

# Designing for Memory Hierarchy

---

## Adding Caches to the Performance Model

William Gropp



- Lets start with a simple 3-level memory model:
  - ◆ Registers
  - ◆ Cache
  - ◆ Main Memory
- The cache is composed of lines of L words; we normally assume that there are at least L lines (often many more)
- If a load or store from the CPU cannot be satisfied from the cache, a *cache miss* occurs.
- When a miss occurs, one line is loaded from memory into the cache (this line contains (a copy of) the referenced memory location)
- If the cache was already full, one line is evicted (written back out to memory if necessary (e.g., a write occurred to some data in the line))



2

## Some Further Assumptions

---

- The cache is perfect:
  - ◆ Each line of memory can be placed anywhere in the cache. This is called a *fully associative* cache for reasons we will discuss later.
  - ◆ When a line is evicted, the cache makes the perfect choice, choosing the line that will not be needed for the longest time *in the future*.
    - In practice, the policy used is called *Least Recently Used (LRU)*: the cache line that has not been used (accessed) for the longest period of time is selected for eviction.

3



## A New Complexity Model

---

- In addition to counting computational operations, count cache misses.
  - ◆ This is one step better than counting loads and stores, as loads and stores assumes a 2-level memory hierarchy – registers and main memory.
  - ◆ There are many other parameters that can be important, but following our strategy, we can use this model as a kind of bound on performance.
  - ◆ Specifically, we do *not* weight a cache miss – we will simply count them. This means that this model cannot be used to compare algorithms that trade work for cache misses.
  - ◆ We also assume that cache misses are not overlapped with other operations (we'll see later that this can be a significant limitation)
- However, with all of these restrictions, this is still a better model than just counting operations



4

# Cache Oblivious Algorithms

- Our new complexity model is  $W(n) + Q(n,L,Z)$  where  $W(n)$  is the work,  $Q(n,L,Z)$  are the number of cache misses,  $n$  is the "problem size",  $L$  is the line size, and  $Z$  is the cache size.
- The goal is to minimize (or at least reduce) this cost relative to the simple algorithms that only consider  $W(n)$ .
- Lets simplify further: Are there algorithms that are independent of  $L$  and  $Z$  that are good approximations to optimal solutions according to this complexity model?
  - ♦ Call these *cache parameter oblivious algorithms*, or *cache oblivious algorithms* for short
  - ♦ The obviously aren't cache oblivious, but they are portable to any system for which this three-level memory model is a good approximation.



5

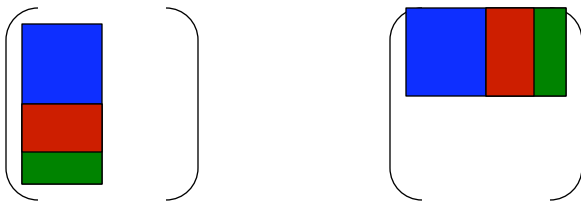
# Divide and Conquer

- Consider the matrix examples: matrix multiply and transpose. We know that one approach that improves cache utilization is to subdivide each loop into blocks. These blocks must be chosen to fit in the relevant level of cache (in a model that is optimizing for a particular machine architecture)
- What if instead we applied *divide and conquer* and stopped at some "good" size?



6

## Divide and Conquer: Transpose

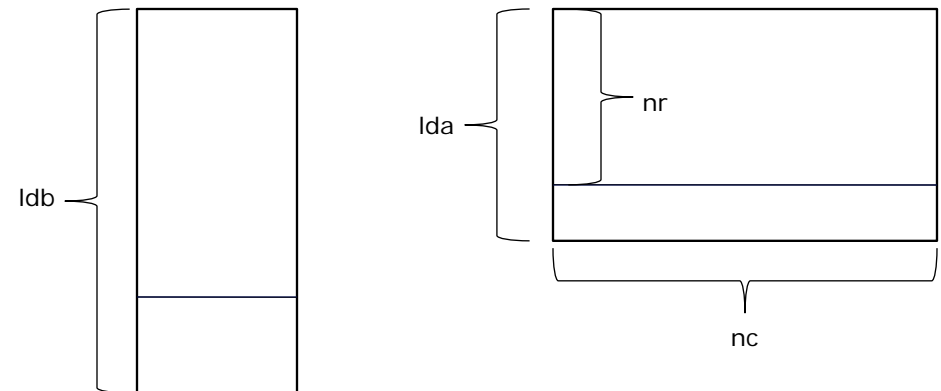


- By recursively subdividing (and keeping the submatrices reasonably square by always dividing the long dimension), we get to a block that will fit in cache, eliminating most (non-compulsory) cache misses.
- By picking a reasonably small size (say, 8k words) for the smallest block (where recursion is terminated), we should reduce the number of cache misses *without an explicit knowledge of the cache size* (other than "bigger than 16k")



