for B_i)

Cache Oblivious Distribution Sort Algorithm

Distribution

(2) Distribute the sorted subarrays into q buckets $B_1, \ldots, B_q$ of size $n_1, \ldots, n_q$, respectively, such that
(a) $\max\{x \mid x \in B_i\} \leq \min\{x \mid x \in B_{i+1}\}$ for $i = 1, 2, \ldots, q-1$.
(b) $n_i \leq 2\sqrt{n}$ for $i = 1, 2, \ldots, q$.
(See below for details.)

- Each subarray has index *next* (next element to read) and *bnum* (bucket to put next element)
- Each bucket B_i has a pivot which is greater than all elements in the bucket
- Initially there is one bucket with a pivot of ∞
- If a bucket gets bigger than $2\sqrt{n}$, the bucket is split into two where each size is at least $\sqrt{n}$
- The lower half gets a new smaller pivot (using median finding algorithm) and upper half keeps the original pivot

Cache Oblivious Distribution Sort Algorithm

Distribution

- Distribute(i,j,m) distributes elements from subarray i,i+1,i+2 … i+m-1 into buckets starting from b_j
- Given precondition: subarray i,i+1,i+2 … i+m-1 have bnum >= j
- Satisfies postcondition: subarray i,i+1,i+2 … i+m-1 have bnum>=j+m

ALGORITHM: DISTRIBUTE(i, j, m)

```
if m = 1
   then CopyElems(i, j)
   else Distribute(i, j, m/2)
        Distribute(i + m/2, j, m/2)
        Distribute(i, j + m/2, m/2)
        Distribute(i + m/2, j + m/2, m/2)
```

Copies all elements in subarray i that can be placed to bucket j

Can show inductively that Distribute(1, 1, $\sqrt{N}$) meets the post condition and hence distributes all the subarrays as desired

Cache Oblivious Distribution Sort Algorithm

Distribution

LEMMA 5.2. *The distribution step involves $O(n)$ work, incurs $O(1 + n/\mathcal{B})$ cache misses, and uses $O(n)$ stack space to distribute n elements.*

- For cache analysis can prove case by case by using α, constant sufficiently small s.t. the space used by a sorting problem of size αM, including the input array, fits completely in cache.

THEOREM 5.3. *Distribution sort uses $O(n \lg n)$ work and incurs $O(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$ cache misses to sort n elements.*

- Using α,can prove bound on cache misses using lemma 5.2 and solving the recurrence

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

(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 $i = 1, 2, \ldots, q-1$.
  (b) $n_i \le 2\sqrt{n}$ for $i = 1, 2, \ldots, q$.
  (See below for details.)
(3) Recursively sort each bucket.
(4) Copy the sorted buckets to array A.

Empirical Results

- Evaluated matrix transpose and matrix multiplication (show cache oblivious algorithms can reach high performance)
- Equipment: 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

(Lacks temporal locality)

