Cache Oblivious Algorithms

Neel Kamal
Zhang JiaHui
Singapore MIT Alliance

A Parallel Project Report

Abstract

This project presents cache-optimal algorithms for Large Integer Multiplication, Floyd All Pair Shortest Path, and Longest Common Sequence. These algorithms are **cache oblivious:** no variables dependent on hardware parameters, such as cache size and cache-line length, need to be turned to achieve optimality. For a cache with size Z and cache-line length L where $Z = \Omega(L^2)$ the number of cache misses for two large integers (of lengths m and n respectively) multiplication is $Q(m,n) = \Theta(\frac{m}{L} + \frac{n}{L} + \frac{m+n}{L} + \frac{mn}{LZ})$. A good application of the Large Integer Multiplication is the RSA encryption and decryption algorithm. Then two cache-oblivious Dynamic Programming algorithms are proposed and analyzed. The number of caches misses for Floyd All Pair Shortest Path of a graph with n nodes is $Q(n) = \Theta(n^2 + \frac{n^3}{L})$. The number of cache misses for Longest Common Sequence of two strings (of lengths m and n respectively) is $(m,n) = \Theta(1 + 2m + \frac{mn}{\sqrt{Z}} + \frac{mn}{L})$.Also, a generalized cache oblivious approach for Dynamic Programming algorithms is given.

We introduce an "ideal-cache" model to analyze the algorithms, and also we implement a "cache-simulator" to simulate the actual behavior of the "ideal-cache" used in our analysis. Finally, the experimental results for the proposed algorithms using cache-simulator are shown in diagrams.

Contents

1 Introduction **1**

2 Large Integer Multiplication **3**

2.1 Large-Int-Mul-1 and Large-Int-Mul-2 3

2.2 Large Integer Multiplication - the RSA Algorithm 8

3 Cache-Oblivious Dynamic Programming Algorithms **9**

3.1 Floyd All Pair Shortest Path 9

3.2 Longest Common Sequence 13

3.3 General Cache Oblivious Approach for Dynamic Programming 16

4 Results **17**

4.1 Cache Simulator 17

5 Results **19**

5.1 Theoretical Results 19

5.2 Experimental Results 19

5.3 Comparison of the Results 20

5.4 Anomalies 24

6 Conclusions and Future Work **25**

List of Figures

3.1 Optimal results matrix d for the sub-problem in iteration k of the Floyd Algorithm 10
3.2 Optimal results matrix d for the sub-problem in iteration k of the Floyd-Cache-Oblivious Algorithm 11
3.3 Solving LCS by Dynamic Programming 13

5.1 Experimental Result of Large Integer Multiplication 20
5.2 Experimental Result of Floyd Algorithm 21
5.3 Experimental Result of Longest Common Sequence 21
5.4 Comparing results of Large Integer Multiplication 22
5.5 Comparing results of Floyd Algorithm 23
5.6 Comparing results of Longest Common Sequence 23

Chapter 1

Introduction

Resource-oblivious algorithms which use resource effectively offer advantages of simplicity and portability over resource-aware algorithms whose resource usage must be programmed explicitly. In this project, we propose several "**cache-oblivious**" (1) algorithms that use cache as effectively as "cache-aware" algorithms.

We use the (Z, L) **ideal cache model** (1) to study the cache complexity of algorithms, and it consists of a computer with a two-level memory hierarchy: an ideal cache of Z words and an arbitrarily large main memory. The word size is assumed to be constant, since the particular constant does not affect the asymptotic analyses. The cache is partitioned into **cache lines**, each consisting of L consecutive words which are always moved together between cache and main memory. We shall generally assume that the cache is **tall** (1), which means:

$$Z = \Omega(L^2) \tag{1.1}$$

which also applies in practice.

If the processor references some word that resides in the cache, a **cache hit** occurs, otherwise, a **cache miss** occurs, and the line containing that word is fetched from main memory into the cache. The ideal cache is also **fully associative** (2) cache lines can be stored anywhere in the cache.

In the analysis, the algorithms are analyzed in terms of two measures. An algorithm with an

input of size n is measured by its **work complexity** $W(n)$ - its conventional running time in a **RAM** model (3) and its **cache complexity** $Q(n, Z, L)$. The cache complexity is the number of cache misses it incurs as a function of the cache size Z and line length L of the ideal cache. Since Z and L are clear from context, we simply use $Q(n)$ to denote the cache complexity.

In Chapter 2, the cache-oblivious Large Integer Multiplication algorithm is presented and analyzed. In Chapter 3, the cache-oblivious approach for Dynamic Programming algorithms is introduced, and two examples are given: Floyd All Pair Shortest Path and Longest Common Sequence. In Chapter 4, the implemented "Cache-Simulator" is introduced. In Chapter 5, the experimental results for the above mentioned algorithms using our implemented cache-simulator are shown, and Chapter 6 offers some concluding remarks and suggests some future work.

Chapter 2

Large Integer Multiplication

2.1 Large-Int-Mul-1 and Large-Int-Mul-2

This chapter describes and analyzes cache-oblivious algorithms for multiplying a large integer of length m with another large integer of length n. The total work done is $\Theta(mn)$ (4) and the cache complexity is $Q(m,n) = \Theta(\frac{m}{L} + \frac{n}{L} + \frac{m+n}{L} + \frac{mn}{LZ}$ Suppose we have two large integers of lengths m and n respectively, and we assume $m > n$. We append zeros to the left hand side of the shorter integer to make both of them have length m. Very obviously, after this operation, we have two large integers of length m each, and the result of multiplication is the same as that of the original problem. Then we solve the problem of multiplying two large integers of length m in a divide and conquer manner, and the final result should have length 2m as illustrated below.

```
            ( A1 A2 )
          x ( B1 B2 )
        ----------------
          ( A2 x B2 )
        ( A1 x B2 )
      ( A2 x B1 )
    ( A1 x B1 )
------------------------------
    ( *** Final Result *** )
```

From the above diagram, actually we divide the multiplicand and multiplier each into two equal-length (m/2) parts. Multiplicand A is split into A1 and A2, and multiplier B is split

into B1 and B2. Then we solve the 4 sub problems: A2 x B2, A1 x B2, A2 x B1, A1 x B1, and combine the results into the Final result of length 2m. When solving the sub problems, we can further divide and conquer. Therefore, we can respectively divide and conquer the problem into smaller and smaller ones, until m = 1, in which case the only two digits are multiplied. We call this algorithm Large-Int-Mul-1. Although this straightforward divide and conquer algorithm contains no tuning parameters, it uses cache optimally. In practice, divide-and-conquer can stop before m reaches 1, because once a sub-problem fits into the cache, its smaller sub-problems can be solved in cache with no further cache misses.

