# Cache Complexity (March 8 version)

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 4435 - CS 9624

# Plan

# Plan

# The $(Z, L)$ ideal cache model (1/4)

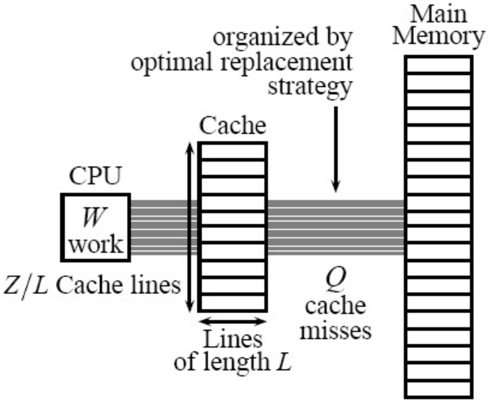



**Figure 1:** The ideal-cache model

# The $(Z, L)$ ideal cache model (2/4)

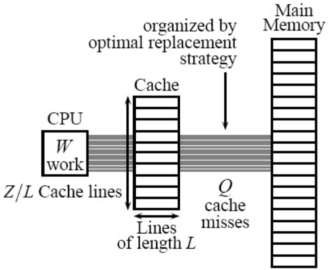



**Figure 1:** The ideal-cache model

- Computer with a **two-level memory hierarchy**:
  - an ideal (data) cache of $Z$ words partitioned into $Z/L$ *cache lines*, where $L$ is the number of words per cache line.
  - an arbitrarily large main memory.
- Data moved between cache and main memory are always cache lines.
- The cache is **tall**, that is, $Z$ is much larger than $L$, say $Z \in \Omega(L^2)$.

# The $(Z, L)$ ideal cache model (3/4)

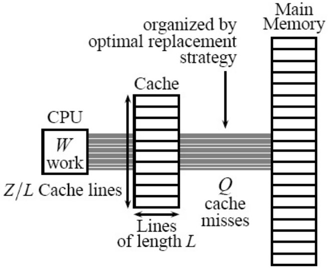



**Figure 1:** The ideal-cache model

- 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 $(Z, L)$ ideal cache model (4/4)

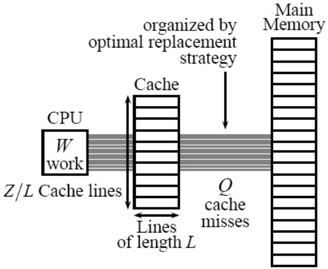



**Figure 1:** The ideal-cache model

- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacing** the cache line whose next access is furthest in the future, and thus it exploits temporal locality perfectly.

# Cache complexity

- For an algorithm with an input of size $n$, he ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.
  - the **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 simply write $Q(n)$ instead of $Q(n; Z, L)$.
- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.
- Otherwise the algorithm is **cache oblivious**.