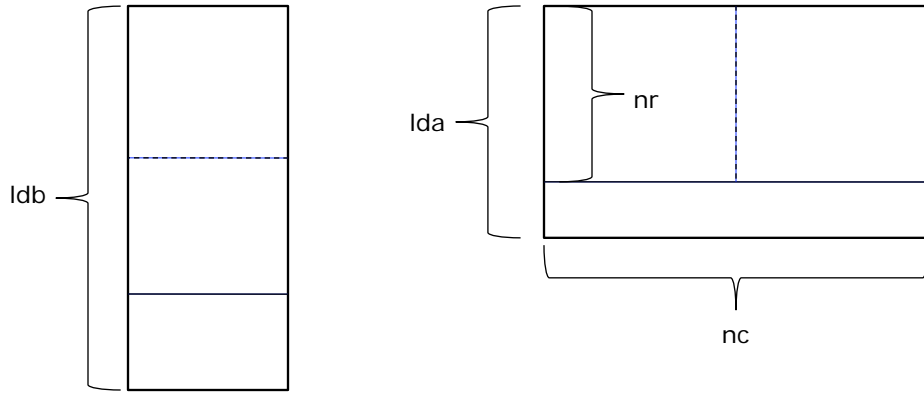
7

## Details of CO Transpose



8

# Details of CO Transpose



9

# The CO Transpose Code

```
int transpose( double *a, int ndra, int nr, int nc, double *b, int ndr )
{
  if (nr < 32) {
    transposeBase( a, ndra, nr, nc, b, ndr );
  }
  else {
    /* subdivide the long side */
    if (nr > nc) {
      transpose( a, ndra, nr/2, nc, b, ndr );
      transpose( a + nr/2, ndra, nr-nr/2, nc, b+(nr/2)*ndr, ndr );
    }
    else {
      transpose( a, ndra, nr, nc/2, b, ndr );
      transpose( a+ndra*(nc/2), ndra, nr, nc-nc/2, b+nc/2, ndr );
    }
  }
}
```



10

## Optimality

- The “cache oblivious” algorithm is asymptotically optimal (under the performance measurement of this model) within a factor of two
  - No data movement (involving the matrices) occurs until the minimum block size is reached. At this point, the “naïve” transpose algorithm may be used, because both input and output submatrices fit in cache. The number of misses is the number of compulsory misses, plus a small number if the blocks are not exact multiples of the line size
    - And they probably aren't exact multiples – that's part of the point of being cache oblivious



11

## Some Results

Size	Naïve	Cache Oblivious
64	3000	2800
128	1900	3400
256	1900	4300
512	1200	1800
1024	320	1100
2048	180	1100
2049	780	1100
4096	150	980
4097	560	1100



12

## Observations

- Cache Oblivious approach provides more consistent performance with a relatively simple implementation
  - ◆ Less sensitive to layout than the naïve implementation
- However, does not preserve full L1-resident performance



13

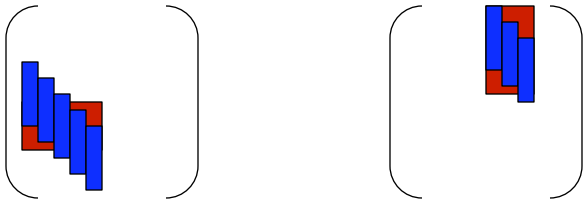
## Refinements and Issues

- Recursion, implemented as function calls, can be relatively expensive (a few 100's of instructions)
  - ◆ Manual implementation with a stack is relatively easy but tedious
  - ◆ Question: Should there be special support for "self recursive" functions (much like tail-recursion optimization, used when the last action in a recursive function is to call itself)?
- Cache utilization is approximate
- Performance model assumes that the cache is *fully associative* – any data item may be placed in any location. More on this later.



14

## Inefficient Use of Cache Line



- Note all elements of each cacheline may be used in the small blocks
- If the number of cachelines in the cache are large enough, if the recursive subdivision stops with a large enough submatrix, the inefficiency is at most a factor of 2 (a cache line is read or written twice, for the blocks either above or below in the same column)
- We assume column-major (Fortran) storage order; for row-major, the cache lines are aligned by rows.



