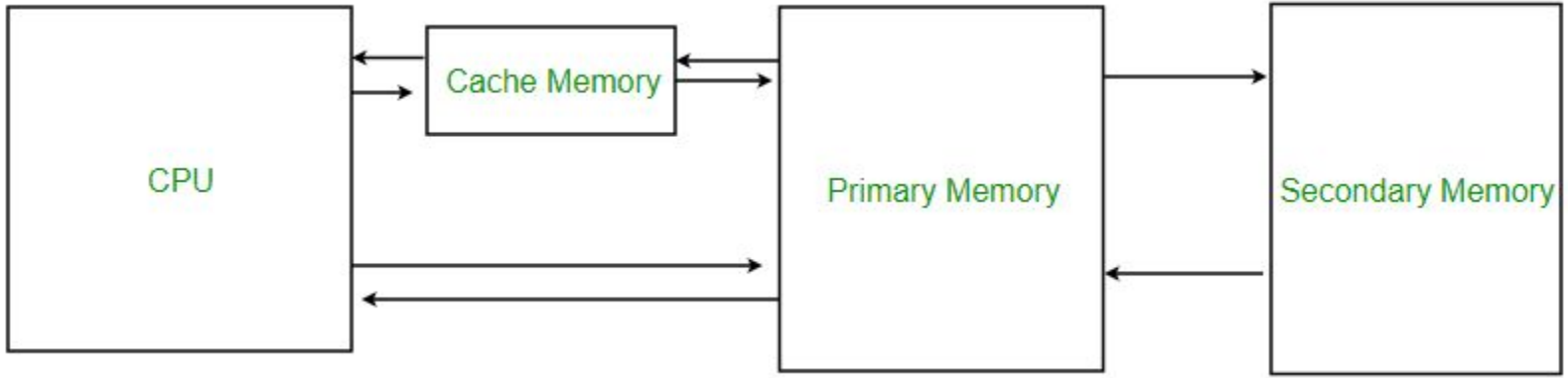
Cache-Oblivious Algorithms

Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran (2012)
Presentation by Ricardo Gayle Jr.

What is a cache?

- supplementary memory system that temporarily stores frequently used instructions and data for quicker processing by the central processing unit (CPU) of a computer.
- Cache holds a copy of only the most frequently used information or program codes stored in the main memory.



Cache oblivious vs Cache aware

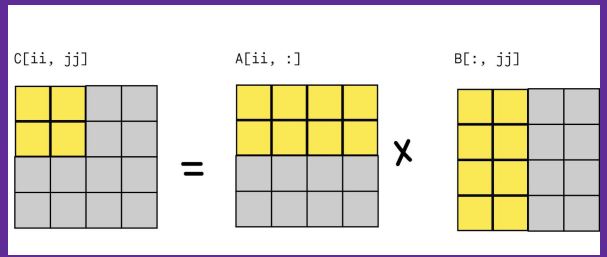
Algorithm does not use any prior knowledge of the machine's cache to be optimized

Algorithms are simpler and more portable

Ex: to be seen

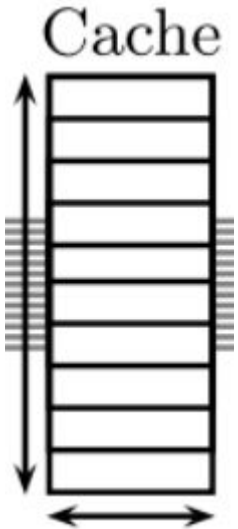
Uses the knowledge of the cache size and length of cache blocks of the machine to optimize cache complexity

Ex: TILED-MULT [Golub and van Loan 1989]