# Cache complexity of the naive matrix multiplication

```
// A is stored in row-major and B in column-major
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

- Assuming $Z \geq 3L$, computing each `C[i][j]` incurs $O(1 + n/L)$ caches misses.
- If $Z$ large enough, say $Z \in \Omega(n)$ then the row $i$ of $A$ will be remembered for its entire involvement in computing $C$.
- For a column of $B$ to be remembered when necessary one needs $Z \in \Omega(n^2)$ in which case the whole computation fits in cache. Therefore, we have

$$Q(n, Z, L) = \begin{cases} O(n^2 + n^3/L) & \text{if} \quad 3L \leq Z < n^2, \\ O(n + n^2/L) & \text{if} \quad 3n^2 \leq Z. \end{cases}$$

# A cache-aware matrix multiplication algorithm (1/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Each matrix $M \in \{A, B, C\}$ consists 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.
- Assume $s$ divides $n$ to keep the analysis simple.
- `blockMult(A,B,C,i,j,k,s)` computes $C_{ij} = A_{ik} \times B_{kj}$ using the naive algorithm

# A cache-aware matrix multiplication algorithm (2/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- We choose $s$ to be the largest value such that the three $s \times s$ submatrices simultaneously fit in cache, that is, $Z \in \Theta(s^2)$.
- An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines.
- From the call cache assumption ($Z \in \Omega(L^2)$), we have $s \in \Theta(\sqrt{Z})$.
- Thus `blockMult(A,B,C,i,j,k,s)` runs within $Z/L \in \Theta(s^2/L)$ cache misses.
- Initializing the $n^2$ elements of $C$ amounts to $\Theta(1 + n^2/L)$ caches misses. Therefore we have

$$Q(n, Z, L) \in \Theta(1+n^2/L+(n/\sqrt{Z})^3(Z/L)) = \Theta(1+n^2/L+n^3/(L\sqrt{Z})).$$

# Plan

# Scanning

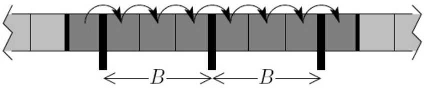



**Figure 2.** Scanning an array of $N$ elements arbitrarily aligned with blocks may cost one more memory transfer than $\lceil N/B \rceil$.

**Scanning $n$ elements stored in a contiguous segment (= cache lines) of memory costs at most $\lceil n/L \rceil + 1$ cache misses.** Indeed:

- In the above figure $N = n$ and $B = L$.
- The main issue here is alignment and we focus on the worst case.
- In the worst case, each of the first and the last read cache lines contains less than $L$ "useful" elements.
- If $L$ does not divide $n$, there are $\lfloor n/L \rfloor$ fully useful cache lines.
- If $L$ divides $n$, there are at most $\frac{n}{L} - 1$ fully useful cache lines.

# Array reversal

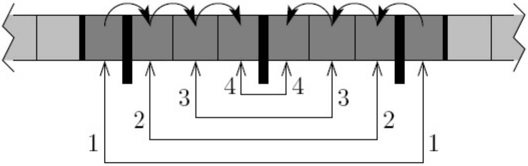



**Figure 3.** Bentley's reversal of an array.

**Reversing an array of $n$ elements stored in a contiguous segment ($=$ cache lines) of memory costs at most $\lceil n/L \rceil + 1$ cache misses, provided that $Z \geq 2L$ holds.** Exercise!

# Median and selection (1/8)

- A selection algorithm is an algorithm for finding the $k$-th smallest number in a list. This includes the cases of finding the minimum, maximum, and median elements.
- A worst-case linear algorithm for the general case of selecting the $k$-th largest element was published by Blum, Floyd, Pratt, Rivest, and Tarjan in their 1973 paper *Time bounds for selection*, sometimes called BFPRT.
- The principle is the following:
  - Find a *pivot* that allows splitting the list into two parts of nearly equal size such that
  - the search can continue in one of them.

# Median and selection (2/8)

```
select(L,k)
{
if (L has 10 or fewer elements)
{
    sort L
    return the element in the kth position
}

partition L into subsets S[i] of five elements each
    (there will be n/5 subsets total).

for (i = 1 to n/5) do
    x[i] = select(S[i],3)

M = select({x[i]}, n/10)