Theorem 1 *The Large-Int-Mul-1 algorithm uses $\theta(m^2)$ work and incurs $Q(m) = \theta(\frac{m^2}{LZ} + \frac{m}{L})$ cache misses if $m > n$, or uses $\theta(n^2)$work and incurs $Q(n) = \theta(\frac{n^2}{LZ} + \frac{n}{L})$ cache misses if $n > m$.*

Proof Proof We assume $m > n$. It can be shown by induction that the total work of Large-Int-Mul-1 is $\theta(m^2)$. Now, let's analyze the number of cache misses.

Case 1 $m > \frac{\alpha Z}{4}$

This case is the most intuitive: the problem size does not fit into the cache when m is big enough. The multiplier and multiplicand are of length m each, and the final result should be kept in an array of length 2m, so the problem working size should be 4m in total. When $m > \frac{\alpha Z}{4}$, the problem is too large to fit into the cache, where Z is the cache size, and is a positive constant. The cache complexity can be described by the recurrence

$$Q(m) \leq \begin{cases} \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{m}{L} \rceil + \lceil \frac{2m}{L} \rceil), & \text{,if m } \epsilon [\frac{\alpha Z}{8}, \frac{\alpha Z}{4}] \\ 4Q(\frac{m}{2}) + O(1), & \text{,otherwise.} \end{cases} \tag{2.1}$$

The base case arises as soon as the problem working size fit in cache. The total number of lines used is $\theta(\lceil \frac{m}{L} \rceil + \lceil \frac{m}{L} \rceil + \lceil \frac{2m}{L} \rceil)$. The only cache misses that occur during the remainder of the recursion are $\theta(\lceil \frac{m}{L} \rceil + \lceil \frac{m}{L} \rceil + \lceil \frac{2m}{L} \rceil)$ cache lines required to bring the multiplier, multiplicand and result array into cache. In the recursive cases, we pay cache misses of the recursive calls plus $O(1)$ cache misses for the overhead of manipulating sub-results. Now let's solve the recurrence.

Suppose after k recursive steps, $m \longrightarrow \frac{m}{2^k}\ \epsilon[\frac{\alpha Z}{8}, \frac{\alpha Z}{4}]$, so

$$\frac{\alpha Z}{8} \leq \frac{m}{2^k} \leq \frac{\alpha Z}{4}]$$

$$\Rightarrow \frac{8m}{\alpha Z} \geq 2^k \geq \frac{4m}{\alpha Z} \tag{2.2}$$

$$Q(m) \leq 4^k Q(\frac{m}{2^k}) \leq 4^k \Theta(\lceil \frac{\frac{m}{2^k}}{L} \rceil + \lceil \frac{\frac{m}{2^k}}{L} \rceil + \lceil \frac{\frac{2m}{2^k}}{L} \rceil) = 4^k \Theta(\lceil \frac{\frac{4m}{2^k}}{L} \rceil) = \Theta(\frac{16m^2}{LZ}) \tag{2.3}$$

Case 2 $m < \frac{\alpha Z}{4}$

From the choice of α, the problem can fit into the cache. Therefore, we have

$$Q(m) = \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{m}{L} \rceil + \lceil \frac{2m}{L} \rceil) = \theta(\frac{4m}{L}) \tag{2.4}$$

Combined (2.3) and (2.4) together, we have the cache complexity for Large-Int-Mul-1 is

$$Q(m) = \theta(\frac{m^2}{LZ} + \frac{m}{L}) \tag{2.5}$$

In the situation when $n > m$, by similar proof, we can show that Large-Int-Mul-1 or uses $\theta(n^2)$ work and incurs $Q(n) = \Theta(\frac{n^2}{LZ} + \frac{n}{L})$cache misses.

□

Now let's consider the second cache-oblivious approach to Large Integer Multiplication, the general divide and conquer idea is the same, but we do not append zeros to the left hand side of the shorter integer to make them equal length this time. The multiplicand A is of length m and the multiplier B is of length n. If $m > n$, we divide multiplicand A into two parts of length m/2 A1 and A2, then we solve the 2 sub problems: A2 x B, A1 x B and combine the sub-results together. Of, if $n > m$, we divide the multiplier B into two parts of length n/2 B1 and B2, then we solve the 2 sub problems A x B1, A x B2 and combined the sub-results together. We recursively divide and conquer until the problem size can fit into the cache. We call this algorithm Large-Int-Mul-2.

Theorem 2 *The Large-Int-Mul-2 algorithm uses $\theta(mn)$ work and incurs $Q(m,n) = \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ})$ Cache misses.*

Proof It can be shown by induction that the total work of Large-Int-Mul-2 is $\Theta(mn)$. Now, let's analyze the number of cache misses.