Cache Terminology

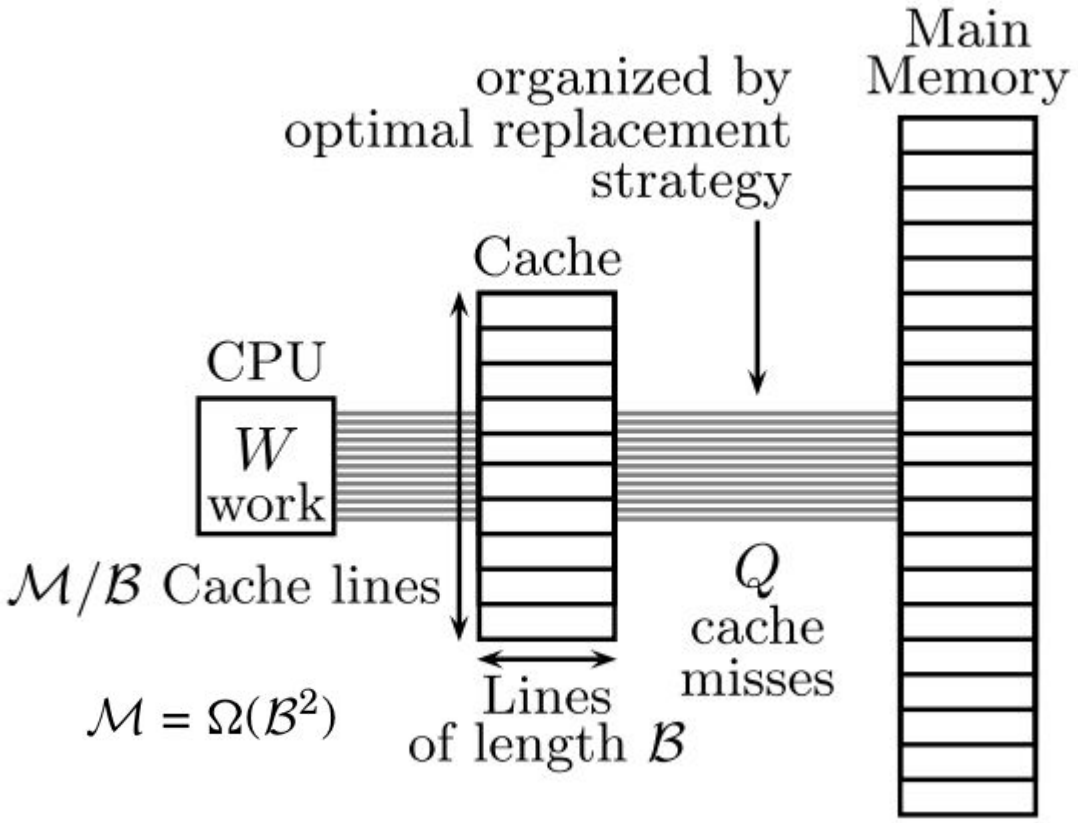
How do we talk about caches?



- M : number of words that can be stored in the cache
- The cache is partitioned into cache lines
- B : the number of consecutive words stored in a cache line
- There are M / B cache lines
- Cache lines always move together between memory
- Q : number of cache misses it incurs as function of M and B
    - A cache-oblivious algorithm is “good” if it uses cache as effectively as a cache aware algorithm

Ideal-Cache Model

- Two-level memory hierarchy
  - Ideal and tall cache
  - Arbitrarily large main memory
- Processor can only use words in the cache
  - Cache hit vs miss
  - Full cache
- What makes it ideal?
  - Fully associative
  - Optimal offline strategy
- Who cares?
  - Model used to analyze an algorithm's cache complexity



Why does the Ideal Cache Model Work? How do we use it?

1. Optimal Replacement
    a. Proven that the number of misses on a (M, B) LRU-cache is at most twice as many on a (M/2, B) ideal-cache
2. Two levels of memory hierarchy
    a. Proof by induction which essentially shows that multi-level LRU caches incur an optimal number of misses
3. Fully associative and automatic replacement
    a. Proven that a fully associative LRU cache can be maintained in memory with no asymptotic loss in performance

The model provides an easy way to examine the cache complexity (Q) of algorithms.

Algorithms

Driving Force: Produce efficient algorithms that work with any computer’s hierarchical memory system

Matrix Multiplication

Problem:
Multiplying an
m x n matrix by an n x p matrix

$W = \Theta(mnp)$

$Q = \Theta(m + n + p + (mn + np + mp)/B + mnp/B\sqrt{M})$

Methodology

If $m = n = p = 1$, REC-MULT performs the scalar multiply-add $C \leftarrow C + AB$. Otherwise, depending on the relative sizes of m, n, and p, we have three cases.