partition L into L1<M, L2=M, L3>M
if (k <= length(L1))
    return select(L1,k)
else if (k > length(L1)+length(L2))
    return select(L3,k-length(L1)-length(L2))
else return M
```

# Median and selection (3/8)

For an input list of $n$ elements, the number $T(n)$ of comparisons satisfies

$$T(n) \leq 12n/5 + T(n/5) + T(7n/10).$$

- We always throw away either `L3` (the values greater than `M`) or `L1` (the values less than `M`). Suppose we throw away `L3`.
- Among the $n/5$ values `x[i]`, $n/10$ are larger than `M`, since `M` was defined to be the median of these values.
- For each `i` such that `x[i]` is larger than `M`, two other values in `S[i]` are also larger than `x[i]`
- So `L3` has at least $3n/10$ elements. By a symmetric argument, `L1` has at least $3n/10$ elements.
- Therefore the final recursive call is on a list of at most $7n/10$ elements and takes time at most $T(7n/10)$.

# Median and selection (4/8)

How to solve

$$T(n) \leq 12n/5 + T(n/5) + T(7n/10)?$$

- We “try” $T(n) \leq c\, n$ by induction. The substitution gives

  $$T(n) \leq n\,(12/5 + 9c/10).$$

  From $n(12/5 + 9c/10) \leq c\, n$ we derive $c \leq 24$.
- The tree-based method also brings $T(n) \leq 24n$.
- The same tree-expansion method then shows that, more generally, if $T(n) \leq cn + T(an) + T(bn)$, where $a + b < 1$, the total time is $c(1/(1 - a - b))n$.
- With a lot of work one can reduce the number of comparisons to $2.95n$ [D. Dor and U. Zwick, *Selecting the Median*, 6th SODA, 1995].

# Median and selection (5/8)

In order to analyze its cache complexity, let us review the algorithm and consider an array instead of a list.

**Step 1:** Conceptually partition the array into $n/5$ quintuplets of five adjacent elements each.

**Step 2:** Compute the median of each quintuplet using $O(1)$ comparisons.

**Step 3:** Recursively compute the median of these medians (which is not necessarily the median of the original array).

**Step 4:** Partition the elements of the array into three groups, according to whether they equal, or strictly less or strictly greater than this median of medians.

**Step 5:** Count the number of elements in each group, and recurse into the group that contains the element of the desired rank.

# Median and selection (6/8)

To make this algorithm cache-oblivious, we specify how each step works in terms of memory layout and scanning. We assume that $Z \geq 3L$.

**Step 1:** Just conceptual; no work needs to be done.

**Step 2:** requires two parallel scans, one reading the array 5 elements at a time, and the other writing a new array of computed medians, incurring $\Theta(1 + n/L)$.

**Step 3:** Just a recursive call on size $n/5$.

**Step 4:** Can be done with three parallel scans, one reading the array, and two others writing the partitioned arrays, incurring again $\Theta(1 + n/L)$.

**Step 5:** Just a recursive call on size $7n/10$.

This leads to

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(1 + n/L).$$

# Median and selection (7/8)

How to solve

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(1 + n/L)?$$

The unknown is what is the **base-case**?

- Suppose the base case id $T(0(1)) \in O(1)$.
- Following the proof of the *Master Theorem* we estimate the number of leaves $L(n) = n^c$ and obtain in $L(n) = L(n/5) + L(7n/10)$, $L(1) = 1$, which brings

$$\left(\frac{1}{5}\right)^c + \left(\frac{7}{10}\right)^c = 1$$

  leading to $c \simeq 0.8397803$.
- Since each leaf incurs a constant number of cache misses we have $T(n) \in \Omega(n^c)$.

# Median and selection (8/8)

How to solve

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(1 + n/L)?$$

- Fortunately, we have a better **base-case**: $T(0(L)) \in O(1)$.
- Indeed, once the problem fits into $O(1)$ cache-lines, all five steps incur only a constant number of cache misses.
- Thus we have only $(n/L)^c$ leaves in the recursion tree.
- In total, these leaves incur $O((n/L)^c) = o(n/L)$ cache misses.
- In fact, the cost per level decreases geometrically from the root, so the total cost is the cost of the root. Finally we have