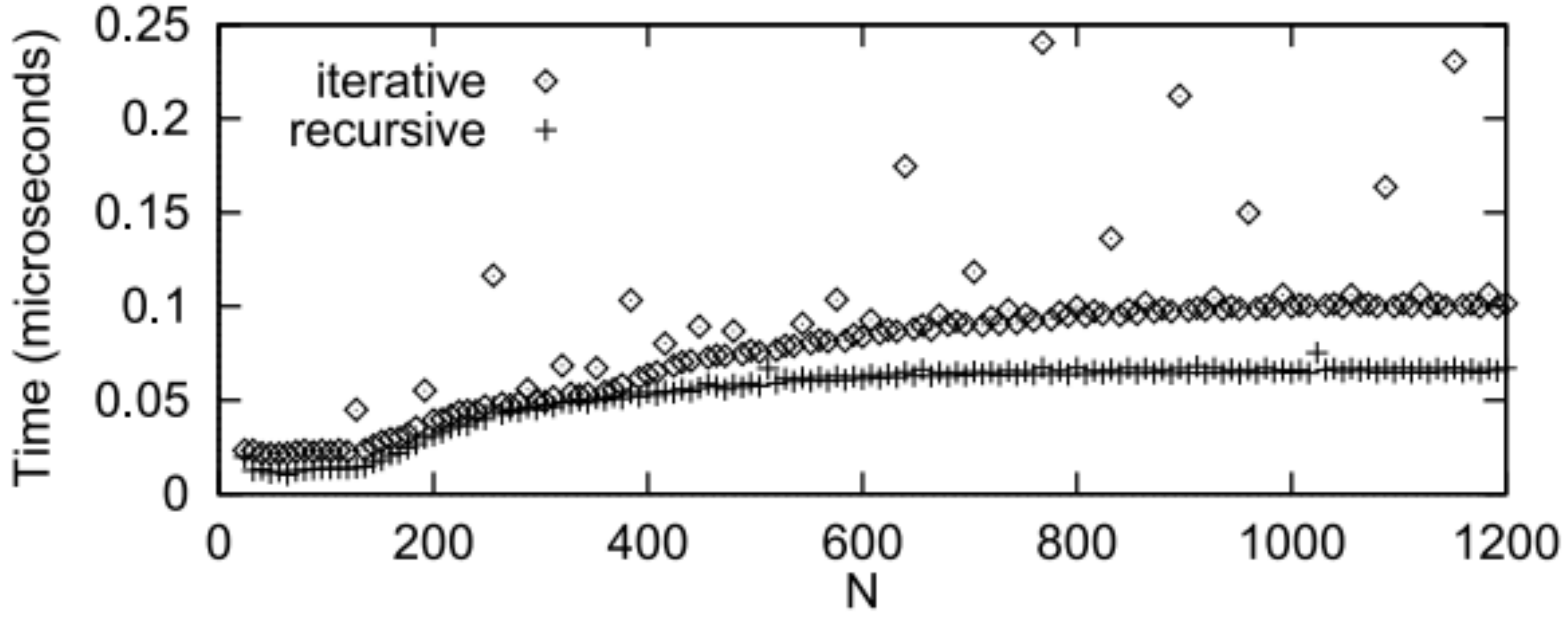


Fig. 4. Average time to transpose an $N \times N$ matrix, divided by N^2.

(High degree of temporal locality)