Case 1 $m, n > \alpha Z$

This case is most intuitive: the problem size is too large to fit into cache, where α a positive constant and Z is the cache size. The cache complexity can be described by the following recurrence.

$$Q(m,n) \leq \begin{cases} \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ}), & \text{,if m,n } \epsilon [\frac{\alpha Z}{2}, \alpha Z] \\ 2Q(\frac{m}{2}, n) + O(1) & \text{else if}(m > n); \\ 2Q(m, \frac{n}{2}) + O(1) & \text{otherwise.} \end{cases} \tag{2.6}$$

The base case arises as soon as the problem working size fit in cache. The total number of lines used is $\theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ})$. The only cache misses that occur during the remainder of the recursion are $\theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ})$ cache lines required to bring the multiplier, multiplicand and result array into cache. In the recursive cases, we pay cache misses of the recursive calls plus $O(1)$ cache misses for the overhead of manipulating sub-results. Now let's solve the recurrence.

Suppose after k1 recursive steps, m$\rightarrow \frac{m}{2^{k_1}} \epsilon [\frac{\alpha Z}{2}, \alpha Z]$, and after k2 recursive steps, n$\rightarrow \frac{n}{2^{k_2}} \epsilon [\frac{\alpha Z}{2}, \alpha Z]$. So

$$\frac{\alpha Z}{2} \leq \frac{m}{2^{k_1}} \leq \alpha Z \tag{2.7}$$

$$\Rightarrow \frac{2m}{\alpha Z} \geq 2^{k_1} \geq \frac{m}{\alpha Z} \tag{2.8}$$

And

$$\frac{\alpha Z}{2} \leq \frac{m}{2^{k_2}} \leq \alpha Z \tag{2.9}$$

$$\Rightarrow \frac{2n}{\alpha Z} \geq 2^{k_2} \geq \frac{n}{\alpha Z} \tag{2.10}$$

Then

$$Q(m,n) \leq 2^{k_1}2^{k_2}Q(\frac{m}{2^{k_1}}, \frac{n}{2^{k_2}}) = 2^{k_1}2^{k_2}\Theta(\frac{2(\frac{m}{2^{k_1}} + \frac{n}{2^{k_2}})}{L}) = \Theta(\frac{2(m2^{k_2} + n2^{k_1})}{L}) = \Theta(\frac{mn}{LZ}) \tag{2.11}$$

Case 2 *($m < \alpha Z$ and $n > \alpha Z$) or ($n < \alpha Z$ and $m > \alpha Z$)*

In this case, one integer can fit into the cache while the other can not. Suppose $m < n$, then we have the following recurrence

$$Q(m,n) \leq \begin{cases} \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ}), & \text{,if n } \epsilon [\frac{\alpha Z}{2}, \alpha Z] \\ 2Q(m, \frac{n}{2}) + O(1) & \text{otherwise.} \end{cases} \tag{2.12}$$

Solve the recurrence, and we have

$$Q(m,n) = \Theta(\frac{mn}{LZ} + \frac{n}{L}) \tag{2.13}$$

Similarly, if $n < m$, we have

$$Q(m,n) = \Theta(\frac{mn}{LZ} + \frac{m}{L}) \tag{2.14}$$

Case 3 $m, n < \alpha Z$

Now, the problem size is small enough to fit in cache, therefore we have the cache complexity as

$$Q(m,n) = \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil) \tag{2.15}$$

Combine the results (2.11), (2.13), (2.14) and (2.15), we have shown that the Large-Int-Mul-2 algorithm has cache complexity of

$$Q(m,n) = \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ}) \tag{2.16}$$

□

2.2 Large Integer Multiplication - the RSA Algorithm

We have presented two cache oblivious algorithms for Large Integer Multiplication, now we would like to discuss their possible applications. One good application is RSA encryption and decryption algorithm. The following is a summary of the RSA algorithm (**?**).

- n = pq where p and q are distinct primes.
- ϕ = (p-1)(q-1)
- $e < n$ such that gcd (e, ϕ)=1
- d = $e^{-1} \mod \phi$.
- c = $m^e \mod n$.
- m = $c^d \mod n$.

The high-lighted two operations n = pq and ϕ = (p-1)(q-1) can be computed using the Large-Int-Mul-2 algorithm. With the fast advance in IT technology, we will need higher and higher security levels. Therefore, the number of digits for n in RSA can become larger and larger. Especially in small Palm-Computer applications, the cache size is extremely small. So, we can save a great number of cache misses by using the cache-optimal Large-Int-Mul-2 algorithm proposed.

Chapter 3

Cache-Oblivious Dynamic Programming Algorithms

In this Chapter, two cache-oblivious algorithms are presented and analyzed: Floyd All Pair Shortest Path (Floyd-Cache-Oblivious) and Longest Common Sequence (LCS-Cache-Oblivious). The Floyd-Cache-Oblivious uses $\Theta(n^3)$ work and incurs $Q(n) = \Theta(n^2 + \frac{n^3}{L})$ cache misses, where n is the total number of nodes in the graph. The LCS-Cache-Oblivious uses $\Theta(mn)$ work, and incurs $Q(m,n) = \Theta(1 + 2m + \frac{mn}{\sqrt{Z}} + \frac{mn}{L})$ cache misses, where m and n are the lengths of the two strings. There is something common in both algorithms - they are using some Dynamic Programming approach. Therefore, a generalized cache oblivious approach for Dynamic Programming algorithms will be given in the end of this Chapter.

3.1 Floyd All Pair Shortest Path

The proposed Floyd-Cache-Oblivious algorithm solves the problem of finding all pair shortest paths given a graph with n nodes. The original Floyd algorithm can be summarized by the following pseudo code (4):

```
for k=1 to n
    for i=1 to n
        for j=1 to n
            d[i][j][k]=min(d[i][j][k-1],d[i][k][k-1]+d[k][j][k-1]
```

This is a Dynamic Programming approach, let d[i][j][k] represents the shortest path from node i to node j via nodes 1 to k. Therefore, d[i][j][n] is the shortest path from node i to node j. For each iteration of k (the outer most loop index), we need to construct an n x n

matrix to store the optimal results for the sub-problem, and this can be illustrated by the following diagram.