$$T(n) \in \Theta(1 + n/L)$$

# Plan

# A matrix transposition cache-oblivious algorithm (1/3)

- **Matrix transposition problem:** Given an $m \times n$ matrix $A$ 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.
- We describe a recursive cache-oblivious algorithm which uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
- The straightforward algorithm employing doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $m \gg Z/L$ and $n \gg Z/L$.

# A matrix transposition cache-oblivious algorithm (2/3)

- If $n \geq m$, the $\textsc{Rec-Transpose}$ algorithm partitions

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

  and recursively executes $\textsc{Rec-Transpose}(A_1, B_1)$ and $\textsc{Rec-Transpose}(A_2, B_2)$.

- If $m > n$, the $\textsc{Rec-Transpose}$ algorithm partitions

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

  and recursively executes $\textsc{Rec-Transpose}(A_1, B_1)$ and $\textsc{Rec-Transpose}(A_2, B_2)$.

# A matrix transposition cache-oblivious algorithm (3/3)

- Recall that the matrices are stored in row-major layout.
- Let $\alpha$ be a constant sufficiently small such that
  - two submatrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - even if each row starts at a different cache line.
- We distinguish three cases:
  - Case I: $\max\{m, n\} \leq \alpha L$.
  - Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.
  - Case III: $m, n > \alpha L$.

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

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

# Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The REC-TRANSPOSE algorithm divides the greater dimension $m$ by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole problem fits in cache. At this point:
  - the input array resides in contiguous locations, requiring at most $\Theta(1 + nm/L)$ cache misses
  - the output array consists of $nm$ elements in $n$ rows, where in the **worst case** every row starts at a different cache line, leading to at most $\Theta(n + nm/L)$ cache misses.
- Since $m \leq \alpha L$, the **total** cache complexity for this base case is $\Theta(1 + n)$, yielding the recurrence (where the resulting $Q(m, n)$ is a **worst case estimate**)

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

whose solution satisfies $Q(m, n) = \Theta(1 + mn/L)$.

# 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 $\Theta(m + n + mn/L)$ cache misses.
- The **worst case cache miss estimate** satisfies the recurrence

$$Q(m,n) = \left\{ \begin{array}{ll} \Theta(m + n + mn/L) & \text{if } m, n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2, n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m, n/2) + O(1) & \text{otherwise;} \end{array} \right.$$

  whose solution is $Q(m,n) = \Theta(1 + mn/L)$.
- **Therefore, the Rec-Transpose algorithm has optimal cache complexity.**
- Indeed, for an $m \times n$ matrix, the algorithm must write to $mn$ distinct elements, which occupy at least $\lceil mn/L \rceil$ cache lines.

# Plan

# A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze 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.**
- This straightforward divide-and-conquer algorithm contains **no voodoo parameters** (tuning parameters) and it uses cache optimally.
- Intuitively, this algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.
- These results require the tall-cache assumption for matrices stored in row-major layout format,
- This assumption can be relaxed for certain other layouts, see (Frigo et al. 1999).
- The case of Strassen's algorithm is also treated in (Frigo et al. 1999).

# A cache-oblivious matrix multiplication algorithm (2/3)

- To multiply an $m \times n$ matrix $A$ and an $n \times p$ matrix $B$, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

- In case (1), we have $m \geq \max\{n, p\}$. Matrix $A$ is split horizontally, and both halves are multiplied by matrix $B$.
- In case (2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied.
