Cache-Oblivious Algorithms

Introduction

1. Fast memory (cache) is handled by hardware and allows the programmer to work in an oblivious way
    - However, this negatively impacts performance
    - Naive binary search achieves worse performance than B-tree
        - B-tree requires tuning B to the specific hardware which affects portability
2. Is there a different algorithm or data structure that would be efficient no matter what the memory hierarchy looks like or how it's managed
    - Matteo Frigo, Charles Leiserson, Sridhar Ramachandran, Harald Prokop discovered a way to be cache oblivious for common algorithms like matrix multiplying and sorting

An Introductory Example

1. Consider the I/O optimal algorithm for reduction
    - Achieves O(n/L) transfers, but is based on L, the number of words the hardware can transfer to fast memory
    - If fast memory is automatically managed (cache, TLB/disk) we can write the naive summing algorithm which is much simpler
        - This algorithm doesn't reference Z or L, so we say it's cache oblivious
    - Can show that the conventional way of doing a reduction also incurs O(n/L) transfers
2. Oblivious (to fast memory): Algorithm makes no reference to fast memory or its parameters (Z, L)
3. Two concerns:
    - How to model automatic fast memory?
    - Can an oblivious algorithm match I/O performance of an "aware" algorithm?
4. Terminology: "Cache" = any automatic fast memory

The Ideal-Cache Model

1. Assume slow and fast memory are divided into blocks of size L words
    - Cache line or "block"
2. When a program issues a load or store, we go to the cache
    - If the data is present in the cache, we refer to it as a "cache hit"
    - Otherwise, we refer to it as a "cache miss"
    - Both load and store misses result in moving data from slow to fast memory
3. When bringing data into the cache, we must bring in an entire block of L words
    - What is brought in with the data depends on how the data is aligned
4. Assume the cache is fully associative
    - Real caches are typically not fully associative
5. If the cache is full, we must choose a line to evict
    - Store-eviction: "Dirty" line must be written to slow memory -> 1 transfer
6. Assume optimal replacement: Evict line or block accessed farthest in the future
7. Summary:
    - Program issues load and store operations
    - Hardware manages Z/L cache lines
    - Slow memory -> cache transfers happen in lines (or "blocks") of size L
    - Accessing a value in cache = cache hit; otherwise, cache miss
    - Cache is fully associative
    - Assume optimal replacement policy
8. Q(n;Z,L) = Number of misses + number of store evictions

What Would an Ideal Cache Do?

1. Consider an ideal cache of size Z = 4 words, L = 1 word

2. Consider the following sequence of accesses:
    - LD [0xBEEF]
    - LD [0xF00D]
    - LD [0xC0DE]
    - LD [0xD00D]
    - LD [0xF00D]
    - LD [0xAB8D]
    - LD [0xBEEF]
    - LD [0xF00D]
    - LD [0xC0DE]
    - LD [0xD00D]
3. What is the state of the cache after the first 9 accesses?
    - [0xBEEF]
    - [0xF00D]
    - [0xC0DE]
    - [0xAB8D]
4. How many evictions occurred?
    - 0xD00D is the only eviction