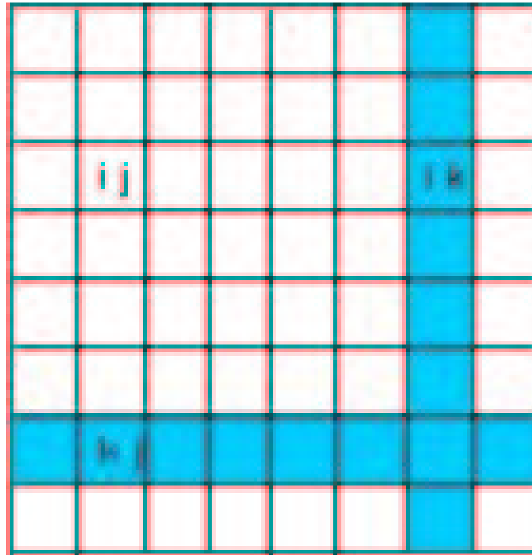

Figure 3.1: Optimal results matrix d for the sub-problem in iteration k of the Floyd Algorithm

In Figure 3.1, the blue colored row is row k (of previous iteration) and the blue colored column is column k (of previous iteration). Therefore, each element dij of iteration k depends on the value of dij, dik and dkj of the previous iteration k-1.

In the propose Floyd-Cache-Oblivious algorithm, we take a divide-and-conquer way to construct the sub-optimal result matrix for each iteration. If the matrix to be constructed is too large to fit into cache, we divide it into 4 equal square parts, each of the size (n/2 by n/2), and then we try to solve the 4 sub problems, i.e. to construct the 4 smaller square (n/2 by n/2) matrix, and then combined the results as shown in Figure 3.2.

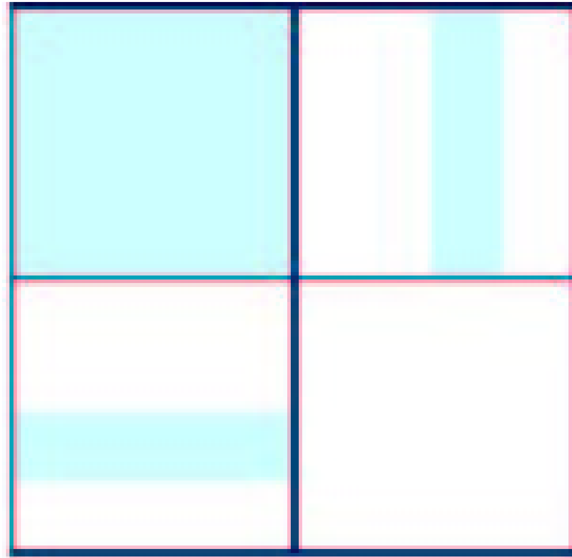

Figure 3.2: Optimal results matrix d for the sub-problem in iteration k of the Floyd-Cache-Oblivious Algorithm

Theorem 3 *The Floyd-Cache-Oblivious algorithm uses $\Theta(n^3)$ work and incurs $Q^{total}(n) = \Theta(n^2 + \frac{n^3}{L})$ cache misses.*

Proof

It can be shown by induction that the total work of Floyd-Cache-Oblivious is $\Theta(n^3)$. Now, let's analyze the number of cache misses.

We have n iterations (k = 1..n), and for each iteration

Case 1 $n > \alpha\sqrt{Z}$

This case is most intuitive: the dimension of the result matrix $n > \alpha\sqrt{Z}$, so the problem size is too large to fit into cache, where α is a positive constant and Z is the cache size. The cache complexity can be described by the following recurrence

$$Q(n) \leq \begin{cases} \theta(\frac{n^2}{L}) & \text{,if n } \epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}] \\ 4Q(\frac{n}{2}) + O(1) & \text{,otherwise.} \end{cases} \tag{3.1}$$

The base case is when n $\epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}]$: the problem can fit into cache. The total number of lines used is $\theta(\frac{n^2}{L})$. The only cache misses that occur during the remainder of the recursion are $\theta(\frac{n^2}{L})$ cache lines required to bring the result n by n matrix and 2 portions (both of length n) of row k and column k. In the recursive cases, we pay cache misses of the recursive calls plus O(1) cache misses for the overhead of manipulating sub-results. Now let's solve the recurrence.

Suppose after k recursive steps,n$->\frac{n}{2^k}\epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}]$

Then

$$Q(n) = 4^k Q(\frac{n}{2^k}) = 4^k \theta(\frac{(\frac{n}{2^k})^2}{L}) = \theta(\frac{n^2}{L}) \tag{3.2}$$

Case 2 $n < \alpha\sqrt{Z}$

This is the case when the dimension of the result matrix is small enough to fit into the cache. The only cache misses are to bring the result matrix (n by n) and two portions (both of length n) of row k and column k.

Therefore, we have cache complexity as

$$Q(n) = \theta(n) \tag{3.3}$$

Combine (3.2) and (3.3), we have the cache complexity for each iteration of k (1..n) to be

$$Q(n) = \theta(n + \frac{n^2}{L}) \tag{3.4}$$

The total cache complexity for n iterations is:

$$Q^{total}(n) = \Theta(n^2 + \frac{n^3}{L}) \tag{3.5}$$

Therefore, it has been shown that $Q^{total}(n) = \Theta(n^2 + \frac{n^3}{L})$ is the cache complexity for the proposed Floyd-Cache-Oblivious algorithm. □

3.2 Longest Common Sequence

The Longest Common Sequence problem is to find the longest common sequence between two strings x of length m and y of length n. The common approach is to use dynamic programming, which is shown in Figure 3.3 (4).

		y1	y2	y3	y4	y5	y6
		B	D	C	A	B	A
x1	A	0	0	0	1	1	1
x2	B	1	1	1	1	2	2
x3	C	1	1	2	2	2	2
x4	B	1	1	2	2	3	3
x5	D	1	2	2	2	3	3
x6	A	1	2	2	3	3	4
x7	B	1	2	2	3	4	4