- In case (3), we have $p \geq \max\{m, n\}$. Matrix $B$ is split vertically, and each half is multiplied by $A$.
- The base case occurs when $m = n = p = 1$.

# A cache-oblivious matrix multiplication algorithm (3/3)

- 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: $m, n, p > \alpha\sqrt{Z}$.
  - 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 $m, n > \alpha\sqrt{Z})$.
  - 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 $p > \alpha\sqrt{Z})$.
  - Case IV: $m, n, p \leq \alpha\sqrt{Z}$.
- Similarly to matrix transposition, $Q(m, n, p)$ is a **worst case cache miss estimate**.

# Case I: $m, n, p > \alpha\sqrt{Z}$. (1/2)

$$Q(m,n,p) = \qquad (4)$$
$$\begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p)+O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p)+O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise} \ . \end{cases}$$

- The base case arises as soon as all three submatrices fit in cache:
  - The total number of cache 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.

# Case I: $m, n, p > \alpha\sqrt{Z}$. (2/2)

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

- In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, plus $O(1)$ cache misses for the overhead of manipulating submatrices.
- The solution to this recurrence is

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

- Indeed, for the base-case $m, m, p \in \Theta(\alpha\sqrt{Z})$.

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

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides $n$ or $p$ by 2 according to cases (2) and (3).
- At some point in the recursion, both $n$ and $p$ are small enough that the whole problem fits into cache.
- The number of cache misses can be described by the recurrence

$$Q(m, n, p) = \tag{5}$$
$$\begin{cases} \Theta(1 + n + m + np/L) & \text{if } n, p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m, n/2, p) + O(1) & \text{otherwise if } n \geq p \ , \\ 2Q(m, n, p/2) + O(1) & \text{otherwise ;} \end{cases}$$

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

- Indeed we have here: $mnp/(L\sqrt{Z}) \leq \alpha np/L$.
- The term $\Theta(1 + n + m)$ appears because of the row-major layout.

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

- 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 REC-MULT algorithm always divides $m$ by 2 according to case (1).
- 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

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

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

- Indeed we have here: $mnp/(L\sqrt{Z}) \leq \alpha\sqrt{Z}m/L$; moreover $Z \in \Omega(L^2)$ (tall cache assumption).

# Case IV: $m, n, p \leq \alpha\sqrt{Z}$.

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

# Typical memory layouts for matrices

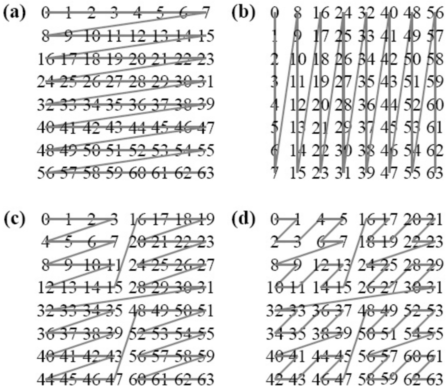
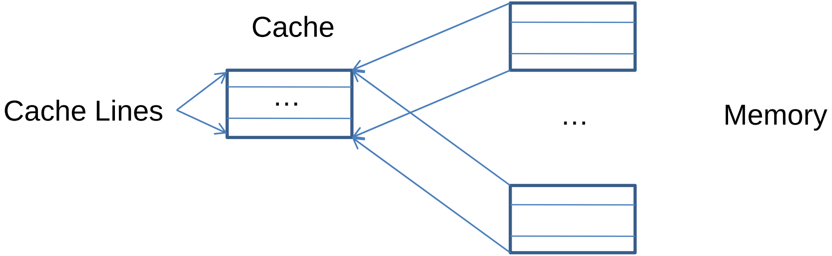



**Figure 2:** Layout of a $16 \times 16$ matrix in **(a)** row major, **(b)** column major, **(c)** $4 \times 4$-blocked, and **(d)** bit-interleaved layouts.

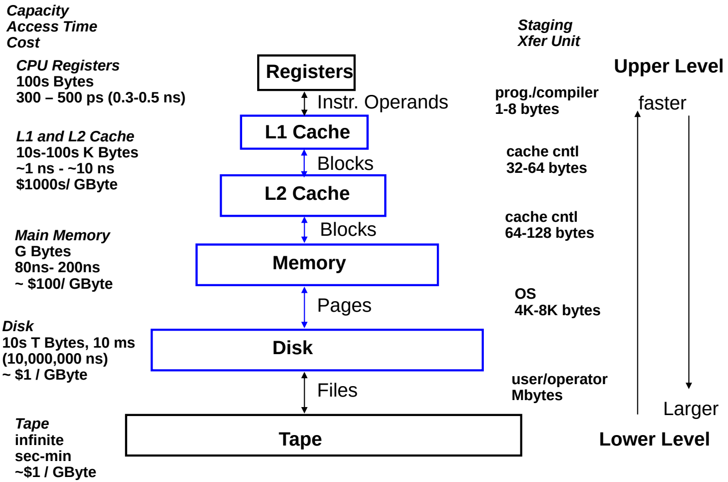
# Plan

# Basic idea of a cache memory




- A cache is a smaller memory, faster to access
- Using smaller memory to cache contents of larger memory provides the illusion of fast larger memory
- Key reason why this works: **temporal locality** and **spatial locality**.

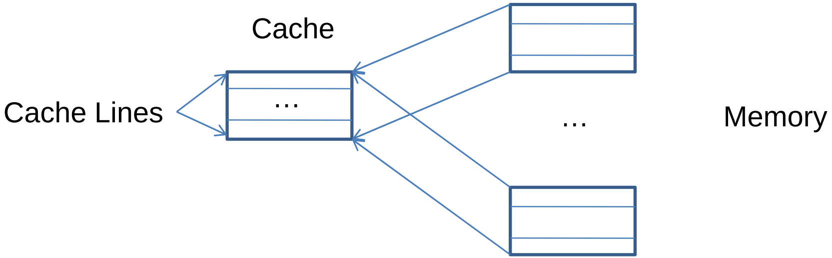
# Levels of the Memory Hierarchy



# Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.
- **False sharing miss:** Occurs when another processor uses different data in the same cache line. Cure: Pad data.

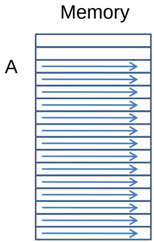
# A simple cache example




- Byte addressable memory
- Size of 32Kbyte with direct mapping and 64 byte lines (512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- "Therefore" successive 32Kbyte memory blocks line up in cache
- A cache access costs 1 cycle while a memory access costs 100 cycles.
- How addresses map into cache
  - Bottom 6 bits are used as offset in a cache line,
  - Next 9 bits determine the cache line

# Exercise 1 (1/2)

