**CSE341T/CSE549T** **August 25, 2014**

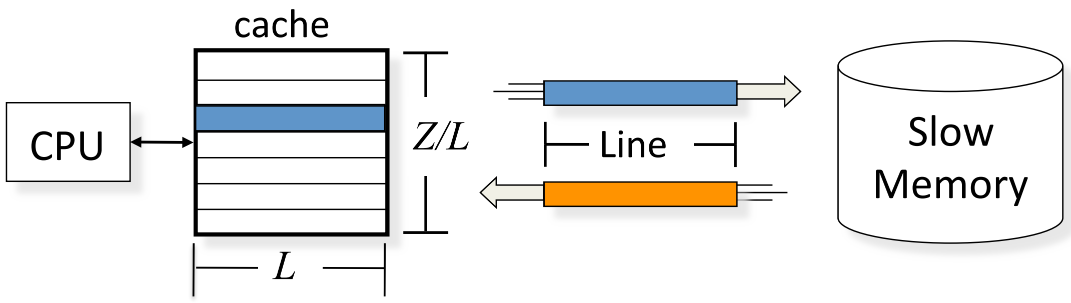
# Cache-Aware Analysis of Algorithms

So far, we have been assuming a *Random Access Model* for memory — accessing any memory location takes constant time. However, modern machines have complex memory hierarchies consisting of many levels of caches and it takes less time to access data that is already in a processor's cache than to access data that is not. In fact, since the speed of processors is growing faster than the memory access latency, it can often be more important to minimize the number of cache misses in your computation than it is minimize the number of computations that you perform on your CPU.

Today, we will start thinking about how we go about analyzing the cache-complexity of algorithms and how we can design algorithms that have a small cache complexity. We will focus on cache complexity of sequential algorithms for now — there is, however, extensive work on how to analyze the cache complexity of parallel algorithms as well. We will see some of this later in the class.

## 1 Ideal Cache Model

In order to analyze the cache-complexity of any algorithm, we need a model. We will use the ideal cache model. In this model, we consider a 2-level memory hierarchy consisting of a cache (fast memory) of size $Z$ and a slow memory of unbounded size. The cache is divided into cache-lines of size $L$ each, for a total of $Z/L$ lines in the cache. The data is moved between the cache and the slow memory at the granularity of the line.



When the CPU accesses some memory location, if the line containing that memory location is already in cache, then we call it a ***cache-hit*** and the cache-cost of accessing that memory location is 0. If that line is not already in cache, then it is a ***cache miss*** — this line is then transfered from the slow memory to cache for the cache-cost of 1. In this process, some other line may be kicked out of cache. Which line gets kicked out depends on the ***cache-replacement policy*** of the cache. Our goal is to design algorithms that have the smallest number of cache misses; that is, we want to access data that is already in cache as much as possible. We will denote the cache complexity by $Q(n; Z, L)$.

**Ideal Cache Assumptions:**

**Tall Cache Assumption:** We will assume that $Z \geq \Omega(L^2)$ — the height of the cache is larger than its width. We don't always need this assumption, but it makes the design of some algorithms easier. It holds for most modern caches.

**Fully Associative Cache:** When a cache miss occurs and we must bring cache line into cache, the cache must decide which particular line in the cache it must be mapped to. The ideal cache model assumes that any cache line can be mapped to any location in the cache. This is generally not true for most real caches — most caches are 4, 8 or 16-way associative. However, one can generally map the algorithms designed for fully associative caches to algorithms for these caches by clever design of data layout.

**Optimal Cache Replacement Policy:** When a cache miss occurs and a new cache line is brought into cache, some previous cache line must be removed from cache. The ideal cache model assumes that the replacement policy is optimal. In reality, the optimal cache policy can not be implemented since it requires that we know the future. However, the optimal policy can be approximated with real policies with only constant factor loss in efficiency.

## 2 Warm Up: Cache-Complexity of Scanning an Array

We start by analyzing a very simple algorithm:

```
SCANARRAY($A$)
1  for $i \leftarrow 1$ to $n$
2      do $A[i] \leftarrow A[i] + 1$
```

Here we are simply accessing an array, one element at a time. If the array is stored contiguously in memory, how many cache misses do we incur? We will always assume that the cache starts out empty. $Q(n; Z, L) = O(n/L + 1)$.

## 3 Cache-Complexity of Matrix Multiplication

Now lets look at a slightly more complicated algorithm. We have seen matrix multiplication in class. Lets first look at the one with triply nested for-loops.

```
MatrixMultiply$(A, B, C, n)$
1  for $i \leftarrow 1$ to $n$
2      do for $j \leftarrow 1$ to $n$
3             do $C[i,j] \leftarrow 0$
4                for $k \leftarrow 1$ to $n$
5                    do $C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]$
```

We first have to decide how these matrices are stored in memory — for now we assume that they are stored in row-major order. To compute the cell $C[i, j]$ we need the $i$th row of $A$ and the $j$th column of $B$. If they are not already in cache, getting the row of $A$ causes $n/L$ cache misses and getting the column of $B$ requires $n$ cache misses. There are three cases:

**Case 1:** $3 < L < n$ — Since we have fewer than $n$ lines in cache, by the time we need the next element of $B$ from a line, the line will have been evicted from cache. Therefore, we will incur $n$ cache misses (due to $B$) for computing every element of $C$, leading to a total of $\Theta(n^3)$ cache misses. (We need the cache to have at least 3 lines since we should be able to read one line each from $A$, $B$ and $C$. )

**Case 2:** $nL + n < Z < n^2$ — We can keep the an entire row of $A$ in cache and we can also keep $n$ columns of $B$ in cache — so we can compute $L$ elements of $C$ with just $n$ cache misses. Therefore, the total number of cache misses is $O(n^3/L)$.

