CS 740, Fall 1996
Homework Assignment 3

Assigned: Oct. 15
Due: Thurs., Oct. 24, 1:00PM

1. Logistics

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single report for your entire group, identifying all of the group members.

Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory.

You will hand in a hard copy document for this assignment. Formatted text is preferred to hand written.

2. Cache Performance

The first set of tasks involves implementing a procedure that optimizes the use of the processor cache(s). You may use any machine for your experiments, as long as it has a cache. You will want to find a lightly loaded machine, since any context switching can affect cache performance. Be sure to document what kind of machine you use, and whatever you can determine about its cache (capacity, associativity, line size, write policy), either through technical documentation or your own experiments.

Note: We are more interested in caches for this assignment than in the particular application. Your writeup should focus on how cache behavior affects performance. You should give qualitative explanations for the performance variations you observe. Note also that good experimental method is essential to obtaining reliable measurements. You should design your experiments carefully and document your method fully.

Imagine you are assigned the task of writing an efficient matrix multiply routine for double precision floating point numbers. The routine takes two $n \times n$ arrays `a` and `b`, in row major order, and stores the matrix product in an array `c`.

A straightforward implementation would be:

```
static void matrix_mult(double *a, double *b, double *c, int n)
{
  int i,j,k;
  /**
     i: selects row of a, c
     j: selects col of b, c
     k: selects col of a, row of b
  **/
  int size = n*n;
```

```
  for (i = 0; i < size; i++)
    c[i] = 0.0;
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
        c[j + i*n] += a[k + i*n] * b[j + k*n];
}
```

When compiled with optimization level `O3`, `gcc` generates good code. It converts all of the multiplications in the array index calculations into additions.

You are given the following design parameters:

- Your routine will be used for both $n = 16$ and $n = 256$.
- Your routine will be called either with a "warm" cache, e.g., when the same argument matrices are multiplied repeatedly, or with a "cold" cache, i.e., where no part of the argument matrices reside in the cache when the routine is called.

These two parameters yield four possible operating conditions for your routine.

2.1. Cache Effects

The first set of optimizations will involve modifications affect only the cache performance, and not the low-level code generated.

Task 1A: Loop Nestings

There are 6 permutations of the nestings of loop indices `i`, `j`, and `k`. Devise a set of experiments to determine the performance of each possible loop nesting for each of the four conditions described above.

Some advice:

- It is helpful to express performance in units of "microseconds per multiply-accumulate." That way you can compare the performance across different values of n.
- You may find the routines in the `ftime` library useful, either the timing function `time_funct`, or the lower level calls `start_timer` and `get_timer`.
- You will need to be clever in devising a test for the cold cache case. You need to ensure that each call to the matrix multiplication procedure accesses arrays that are not cache resident.
- Even though the Unix timing routines yield time values in microseconds, the resolution of the clock is much coarser. You need to time your functions over a long enough interval to overcome this limitation.

Document your experimental method and your findings. Give qualitative explanations for the variations you see in your measurements. Factors that affect cache performance include:

Total capacity. Typical cache sizes range from 16KB to 256KB.

Line Size effects. Accessing one word causes the cache to be loaded with some adjacent words.

Collisions. Suboptimal performance may occur due to the addresses being referenced. This is particularly true for direct mapped caches.

Task 1B

Try embedding your arrays in ones that are slightly larger, but not powers of 2. Even though this will increase the number of multiply-accumulates, you may find a net improvement in performance. [You may assume that this embedding occurs before the multiply routine is called, so you need not include its cost in your measurements.] Document your findings and explain any performance changes you observe.

Task 1C

Modify the routine(s) to transpose array `b` before computing the matrix product. Rerun your experiments to see what effect this has on performance. [You must include the cost of performing the transposition in your measurements]. Explain how this transposition affects the performance.

2.2. Other Optimizations

The next set of optimizations change the set of memory locations that get accessed, and possibly change some of the overhead computations.