```
// sizeof(int) = 4 and Array laid out sequentially in memory
#define S ((1<<20)*sizeof(int))
int A[S];
// Thus size of A is 2^(20) x 16 bytes
for (i = 0; i < S; i++) {
        read A[i];
}
```




Total access time? What kind of locality? What kind of misses?

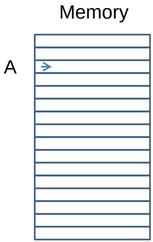
# Exercise 1 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i];
}
```

- `S` reads to `A`.
- 16 elements of `A` per cache line
- 15 of every 16 hit in cache.
- Total access time: $15(S/16) + 100(S/16)$.
- spatial locality, cold misses.

# Exercise 2 (1/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[0];
}
```




Total access time? What kind of locality? What kind of misses?

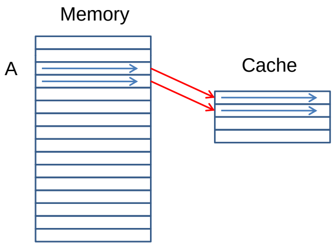
# Exercise 2 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[0];
}
```

- `S` reads to `A`
- All except the first one hit in cache.
- Total access time: $100 + (S - 1)$.
- Temporal locality
- Cold misses.

# Exercise 3 (1/2)

```
// Assume 4 <= N <= 13
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i % (1<<N)];
}
```




Total access time? What kind of locality? What kind of misses?

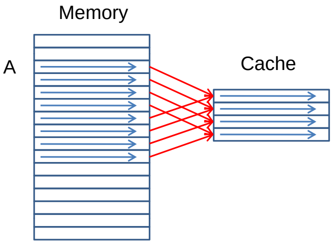
# Exercise 3 (2/2)

```
// Assume 4 <= N <= 13
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i % (1<<N)];
}
```

- S reads to A
- One miss for each accessed line, rest hit in cache.
- Number of accessed lines: $2^{N-4}$.
- Total access time: $2^{N-4}100 + (S - 2^{N-4})$.
- Temporal and spatial locality
- Cold misses.

# Exercise 4 (1/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
 read A[i % (1<<N)];
}
```




Total access time? What kind of locality? What kind of misses?

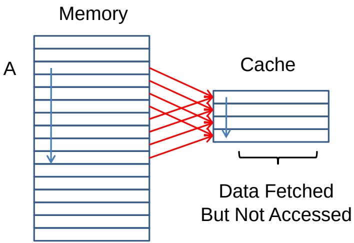
# Exercise 4 (2/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
 read A[i % (1<<N)];
}
```

- `S` reads to `A`.
- First access to each line misses
- Rest accesses to that line hit.
- Total access time: $15(S/16) + 100(S/16)$.
- Spatial locality
- Cold and capacity misses.