(1) If $m \geq \max\{n, p\}$, we split the range of m according to the formula

$$\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} . \tag{2}$$

The algorithm recurs twice to compute $C_1 = C_1 + A_1 B$ and $C_2 = C_2 + A_2 B$.

(2) If $n \geq \max\{m, p\}$, we split the range of n according to the formula

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 . \tag{3}$$

Specifically, the algorithm first computes $C \leftarrow C + A_1 B_1$ recursively, and then it computes $C \leftarrow C + A_2 B_2$, also recursively. In particular, we do not allocate temporary storage for the intermediate products implied by Eq. (3).

(3) If $p \geq \max\{m, n\}$, we split the range of p according to the formula

$$\begin{pmatrix} C_1 & C_2 \end{pmatrix} = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} A B_1 & A B_2 \end{pmatrix} . \tag{4}$$

Cache Complexity Analysis

Let $\alpha > 0$ be the largest constant small enough where $max\{m, n, p\} \leq \alpha\sqrt{M}$

- Case 1: $m,n,p \leq \alpha\sqrt{M}$

$$Q(m, n, p) = \Theta(1 + (mn + np + mp)/\mathcal{B})$$

- Case 2: $n,p > \alpha\sqrt{M}$

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

$$Q(m, n, p) = \Theta(np/\mathcal{B} + mnp/\mathcal{B}\sqrt{\mathcal{M}})$$

- Case 3: $m > \alpha\sqrt{M}$

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

$$Q(m, n, p) = \Theta(m + mnp/\mathcal{B}\sqrt{\mathcal{M}})$$

- Case 4: $m,n,p > \alpha\sqrt{M}$

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

$$Q(m, n, p) = \Theta(mnp/\mathcal{B}\sqrt{\mathcal{M}})$$

$Q = \Theta(m + n + p + (mn + np + mp)/B + mnp/B\sqrt{M})$

Funnelsort

Problem:
Sort a list of n items

$W = O(n \lg n)$

$Q = O(1 + (n/B)(1 + \log_M n))$

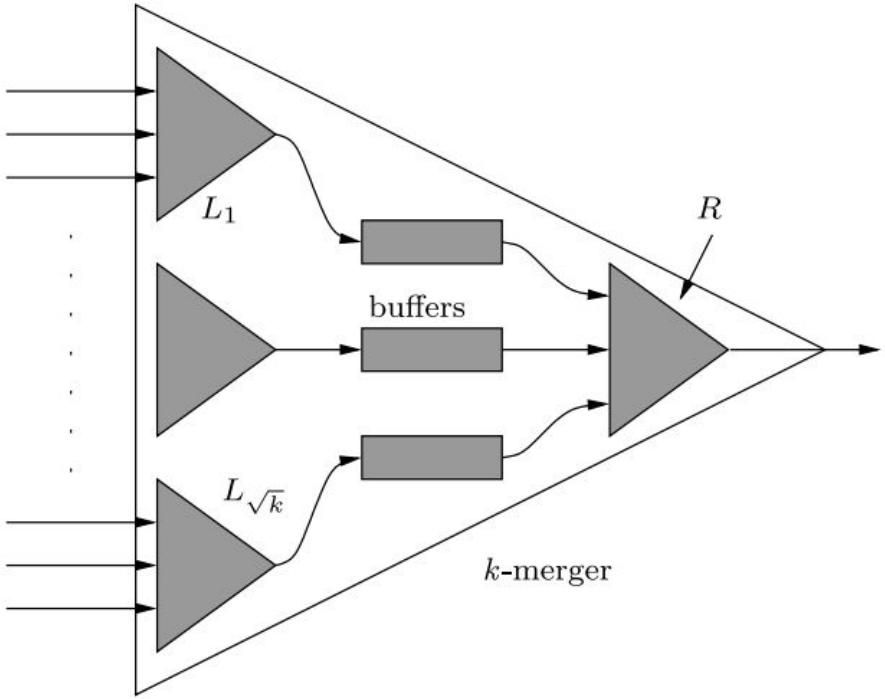
Methodology

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