Task 2A

For some of the loop nestings, it would be possible to accumulate the sums in a register rather than in memory. Try out this approach for the relevant loop nestings and for the different operating conditions. Compare these results to the measurements in Task 1A–C. Explain the cache effects that affect their performance.

Task 2B

GCC does a good job at converting the array index calculations into efficient code, but you may be able to do better. To avoid wasting too much time, stay within the following guidelines:

- Keep all of your code in C.
- Concentrate on the inner loop code.

Describe which combination of features gives the best performance. What is the net improvement over the original code?

3. Parallel prefix and carry-lookhead addition

This problem studies parallel prefix computations, and the important special case of carry-lookahead addition.

3.1. Ripple adders

Suppose that we want to add two n-bit numbers x and y, where

$$\begin{aligned} x &= x_{n-1}, x_{n-2}, \ldots, x_0, \text{ and} \\ y &= y_{n-1}, y_{n-2}, \ldots, y_0. \end{aligned}$$

If we were to add these numbers by hand, the least significant bit of the sum would be computed by adding the least significant bits of x and y, i.e., $s_0 = x_0 \hat{} y_0$, where, following the C convention, $\hat{}$ denotes the bitwise exclusive-or operator. We would also compute the carry into the next bit position, $c_1 = x_0 \ \& \ y_0$, where & denotes the logical and operator. Bit s_1 of the sum is then given by $s_1 = x_1 \hat{} y_1 \hat{} c_1$, and the carry into bit position 2 is given by $c_2 = (x_1 \ \& \ y_1) \mid (x_1 \ \& \ c_1) \mid (y_1 \ \& c_1)$. In general, s_i and c_{i+1} are given by $s_i = x_i \hat{} y_i \hat{} c_i$ and $c_{i+1} = (x_i \ \& \ y_i) \mid (x_i \ \& \ c_i) \mid (y_i \ \& \ c_i)$.

A *ripple-carry adder* implements the "by-hand" algorithm for adding two numbers. An n-bit ripple-carry adder consists of n single-bit adders, each taking as input three bits, x_i, y_i, and c_i, and computing two outputs, s_i and c_{i+1}. These adders are strung together as shown below.

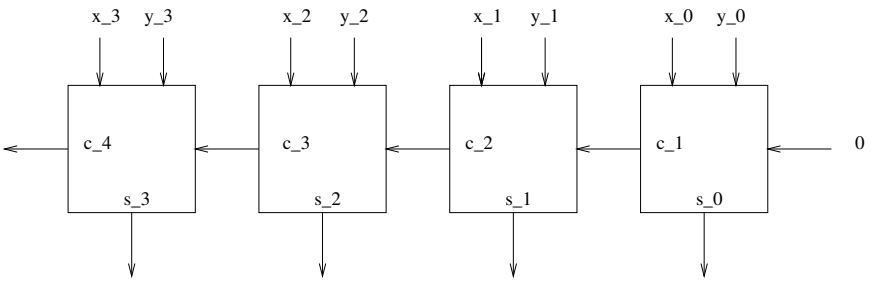

Figure 1: A 4-bit ripple-carry adder.

One disadvantage of the ripple-carry adder is that there is a path of length n through the logic of the circuit from x_0 (or y_0) to s_{n-1}. We say that this circuit has depth n. For large integers ($n = 64$) this path would be the critical path in the integer ALU.

3.2. Parallel prefix

The key to constructing a circuit with smaller depth is to view addition as a *prefix computation*. A prefix computation takes an input sequence $a_0, a_1, \ldots, a_{n-1}$, and a binary associative operator $\oplus$,

and produces an output sequence $b_0, b_1, \ldots, b_{n-1}$, where $b_i = a_0 \oplus a_1 \oplus \cdots \oplus a_i$. (An operator $\oplus$ is associative if $(c \oplus d) \oplus e = c \oplus (d \oplus e)$, for any c, d, and e.) As an example, when $\oplus$ is integer addition, b_i is the sum of the numbers a_0 through a_i. Similarly, when $\oplus$ is the min operator, b_i is the minimum of the values a_0 through a_i.