LRU Replacement

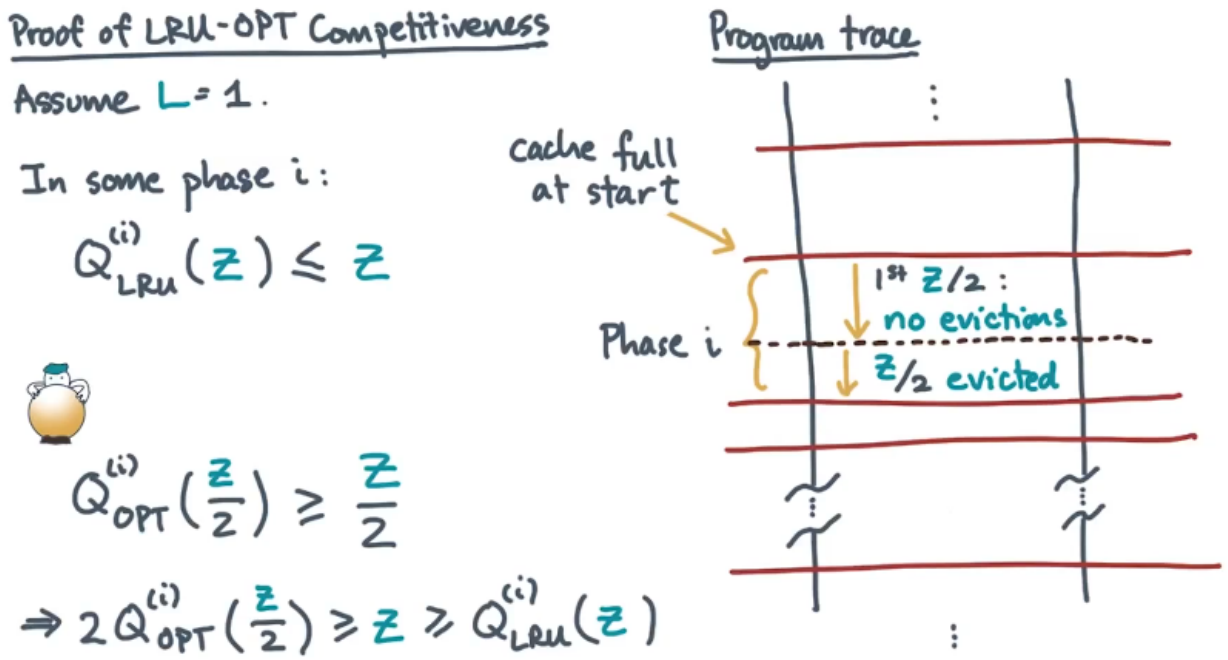
1. Consider an ideal cache of size Z = 4 words, L = 1 word
2. Consider the following sequence of accesses:
    - LD [0xBEEF]
    - LD [0xF00D]
    - LD [0xC0DE]
    - LD [0xD00D]
    - LD [0xF00D]
    - LD [0xAB8D]
    - LD [0xBEEF]
    - LD [0xF00D]
    - LD [0xC0DE]
    - LD [0xD00D]
3. What is the state of the cache after the first 9 accesses? Assume an LRU replacement algorithm?
    - [0xBEEF]
    - [0xF00D]
    - [0xC0DE]
    - [0xAB8D]
4. How many evictions occur if we assume an LRU replacement algorithm?
    - 3; 0xAB8D replaces 0xBEEF, 0xBEEF replaces 0xC0DE, 0xC0DE replaces 0x0xD00D

How Ideal is an Ideal Cache?

1. Key assumptions of an ideal cache:
    - Optimal replacement
    - Two-levels of memory
    - Automatic replacement
    - Full associativity
2. Lemma for LRU vs Optimal cache replacement
    - Lemma: Qlru(n;Z,L) <= 2 * Qopt(n;Z/2,L)
    - Interpretation: Qlru and Qopt can be asymptotically close
3. Corollary ("Regularity condition"):
    - Qopt(n;Z,L) = O(Qopt(n;2 * Z,L)
    - => Qlru(n;Z,L) = O(Qopt(n;Z,L))

Proof of the LRU-OPT Competitiveness Lemma

1. Consider a non-Strassen matrix multiply algorithm
   - Suppose we show Qopt(n;Z,L) = O(n^3/L/sqrt(Z))
     - Assume L=1 -> Qopt = O(n^3/sqrt(Z))
   - Qopt(n;2Z) = O(n^3/sqrt(2Z)) = O(n^3/sqrt(Z))