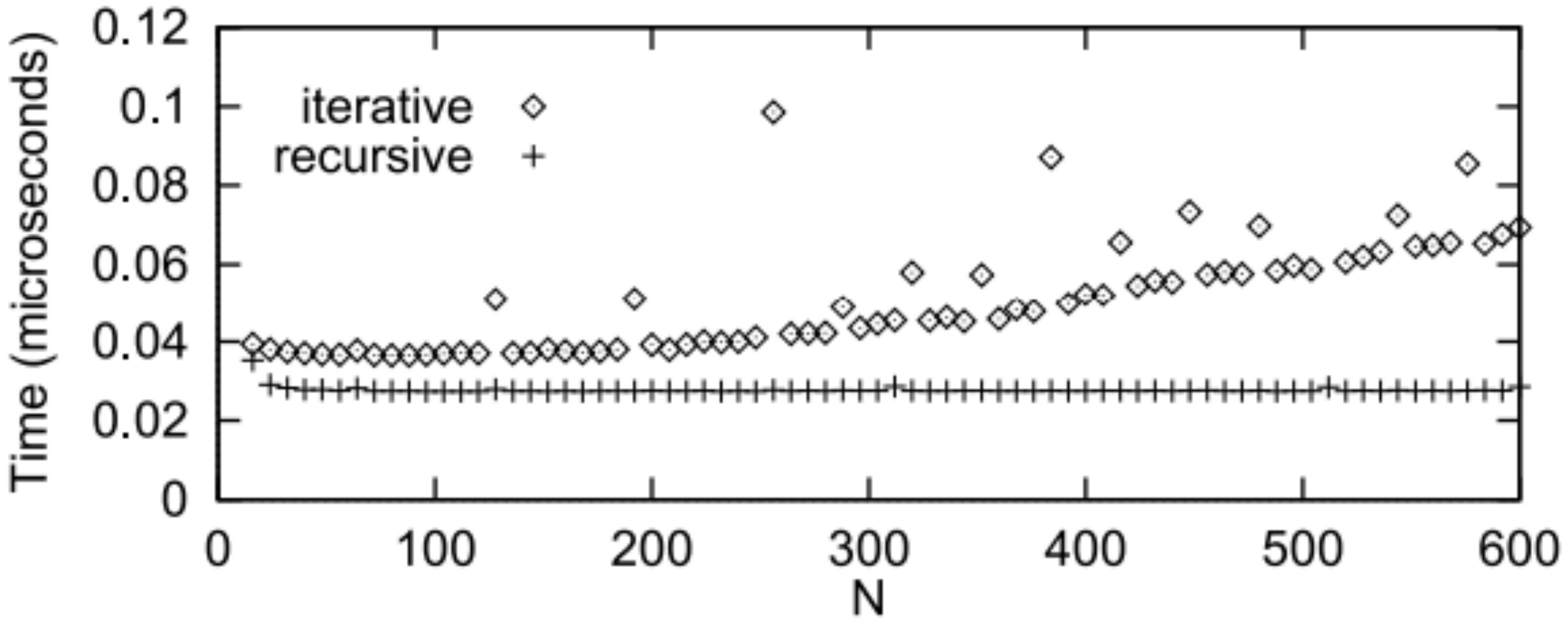
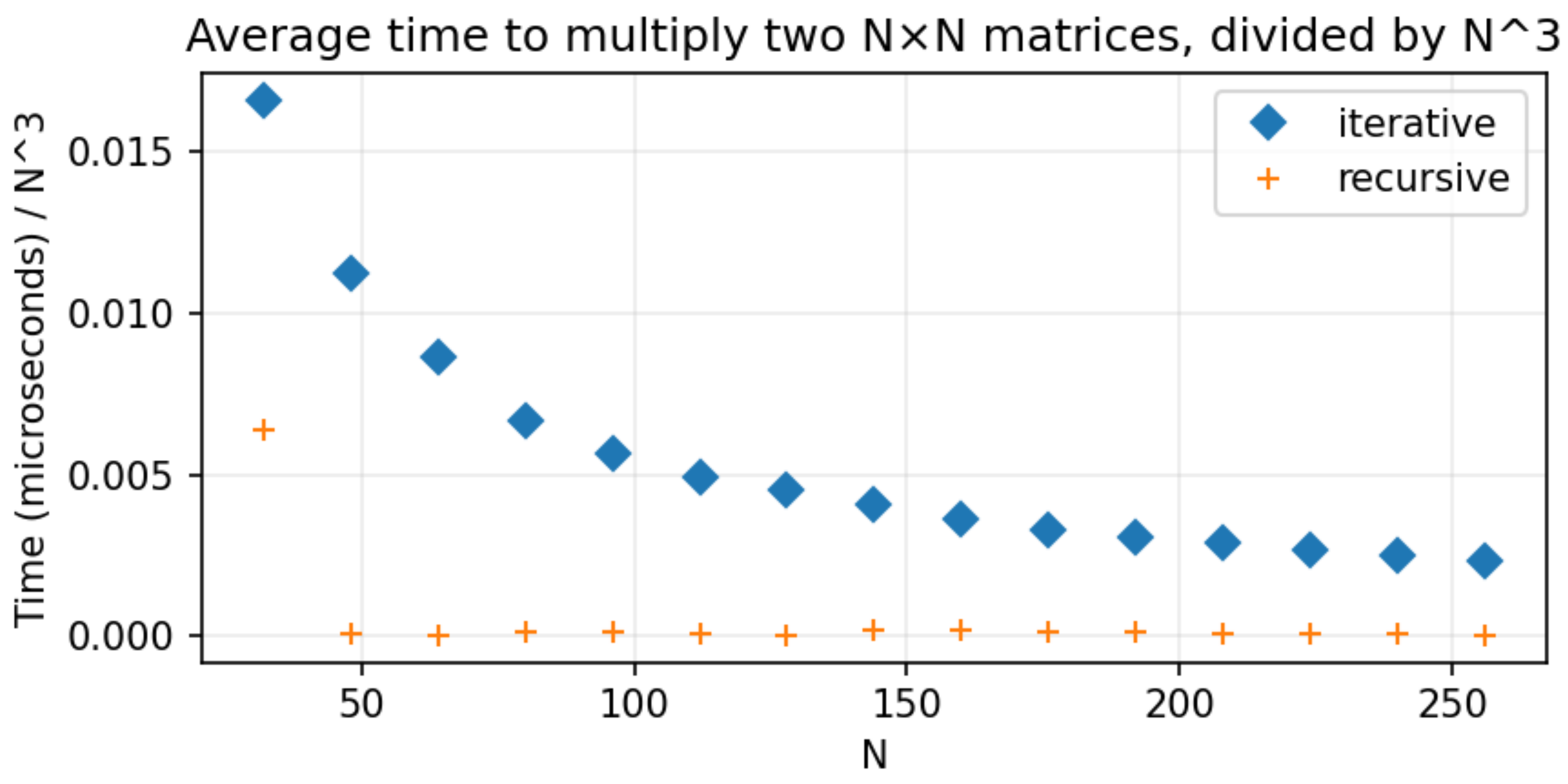


Fig. 5. Average time taken to multiply two $N \times N$ matrices, divided by N^3.

Empirical Results (on local MacBook Pro)



Using Numpy

Possible Directions, Strengths, Weaknesses

- Is it possible to integrate the ideal cache model with I/O to consider for both caches and primary and secondary storage?
- The ideal cache model makes restricting assumptions of (could some be relieved?)
  - Optimal replacement
  - Exactly two levels of memory
  - Automatic replacement
  - Full associativity
- The ideal cache model also provides theoretical guarantees

THEOREM 6.6. *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.*

e.g. SM shared memory / scratch pad for GPUs