**Case 3:** $3n^2 < Z$ — We can do the entire computation in cache, so we pay only for the initial read of the matrices which costs $\Theta(n + n^2/L)$.

Therefore, we need a really large cache to get the cost below $O(n^3)$. Note that $O(n^3)$ is the worst cache complexity we can get, since the total work of the algorithm is $O(n^3)$; in fact, we may as well not have a cache.

The fact that $B$ is stored in a row-major layout is a big problem. How about if we store $B$ in a column major layout and keep $A$ in row-major layout? In this case, computing a particular cell $C[i, j]$ causes only $\Theta(1 + n/L)$ cache misses if nothing is in cache. Lets look at the three cases again:

**Case 1:** $3 < L < n^2$ — Each element of $C[i, j]$ now only needs $n/L + 1$ cache misses, since we only need to read one row of $A$ and one column of $B$. We can reuse the row of $A$ when computing $C[i, j + 1]$, but we have to read the next column of $B$. Therefore, each cell costs $n/L + 1$ cache misses. Since there are a total of $n^2$ cells, the total cost is $\Theta(n^2 + n^3/L)$.

**Case 3:** $3n^2 < Z$ — We can do the entire computation in cache, so we pay only for the initial read of the matrices which costs $\Theta(n + n^2/L)$.

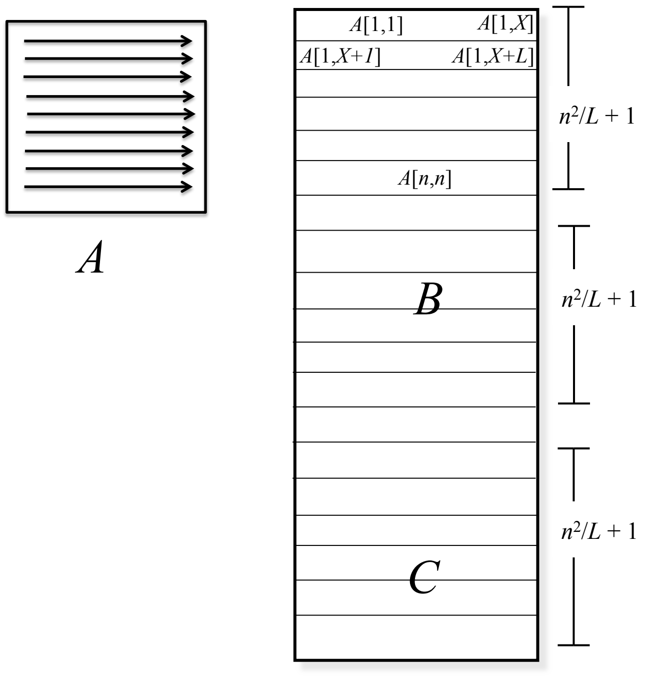


**Figure 1**: Row Major Layout for Matrices

Therefore, we did a little bit better. Lets us think about what we need to do in order to get fewer cache misses:

**Spatial Locality:** Once we read a line of cache, we want to use all of the data in that line. Getting a line to read just one word is wasteful. When $B$ was in row major order, we were using spatial locality in $A$, but not in $B$, since we were bringing a line into cache, reading a word and then discarding it again. However, when we switched to column major order in $B$, we start making good use of spatial locality for both $A$ and $B$, since we read a line in, and then use the entire line.

**Temporal Locality:** Once we read a line into cache, we want to read the data from that line over and over again. In both these algorithms, when the cache is small, we are never able to use temporal locality. When cache is bigger than $n$, we manage to use temporal locality for $A$, but not for $B$.

So now, lets think about how we can make better use of temporal locality. First, is there any possibility of using temporal locality? If we look at the scan algorithm, we can not possibly use temporal locality, since each element is read only once. In matrix multiplication, however, we access every word $n$ times. So there is a large scope for using temporal locality. Here's the better algorithm for using temporal locality:

```
BLOCKMATRIXMULTIPLY($A, B, C, n$)
1  for $i \leftarrow 1$ to $n/b$
2       do for $j \leftarrow 1$ to $n/b$
3           for $k \leftarrow 1$ to $n/b$
4               do BLOCKMULT($A, B, C, i, j, k, b$)
```

BLOCKMULT is a triply nested loop matrix multiplication which multiplies matrices of size $b$. We can choose to store $A$ and $B$ in row or column major order. We choose $b$ to be the largest value such that three matrices of size $b \times b$ fit in cache. Therefore, $Z = \Theta(b^2)$. A $b \times b$ matrix fits in $b + b^2/L$ cache lines. Due to the tall cache assumption, we know that $Z > L^2$; therefore, $b > L$ and we can fit $\Theta(b)$ lines in cache when $b = \sqrt{Z}$. If we didn't have the tall cache assumption, even with $b = \sqrt{Z}$, we may not be able to fit an entire $b \times b$ matrix in cache if it is in column major or row major order.