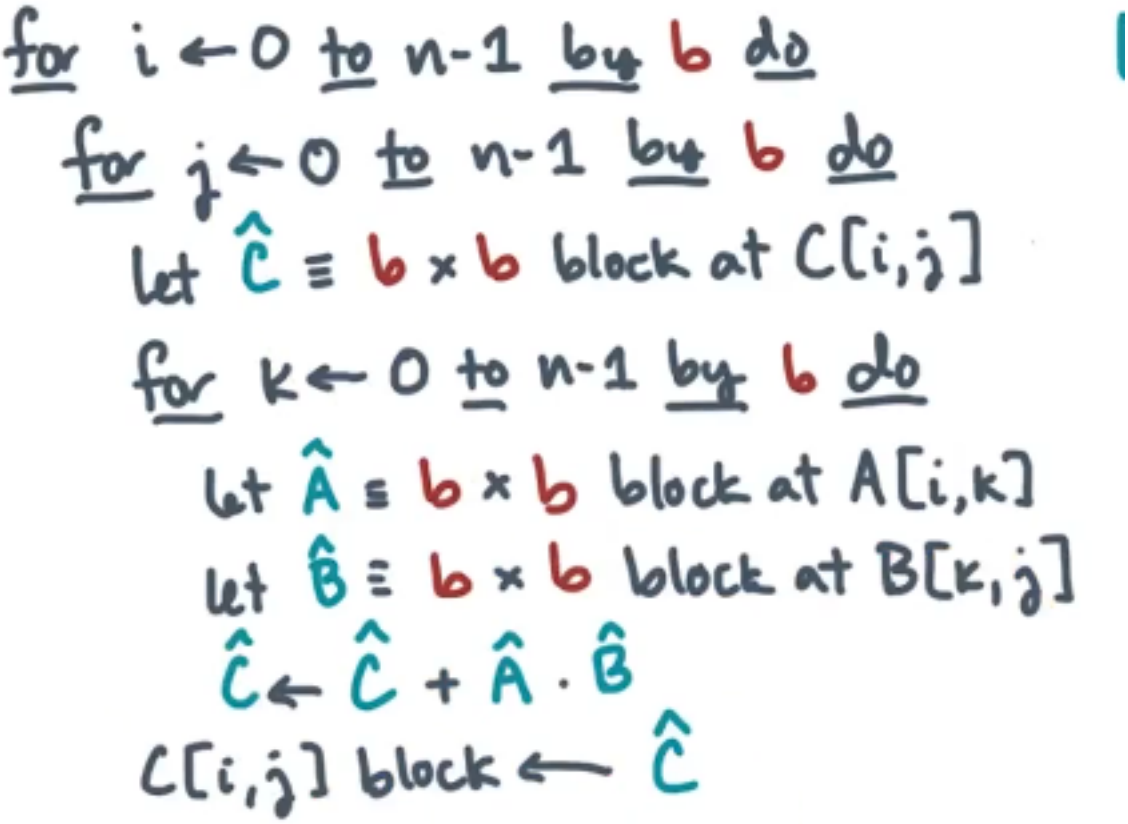
Proof of LRU-OPT Competitiveness

LRU Sorting

1. Suppose a sorting algorithm has an ideal (Z,L) cache
   - Qopt(n;Z,L) = O(n/L * log(n/L) / log(Z/L)) caches misses
2. How many cache misses will there be on a (Z,L) cache with LRU replacement?
   - Qopt(n;2Z,L) = O(n/L * log(n/L) / log(2Z/L)) caches misses
   - log(2Z/L) = log(Z/L) + 1
   - Because this is a constant factor, Qlru = Qopt

The Tall-Cache Assumption

1. Tall cache assumption: Cache should be taller in terms of number of lines than it is wide in terms of number of words per line
   - Z >= L^2
   - This means than even if a cache is big enough to hold a block (i.e., in a matrix multiply algorithm) it still might not fit in cache
     - If a column doesn't fill a full line, it can mean that there won't be enough lines to hold all of the columns needed
2. Moral: Data layout matters!



Aware Matrix Multiply Psuedocode

Which "Cache" is Tall?