15

## Some Comparisons

- From Cache-Oblivious Algorithms EXTENDED ABSTRACT Matteo Frigo Charles E. Leiserson Harald Prokop Sridhar Ramachandran (1999)
- Note cost: 30ns/2Flops, or 67 Mflops
  - ◆ Depending on platform, this is either good or bad, relative to the best available code
- This result shows that cache oblivious programs can be faster than the simple, naïve code, with only a small increase in complexity
- They do not compare with the best available implementations

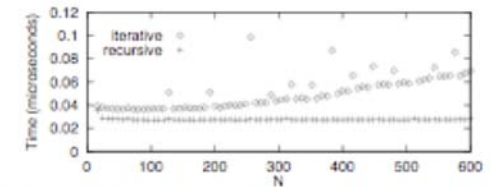


Figure 5: Average time taken to multiply two  $N \times N$  matrices, divided by  $N^3$ .



16

# Cache Oblivious Searching

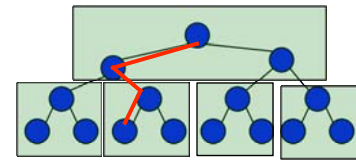
- Consider a static set of keys that needs to be searched repeatedly, and assume that a binary tree is the preferred approach
- If the tree is built using memory allocated for each entry, linked with pointers, then each link will often cause a cache miss.
  - ◆ For suitably large set of keys, a cacheline is likely to be ejected before another key on that cacheline is accessed
  - ◆ Thus,  $O(\log n)$  cache misses



17

# A CO Data Structure

- A binary tree of height  $\log_2 n$  fits in  $n-1$  words.
- Organize the tree as a collection of subtrees, each of which is stored in consecutive locations
- $\log_2 L$  accesses are in a cache line of  $L$  keys, so the total number of cache misses is roughly  $\log_2 n / \log_2 L$ , where  $n$  is the number of keys.



18

# More on Caches

- We've assumed that when a cache line is loaded, it can go anywhere in the cache (the full cache is available for any line loaded from memory)
- This is rarely true
- As for the TLB example, computing the location of an address in cache must be very fast, thus it is impractical to search through a large cache to find where a memory line was placed.
- Typically, a cache is divided into *sets*; an address is mapped to a set, and the choice of which set is made based on the cache replacement policy.
- For example, a 32KB cache might have 64 sets each with 4 entries (each entry is a cache line 128 Bytes).

Rest of address	Set #	Location in cache line
19-51 bits	6 bits	7 bits

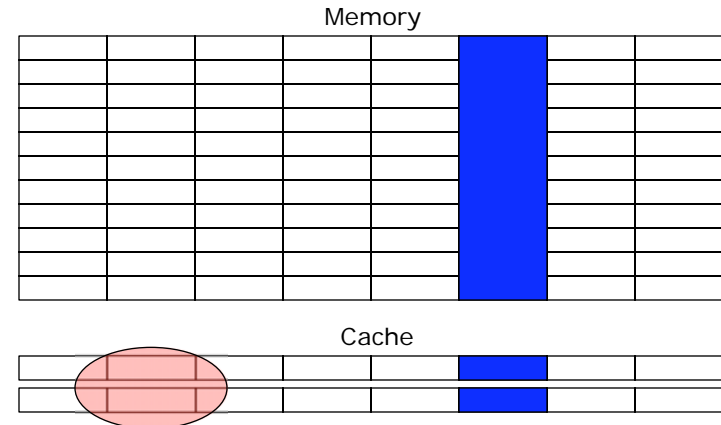
19-51 bits

6 bits

7 bits

19

# Example of Cache Sets



All memory items in a column get mapped to the set – the cache appears to have size two in this case!



20

## Why Do We Care?

Size	Naïve	Cache Oblivious
2048	180	1100
2049	780	1100
4096	150	980
4097	560	1100

- Note the performance variation for the simple algorithm when the matrix size changes by a one row/column
- Successive elements from the same row are being mapped to the same set, reducing the effective cache size to the number of entries in the set



21

## Reducing the Impact of Cache Sets

- Powers of two are bad for strides through memory – likely to map to the same location in the available sets
  - ♦ Solution: *Pad* data structures to change the access stride
  - ♦ Cost: Some extra memory
- Modern systems may have sets of size 8 or more
  - ♦ Be careful about the number of different variables
    - Large arrays may be allocated on power-of-two boundaries (simplifying memory allocation) but at the risk of mapping to the same location in the available sets (as if the collection of variables was really a single variable with strides that are a power of two).



22

## When Copying is Not Bad

- When the successive accesses to data will be more efficient (or predictable) if the data is in a special form
- For example, rearranging the data to avoid mapping to the same cache set.
- Dense matrix-matrix multiply codes often do this (copy to a temporary, block-oriented structure) to improve performance for subsequent accesses (note there are  $n^2$  data items but  $n^3$  operations – each data item is used  $n$  times, so rearranging for efficient access is worthwhile in this case)



23