Uses a k-merger built out of √k recursive √k-mergers with FIFO queue buffers

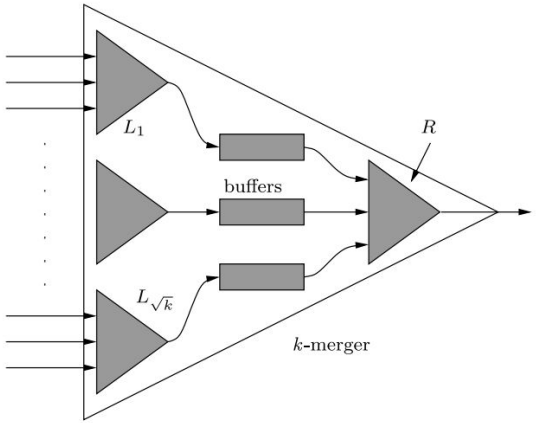
Buffers are oversized at size $2k^{3/2}$, twice number of elements outputted by the √k merger

- Performing inserts and removes on a circular queue can cause a small (O(1 + r/B)) number of cache misses if two cache lines are available for the buffer



Cache Complexity Analysis

- $S(k) \leq (\sqrt{k} + 1)S(\sqrt{k}) + O(k^2) = O(k^2)$
  - *Merger requires* $O(k^2)$ space for buffers and each of the smaller mergers
- $Q_{merge}(k) = O(1 + k + k^3/B + (k^3 \log_M k)/B$
  - Proven through proof by cases where a subproblem can and can not fit into the cache



To sort *n* elements:

- If $n < \alpha M$, the biggest k-merger is the $n^{1/3}$-merger, so $S(n) = O((n^{1/3})^2) = O(n)$ and thus only $Q(n) = O(1 + n/B)$.
- If $n > \alpha M$,

$$Q(n) = n^{1/3}Q(n^{2/3}) + Q_{merge}(n^{1/3})$$

$$= O(1 + (n/B)(1 + \log_M n))$$

- It's been proven that this upper bound is also the lower bound for the number of cache misses needed for any sorting algorithms

Other Algorithms

Divide and Conquer is Key!

Matrix Transposition and FFT

- Problem: Transpose an *m x n* matrix
- Solution: Recursively break the matrix into submatrices
- $Q(m, n) = O(1 + mn/B)$

Distribution Sort

- Problem: Sort *n* items
- Solution: uses a “bucket splitting” technique to select pivots is a distribution sort-- also divide and conquer
- $Q(n) = O(1 + (n/B)(1 + \log_M n))$

Competitive Results

The tests were run 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.

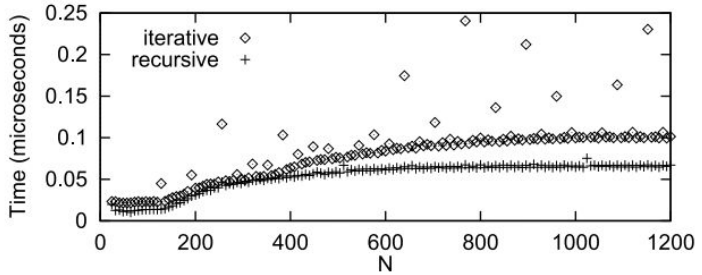


Fig. 4. Average time to transpose an $N \times N$ matrix, divided by N^2.

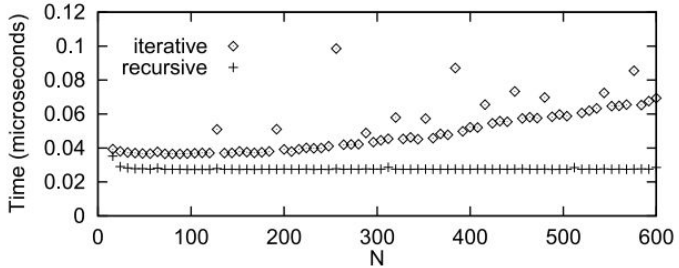


Fig. 5. Average time taken to multiply two $N \times N$ matrices, divided by N^3.

Thank you!
Questions and comments?