A circuit for computing the sequence b can be derived from the following recurrence

$$b_i = \begin{cases} a_0 & i = 0 \\ b_{i-1} \oplus a_i & i > 0 \end{cases}$$

From this recurrence, we see that the depth $D(n)$ of the circuit performing a prefix computation on a sequence of n elements is given by

$$D_n = \begin{cases} 0 & n = 1 \\ D_{n-1} + 1 & i > 1 \end{cases} .$$

Here we are counting the number of $\oplus$ operations on the longest path through the circuit. The solution to this recurrence is $D(n) = n$ – another linear-depth circuit.

The following algorithm, however, can be implemented as a circuit with much smaller depth.

1. Compute the sums of pairs of consecutive elements in a, i.e., compute $g_i = a_{2i} \oplus a_{2i+1}$, for $0 \leq i < n/2$ (let's assume that n is even).
2. Recursively perform a prefix computation on the sequence $g_0, \ldots, g_{(n/2)-1}$. Let $h_0, \ldots, h_{(n/2)-1}$ denote the output sequence produced by this computation.
3. Compute the final output values $b_0, \ldots, b_{n-1}$ as follows:

$$b_i = \begin{cases} h_{(i/2)-1} \oplus a_i & i \text{ even} \\ h_{(i-1)/2} & i \text{ odd} \end{cases}$$

Task 3A

Based on this algorithm, sketch a circuit for performing the prefix computation. Write a recurrence describing the depth of your circuit, and solve the recurrence. In writing the recurrence, you may assume that n is a power of 2.

Now suppose that you are required to design a circuit in the form of an n-leaf complete binary tree, where initially a_i is stored at the ith leaf. Show that by passing data first all the way up the tree, then across to siblings, and then all the way down the tree, it is possible to perform the prefix computation in $(2 \log n) + 2$ steps.

3.3. Segmented prefix computations

Before showing how to cast addition as a prefix computation, we need to make a small generalization. In a *segmented prefix computation*, each element in the input sequence a may be preceded by

a "/", indicating that a new prefix computation is to start at that element. If, for example, the input sequence is

$$a = a_0, a_1, /a_2, a_3, /a_4,$$

then the output values are

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= a_0 \oplus a_1 \\ b_2 &= a_2 \\ b_3 &= a_2 \oplus a_3 \\ b_4 &= a_4. \end{aligned}$$

Thus, in this example there are three separate prefix computations, one on the sequence a_0, a_1, one on a_2, a_3, and one on a_4. We call each of these computations a segment.

Once we know the value of the carry bit c_i into each position i, we can quickly compute the sum $s_i = x_i \hat{} y_i \hat{} c_i$. We'll use segmented prefix computations to compute the carries. The key to understanding this is to think about how carries propagate from one bit position to the next. There are three cases:

1. If $x_i = 0$ and $y_i = 0$, then independent of the value of c_i, there will be no carry into bit position $i + 1$, i.e., $c_{i+1} = 0$. In this case we say that the carry is *stopped* at position i.
2. If $x_i = 1$ and $y_i = 1$, then independent of the value of c_i, there will be carry into bit position $i + 1$, i.e., $c_{i+1} = 1$. In this case we say that a carry is *generated* at position i.
3. Otherwise, if exactly one of x_i and y_i is 1, then $c_{i+1} = c_i$. In this case, the carry is *propagated* from position i to position $i + 1$.

Task 3B

Show how to compute the carry out of each position using a segmented prefix computation. You will have to define the elements of the input sequence, including the positions of the /'s, and also the binary associative operator $\oplus$.

Task3C

Finally, show that the segmented prefix computation is actually just another prefix computation (and hence can be solved by the circuit that you have already designed). Do this by defining, in terms of $\oplus$, a new binary associative operator $\otimes$ that operates on elements that may or may not be preceded by a /. A table showing the result of applying the operator $\otimes$ has been partially filled in for you below. Your task is to fill in the missing entries.

$\otimes$	d	$/d$
c	$c \oplus d$	
$/c$		

Now prove that your $\otimes$ operator is associative.