Figure 3.3: Solving LCS by Dynamic Programming

The main idea is to construct a table c (the yellow portion of Figure 3.3), which contains the optimal results of the sub problems. This table should be constructed using a left to right and top to down manner, and each element depends on the value of its upper, left, left upper elements. For example, in Figure 4, the green cell depends on the 2 blue and 1 red cells (4):

If x[i] == y[j]
  c[i][j] = c[i -1][j -1]+1;
else
  c[i][j] = max{c[i -1][j]; c[i][j -1]};

The proposed LCS-Cache-Oblivious solves the problem in a divide and conquer way, more specifically, it constructs the table c using divide and conquer strategy. The table is of dimension m by n, if m ¿ n, we divide the table into two parts of m/2 by n each, otherwise we divide the table into two m by n/2 parts. Then we solve the 2 sub problems and combine the results. We keep on divide and conquer until the sub table size is small enough to fit into the cache.

Theorem 4 *The LCS-Cache-Oblivious algorithm uses* $\Theta(mn)$ *work and incurs* $Q(m,n) = \Theta(1 + 2m + \frac{mn}{\sqrt{Z}} + \frac{mn}{L})$ *cache misses.*

Proof It can be shown by induction that the total work of LCS-Cache-Oblivious is $\Theta(mn)$. Now, let's analyze the number of cache misses.

Case 1 $m, n > \sqrt{Z}$

This is the case that both dimensions are too large to fit into the cache. The cache complexity can be described by the following recurrence.

$$Q(n) \leq \begin{cases} \theta(\frac{mn}{L}) & \text{,if m,n } \epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}] \\ 2Q(\frac{m}{2}, n) + O(1) & \text{,otherwise,if } (m > n) \\ 2Q(m, \frac{n}{2}) + O(1) & \text{,otherwise.} \end{cases} \tag{3.6}$$

The base case is when m,n $\epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}]$: the problem can fit into cache. The total number of lines used is $\theta(\frac{mn}{L})$. The only cache misses that occur during the remainder of the recursion are $\theta(\frac{mn}{L})$ cache lines required to bring the m by n result table into the cache. In the recursive cases, we pay cache misses of the recursive calls plus O(1) cache misses for the overhead of manipulating sub-results. Now let's solve the recurrence.

Suppose after k1 recursive steps,m$- > \frac{m}{2^{k1}}\epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}]$

Suppose after k2 recursive steps,n$- > \frac{n}{2^{k2}}\epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}]$

Then Then

$$Q(m,n) = 2^{k1}2^{k2}Q(\frac{m}{2^{k1}}, \frac{n}{2^{k2}}) = 2^{k1}2^{k2}\theta(\frac{\frac{m}{2^{k1}}\frac{n}{2^{k2}}}{L}) = \theta(\frac{mn}{L}) \tag{3.7}$$

Case 2 $(m > \sqrt{Z}$ *and* $n < \sqrt{Z})$ *or* $(n > \sqrt{Z}$ *and* $m < \sqrt{Z})$

In this case, one dimension can fit into the cache while the other can not. Suppose $m < n$, then we have the following recurrence

$$Q(n) \leq \begin{cases} \theta(1+m) & \text{,if n } \epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}] \\ 2Q(m, \frac{n}{2}) + O(1) & \text{,otherwise.} \end{cases} \tag{3.8}$$

Solve the recurrence, we have

$$Q(m,n) = \theta(\frac{mn}{\sqrt{Z}}) \tag{3.9}$$

In the case when $n < m$, the recurrence is

$$Q(n) \leq \begin{cases} \theta(1+m) & \text{,if m } \epsilon[\frac{\alpha\sqrt{Z}}{2}, \alpha\sqrt{Z}] \\ 2Q(\frac{m}{2}, n) + O(1) & \text{,otherwise.} \end{cases} \tag{3.10}$$

Solve the recurrence, we have

$$Q(m,n) = \Theta(m) \tag{3.11}$$

Case 3 $m, n < \sqrt{Z}$

This is the case when both dimensions can fit into the cache. Therefore, the cache complexity is

$$Q(m,n) = \Theta(1+m) \tag{3.12}$$

Combine (3.7), (3.9), (3.11) and (3.12) all together, we have

$$Q(m,n) = \Theta(1 + 2m + \frac{mn}{\sqrt{Z}} + \frac{mn}{L}) \tag{3.13}$$

Therefore, it has been shown that LCS-Cache-Oblivious incurs $Q(m,n) = \Theta(1 + 2m + \frac{mn}{\sqrt{Z}} + \frac{mn}{L})$ cache misses.

□

3.3 General Cache Oblivious Approach for Dynamic Programming

Dynamic Programming is often used to solve problems with the following two characterizes

- to find an optimal solution

- sub-problems overlap

And there are generally 2 approaches for dynamic programming approaches

- bottom up (by recursion usually)

- top down but with a table to memorize earlier solutions

Our general cache oblivious approach for dynamic programming is based on the "top down but with a table to memorize earlier solutions".

As illustrated in the two examples Floyd-Cache-Oblivious and LCS-Cache-Oblivious, we can use a divide and conquer way to build the result table. More specifically, if the dimensions of the table is too large to fit into the cache, we can chop the table into smaller parts (divide into sub problems), and try to construct the smaller sub tables (solve the sub problems), and then combine the results. We can keep on divide and conquer until the dimensions of the table to be constructed are small enough to be put into the cache properly. Of course, sometimes, there exists a certain order that we should follow when constructing the sub tables. Also, special care should be taken when solving specific problems. Therefore, divide and conquer is a generally cache oblivious approach for dynamic programming, but there are small variations from problem to problem.