1. Which caches are tall? Assume word size = 8 bytes
   - 16 registers, 16 bytes each
     - 16 lines, 2 words per line -> True
   - 32 KiB L1 cache with 64 byte lines
     - 512 lines, 8 words per line -> True
   - 8 MiB L2 cache with 128 byte lines
     - 2^16 lines, 16 words per line -> True
   - 256-entry TLB with 8 KiB pages (slow memory = RAM)
     - 256 lines, 1024 words per line -> False
   - 32 GiB RAM with 32 KiB pages (slow memory = disk)
     - 2^20 lines, 2^12 words per line -> True
2. Need to check if Z/L is greater than L

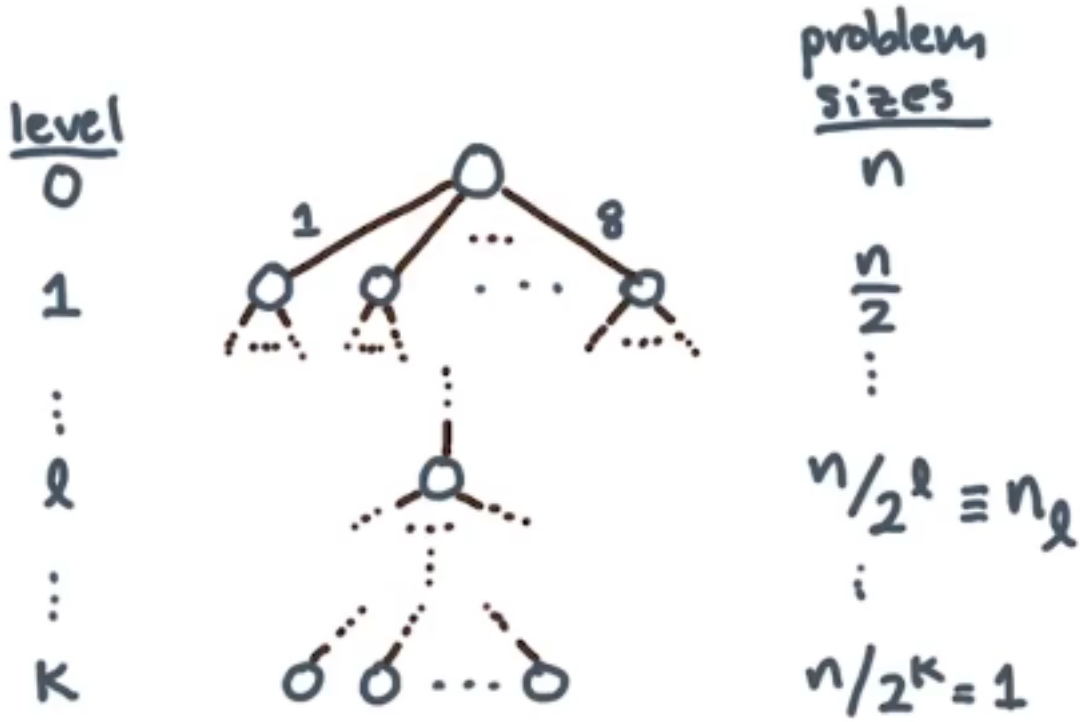
Cache-Oblivious Matrix Multiply

1. Cache-aware matrix multiply has Q(n;Z,L) = O(n^3/L/sqrt(Z)) transfers
   - b should be chosen to be sqrt(Z) to use the cache optimally
   - I/O optimal
2. Is there a cache-oblivious and optimal algorithm?
   - Yes, use divide and conquer
3. Algorithmic complexity

- F(n) = 8 * F(n/2) if n > 1
- 2 if n == 1
- Solve using Master Theorem -> 2n^3

4. Transfer complexity
   - Operands fit in cache at level l:
     - 3nl^2 <= Z -> nl <= f * sqrt(Z)
   - Q(n;Z,L) = O(n^2/L) if n <= f * sqrt(Z)
   - 8 * Q(n/2;Z,L) + O(1) otherwise
   - Solve using Master Theorem -> O(n^3/L/sqrt(Z))
     - Matches the cache-aware algorithm without referring to Z or L

```
mm(n; A, B, C)
if n == 1 then {
    C <- C + A * B;
}
else {
    for i <- 1 to 2 do
        for j <- 1 to 2 do
            for k <- 1 to 2 do
                mm(n/2; Aik, Bkj, Cij);
}
```