Now lets calculate the total number of cache misses: Each call to BLOCKMULT causes $\Theta(b + b^2/L) = \Theta(\sqrt{Z} + Z/L) = \Theta(Z/L)$ cache misses. There are $\Theta(n^3/b^3) = \Theta(n^3/Z^{3/2})$ calls to BLOCKMULT for a total of $\Theta(n^3/L\sqrt{Z})$. In addition, just initializing $C$ causes $1 + n^2/L$ cache misses. Therefore, the total number of cache misses is $\Theta(1 + n^2/L + n^3/(L\sqrt{Z}))$. For large matrices, the last term dominates.

## 4 Looking Ahead to Next Lecture

This matrix multiplication algorithm is a ***cache-aware*** algorithm. We chose $b$ as $\sqrt{Z}$; therefore, we had to know the value of $Z$ — the size of cache — in order to write this algorithm. In many cases, you may not know the size of cache. Also, modern machines have many levels of caches. In that case, you either have to decide to optimize at only one level of cache or you have to make your algorithm very complex in order to block at all the different levels. You also have issues when more than one program is running on the machine and sharing the cache — you may not know how large your portion of the cache is.

Next week's paper is about ***cache-oblivious algorithms***. We will still use the ideal cache model, but we are not allowed to use the parameters $Z$ and $L$ in our design of algorithms. We will still use them in the analysis of algorithms. To get you thinking about it, consider the recursive matrix multiplication algorithm you saw in class on Wednesday (but without the parallelism) and assume that $A$ is stored in row-major order and $B$ is stored in column major order. Can you analyze the number of cache misses for that algorithm?

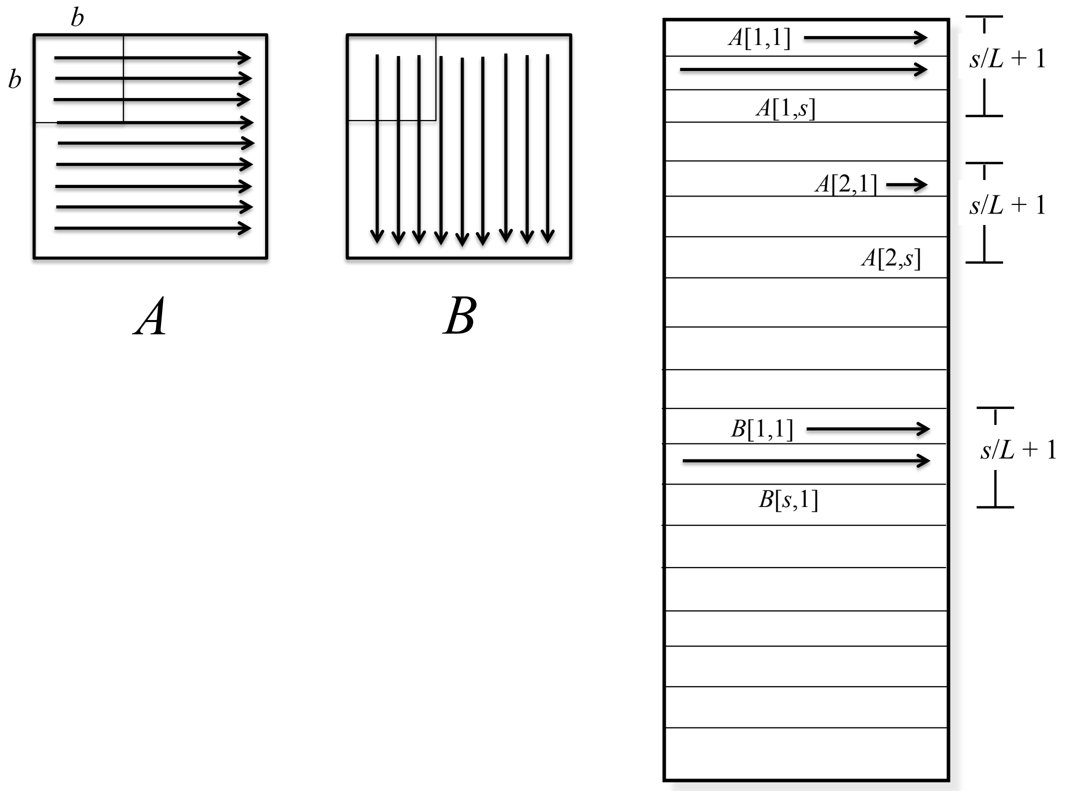


**Figure 2**: Blocked Layout