Chapter 4

Results

4.1 Cache Simulator

The idea is to be able to count the number of cache misses. So instead of actually creating the storage space and doing the read and write operation from the dummy cache, we implemented a mock cache. So actually no data computation storage/read is done but instead a count and mapping of it is stored. So that finally we can get exact number of cache misses.

Sample program:

```
#define S (2 * 1024)       // cache capacity
#define L (32)             // cache line size
#define A (4)              // cache associativity
#define E (sizeof(double)) // matrix element size
#define N (100)            // matrix size

void main ()
{
    Cache *cache = new CacheT<S, L, A>();
    cache->Invalidate();

    Array2D a(N, N, E, cache);
    Array2D b(N, N, E, cache);
    Array2D c(N, N, E, cache);
```

```
    Int A[N][N];
    Int B[N][N];
    Int C[N][N];

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                  //For the Actual computation
          C[i][j] += A[i][k] * B[k][j];

          //for the simulation
          c(i, j, 3) += a(i, k, 1) * b(k, j, 2);

    stringstream f;
    f << N << "\t" << "ijk";
    cout << cache->Statistics(f.str());
}
```

So we can see that, there is difference between a simulation code and actual code to do the computation.

Chapter 5

Results

Three cache oblivious algorithms have been proposed in Chapter 2 and 3. These algorithms may not perform the best in terms of the total work complexity, but certainly they are the most famous approaches as well as most convenient and commonly used ways to do what they intend to do. Most importantly, they are all cache-oblivious, and they use the cache in a optimal way. Now, let us present the experimental results based on the implemented cache simulator of the three proposed cache oblivious algorithms.

5.1 Theoretical Results

Listing the Results from the theoretical analysis:

Problems	**Number of Cache Misses**
Large Integer Multiplication	$Q(m,n) = \theta(\lceil \frac{m}{L} \rceil + \lceil \frac{n}{L} \rceil + \lceil \frac{m+n}{L} \rceil + \frac{mn}{LZ})$
All-pair shortest Paths	$Q(n) = \Theta(n^2 + \frac{n^3}{L})$
Longest Common Sequence	$Q(m,n) = \Theta(1 + 2m + \frac{mn}{\sqrt{Z}} + \frac{mn}{L})$

5.2 Experimental Results

Experimental results are shown using three graphs. Each of these graphs represent one problem. We list the problem size in the table below. It is interesting to note that how the number of cache misses decreases with increase in the cache line size. Also its interesting to note that the decrease is of the similar order as the theoretical analysis. This becomes more clear in the next section where we show the ratio of the two results.

Description of Axis:

X Axis = Cache Line Size

Y Axis = Number of Cache Misses

Problems	Size of inputs
Large Integer Multiplication	Length of integers = **1000** digits
All-pair shortest Paths	Number of nodes = **100**
Longest Common Sequence	Number of characters in Sequence = **1000**

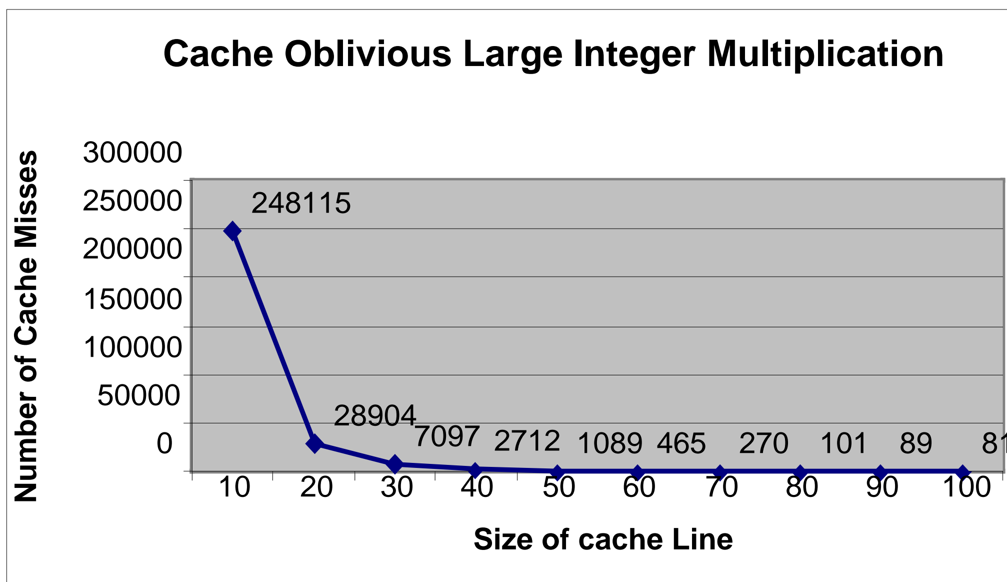

Figure 5.1: Experimental Result of Large Integer Multiplication

5.3 Comparison of the Results

As we have both the results (theoretical and experimental) we now present the comparison of the two.

Description of Axis:

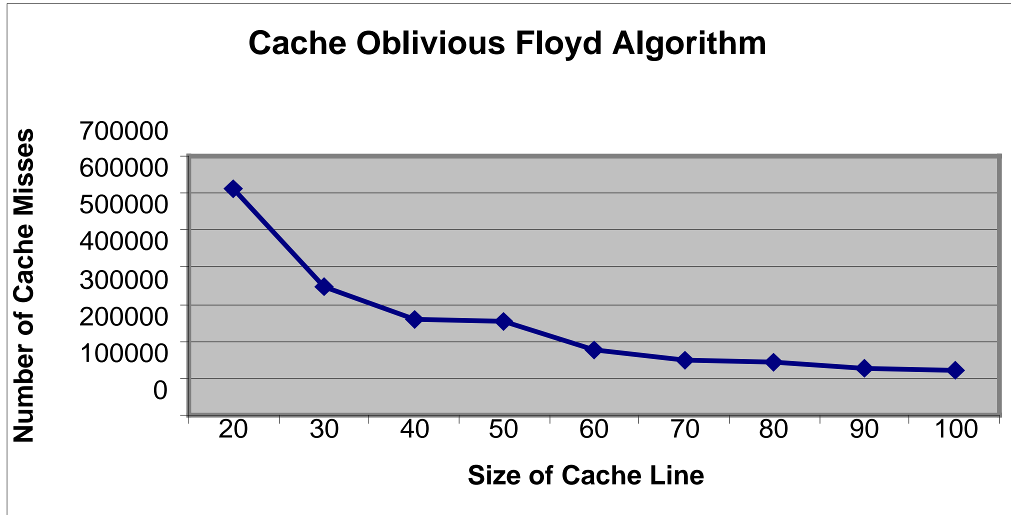

Figure 5.2: Experimental Result of Floyd Algorithm

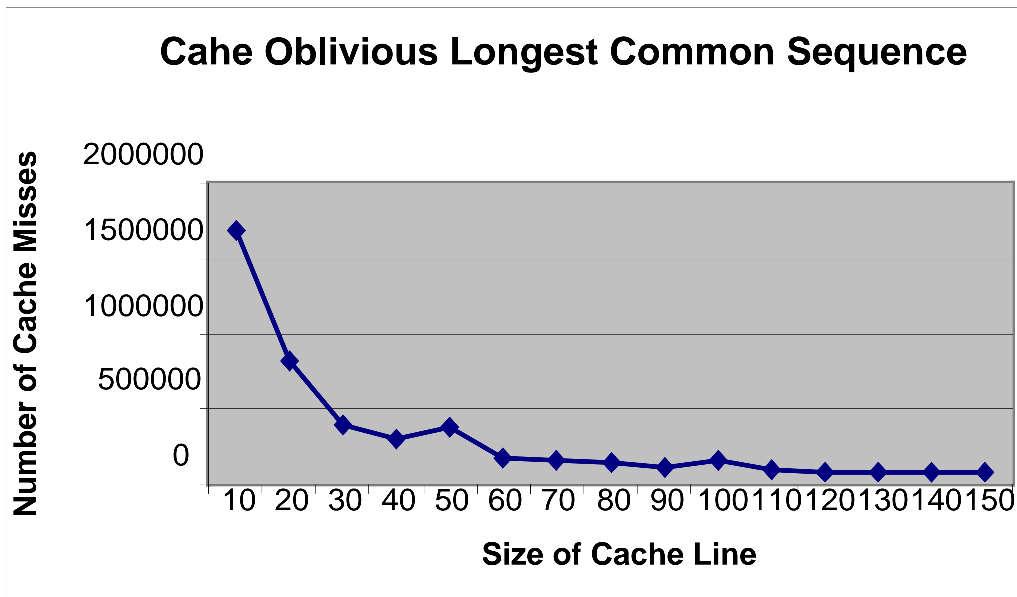

Figure 5.3: Experimental Result of Longest Common Sequence

X Axis = Cache Line Size

Y Axis = $\frac{\textbf{Number of Cache Misses by Theoretical Result}}{\textbf{Number of Cache Misses by Experimental Result}}$

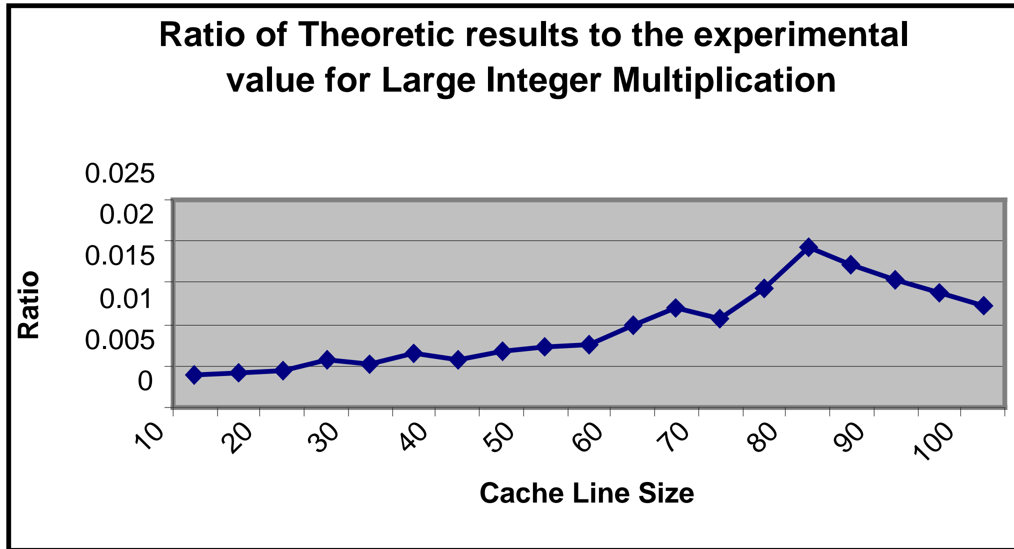

Figure 5.4: Comparing results of Large Integer Multiplication

Explaining the comparison

For example:

Problem: Large Integer Multiplication

L= Cache Line Length

Case 1: L = 20

Theoretical Result = O (1000/8) Simulator Result = 28904 **Ratio = 0.0041**

Case 2: L=30

Theoretical Result = O (1000/27) Simulator Result = 7097 **Ratio = 0.0044**

Case 3: L=40

Theoretical Result = O (1000/64) Simulator Result = 2712 **Ratio = 0.0052**

...