Oblivious Matrix Multiply Breakdown

Cache-Oblivious Binary Search

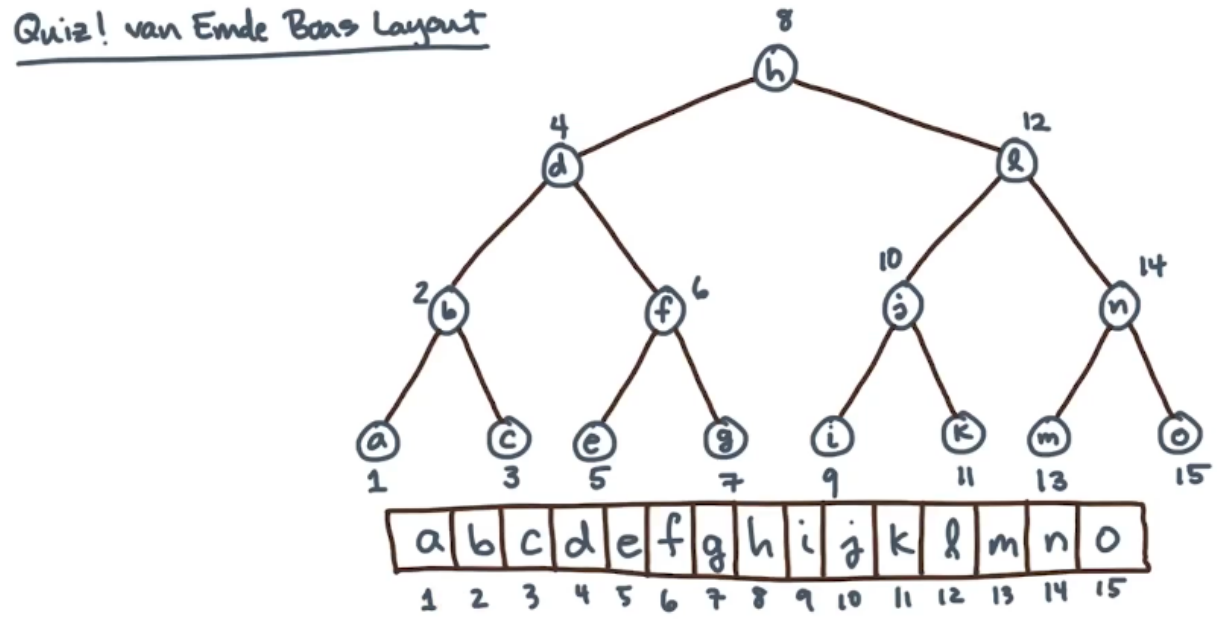
1. Naive binary search algorithm is Q(n;Z,L) = O(log(n/L)) transfers
   - Q(n;Z,L) = 1 + Q(n/2;Z,L) if n > L
   - 1 if n <= L

- Optimal algorithm is Q(n;Z,L) = O(log(n)/log(L)) transfers

2. We can achieve the optimal complexity by leaving the algorithm in tact but changing the layout
3. The Van Emde Boas layout arranges nodes in the binary tree that are close to each other in memory
   - Put blocks of nodes of size L near each other
   - This means that we will only incur cache misses when we bring in a new block of L nodes
   - This provides the optimal O(log(n)/log(L)) number of cache misses because the maximum path length is O(log(n)) and the maximum number of misses is O(log(L))

Van Emde Boas Layout

1. Give the linearized order of nodes in a Van Emde Boas layout
   - h, d, l, b, a, c, f, e, g, j, i, k, n, m, o



Van Emde Boas Layout

Conclusion

1. Cache obliviousness has a lot of solid theory but lacks practical implementations or evaluations of these algorithms
2. As environments move to be more virtualized, programs have to compete for resources (cache, network bandwidth)
   - In such environments, aiming to be as oblivious as possible could lead to the best performance