# Exercise 5 (1/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
read A[(i*16) % (1<<N)];
}
```




Total access time? What kind of locality? What kind of misses?

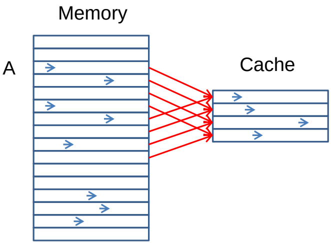
# Exercise 5 (2/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
read A[(i*16) % (1<<N)];
}
```

- `S` reads to `A`.
- First access to each line misses
- One access per line.
- Total access time: $100S$.
- No locality!
- Cold and conflict misses.

# Exercise 6 (1/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[random()%S];
}
```




Total access time? What kind of locality? What kind of misses?

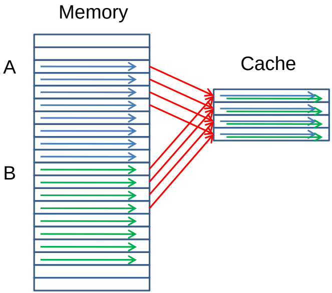
# Exercise 6 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[random()%S];
}
```

- `S` reads to `A`.
- After $N$ iterations, for some $N$, the cache is full.
- Them the hance of hitting in cache is $32Kb/16Mb = 1/512$
- Estimated total access time: $S(511/512)100 + S(1/512)$.
- Almost no locality!
- Cold, capacity conflict misses.

# Exercise 7 (1/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```




Total access time? What kind of locality? What kind of misses?

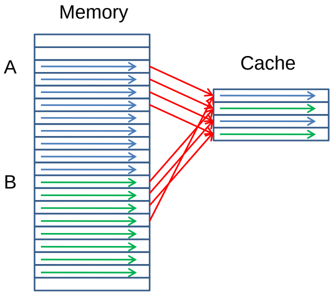
# Exercise 7 (2/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B interfere in cache: indeed two cache lines whose addresses differ by a multiple of $2^9$ have the *same way to cache*.
- Total access time: $200S$.
- Spatial locality but the cache cannot exploit it.
- Cold and conflict misses.

# Exercise 8 (1/2)

```
#define S ((1<<19+16)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



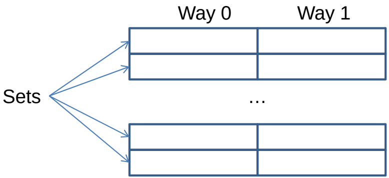

Total access time? What kind of locality? What kind of misses?

# Exercise 8 (2/2)

```
#define S ((1<<19+16)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B almost do not interfere in cache.
- Total access time: $2(15S/16 + 100S/16)$.
- Spatial locality.
- Cold misses.

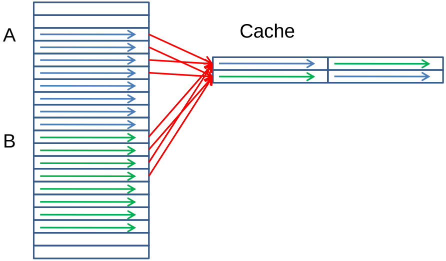
# Set Associative Caches




- **Set associative caches** have sets with multiple lines per set.
- Each line in a set is called a way
- Each memory line maps to a specific set and can be put into any cache line in its set
- In our example, we assume a 32 Kbyte cache, with 64 byte lines, 2-way associative. Hence we have:
  - 256 sets
  - Bottom six bits determine offset in cache line
  - Next 8 bits determine the set.

# Exercise 9 (1/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```




Total access time? What kind of locality? What kind of misses?

# Exercise 9 (2/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- `S` reads to `A` and `B`.
- `A` and `B` lines hit same set, but enough lines in a set.
- Total access time: $2(15S/16 + 100S/16)$.
- Spatial locality.
- Cold misses.

# Acknowledgements and references

**Acknowledgements.**

- Charles E. Leiserson (MIT) and Matteo Frigo (Intel) for providing me with the sources of their article *Cache-Oblivious Algorithms*.
- Charles E. Leiserson (MIT) and Saman P. Amarasinghe (MIT) for sharing with me the sources of their course notes and other documents.

**References.**

- *Cache-Oblivious Algorithms* by Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran.
- *Cache-Oblivious Algorithms and Data Structures* by Erik D. Demaine.