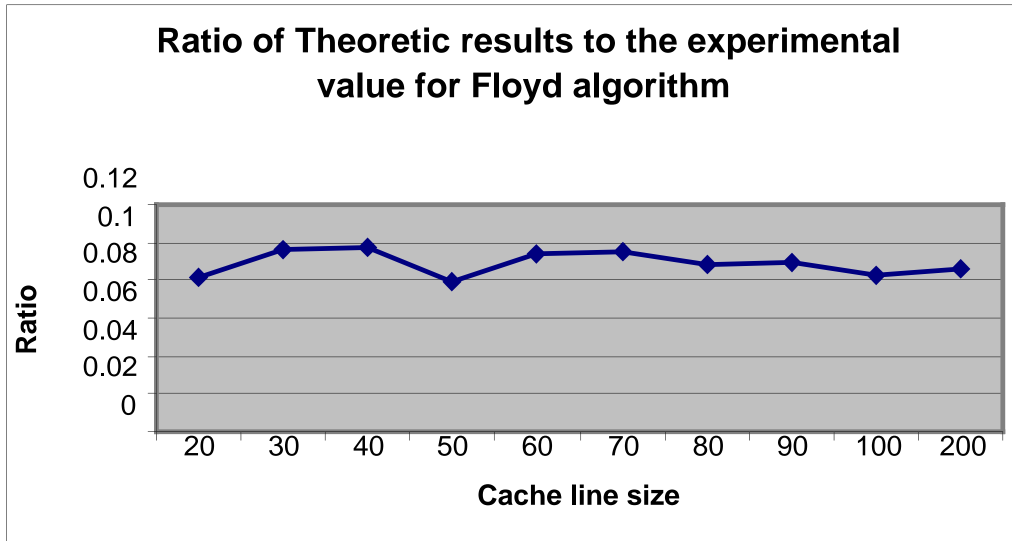

Figure 5.5: Comparing results of Floyd Algorithm

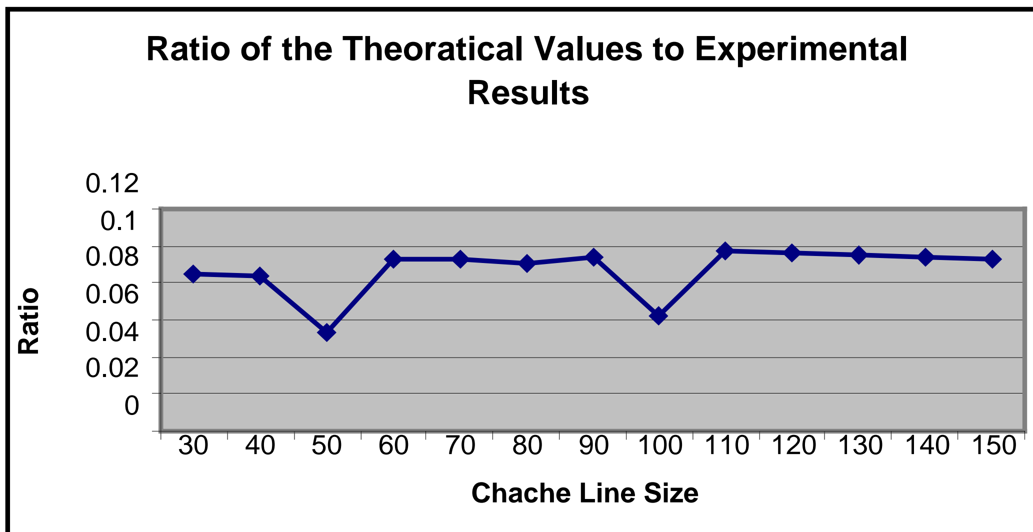

Figure 5.6: Comparing results of Longest Common Sequence

5.4 Anomalies

There are bound to have anomalies between theoretical and experimental results. It's mostly because we do not care for finer details (we use the O-notation or the Θ-notation) while doing the theoretical analysis while in experimentation we have to be exact. To be more specific, in this project we have implemented our own cache simulator which by no means is the same as the actual cache. So the anomalies do not surprise us. In fact its interesting to note that even the anomalies are similar in nature in all the three graphs. Note that when the cache size is multiple of 50 (like 50, 100 etc) we have a peak in the graphs of the LCS and Floyd problems. This may be because of many reasons. One suggested reason is that the cache line does not behave properly for a multiple of 50.

Another anomaly occurs in large integer multiplication problem. In fact the ratio curve goes smooth except for a few points at cache line size around 80-90, i.e. the ratio at these points are slightly higher than average, but the difference is still within 0.01.

Chapter 6

Conclusions and Future Work

In this research projects, several cache-oblivious algorithms are presented and analyzed based on the idea cache model. *Large-Int-Mul-1* and *Large-Int-Mul-2* target the problem of large integer multiplication, and they are cache-optimal. While, the total work done to multiply two large integers of lengths m and n is not yet optimal, and there is some algorithm based on Fast Fourier Transform which can achieve a better performance than $O(mn)$ in literature. One of possible future work is to develop the cache-oblivious large integer multiplication algorithm which does less total work based on Fast Fourier Transform.

Also, the general model of cache oblivious approach for Dynamic Programming has been proposed, and two examples are given: *Floyd-Cache-Oblivious* algorithm and *LCS-Cache-Oblivious algorithm*. The general model is based on the "top-down" approach of Dynamic Programming, and it uses a divide-and-conquer way to build a table to memorize earlier solutions. The base case occurs when the problem working space can fit into the cache properly. How about other algorithms developed in the ideal cache model? Can we make them cache-oblivious? The answers to these questions suggest another field of possible future work: to determine the range of practicality of cache-oblivious algorithms.

We realized that it is good to have simulation result to verify the bounds that we proposed. And it was particularly very interesting and satisfying to see the close match between the two kind of results.

Bibliography

[1] H. P. S. R. Matteo Frogo, Charles E. Leiserson. Cache Oblivious Algorithm Extended Reference. 1

[2] D. A. P. J. L. Hennessy, *Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 2nd edition*. 1996. 1

[3] H. J. Aho, A.V. and J. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-wesley publishing company, 1974. 1

[4] R. L. R. C. S. Thomas H. Cormen, Charles E. Leisersion, *Introduction to Algorithms*. MIT Press, 2002. 2.1, 3.1, 3.2, 3.2