CS 213, Fall 2002
Lab Assignment L4: Code Optimization
Assigned: October 10
Due: October 24, 11:59PM

Randy Bryant (`Randy.Bryant@cs.cmu.edu`) is the lead person for this assignment.

1 Introduction

This assignment requires you to tune the performance of some simple C functions. There are two parts to the assignment. The first involves code where performance is constrained by arithmetic operations. In the second part, performance is constrained by memory operations.

Logistics

Everyone should work individually on this assignment. The only thing handed in will be your versions of the two files, which you will submit via our server.

You can retrieve the necessary files for this assignment by copying the following file into a protected directory:

```
/afs/cs.cmu.edu/academic/class/15213-f02/L4/L4.tar
```

Extract the code files with the command:

```
unix>  tar xf L4.tar
```

This will install a number of files in the directory. The only two that you will edit are `poly.c` and `row-col.c`.

Part I: Polynomial evaluation

A *polynomial of degree d* consists of $d+1$ coefficients $a_0, a_1, \ldots, a_d$. Given some value x, we evaluate the polynomial by computing:

$$P(x) \quad = \quad a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d \tag{1}$$

For integer coefficients and values of x, this can be performed by the following code:

```
int poly_eval(int *a, int degree, int x)
{
    int result = 0;
    int i;
    int xpwr = 1; /* Successive powers of x */
    for (i = 0; i <= degree; i++) {
        result += a[i]*xpwr;
        xpwr   *= x;
    }
    return result;
}
```

In this code, we achieve linear complexity by computing successive powers of x as we evaluate the coefficients in order. We would then expect the total number of cycles to evaluate a polynomial of degree d to be of the form

$$C(d) \quad = \quad \alpha d + \beta$$

We will measure the performance of the evaluation function in two ways:

CPE: The number of clock cycles per element, that is, the term α in the equation shown above. This measure is useful for analyzing how the performance grows for high degree polynomials.

C(10): The total number of clock cycles to evaluate a degree-10 polynomial. This measure is useful for analyzing the performance of the function on a polynomial of more typical degree.

Our original implementation has a CPE of 4.59, and a C(10) of 171.

Your job is to create versions of the functions that minimize the two measures. You will find that the two measures call for somewhat different optimizations.

One technique you might find useful is known as *Horner's Rule*. By exploiting the associative, commutative, and distributive laws, we can write the polynomial of Equation 1 as:

$$P(x) \quad = \quad a_0 + (a_1 + (a_2 + (\cdots + (a_{d-1} + a_d x)x \cdots)x)x)x \tag{2}$$

This reduces the number of multiplications to around 1/2 of what is required in the reference code shown above. You will find that a direct implementation of this rule actually slows down the function, but you can speed things up with the usual sorts of optimizations, such as loop unrolling.
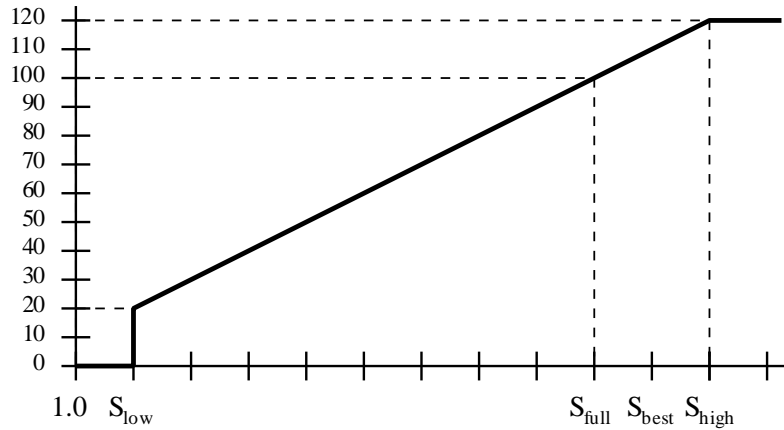
2

Figure 1: Computation of Performance Score. Your score depends on how much speedup your implementation gives over the reference version, compared to that of our best implementation

You will find it best to try a number of different implementations, with different degrees of unrolling, different associations of operations, etc. Our testing code provides a convenient way to test (for correctness) and measure (for performance) your many variants. That is, you can implement any number of functions with the calling convention shown in the code for `poly_eval` within the file `poly.c`. At the end of this file you will find a table, in the form of an initialized array, where you can put entries for different implementations of the function. You will find this is a useful way to keep a record of the different solutions you create. You should arrange your entries in the table so that the function with lowest CPE comes first, and the function with lowest C(10) comes second. The initial version of the table lists function `poly_eval` for these two entries.

To test and measure your implementations, use the command "`make poly_test`" to compile the test program and then run this program with the command "`./poly_test`". It will test each function in your table for correctness and then compute its CPE and C(10) values.

To get an official reading of performance, or to submit your solution, you should use our benchmark server. You can access this server via the class web page by following the links for lab L4. You can submit as many versions of your `poly.c` file as you like. The last one submitted will be considered to be your final solution.

Your grade for this part will be based on two measures: performance and code quality. Performance will be measured by how much speedup your solution gives over the original solution. That is, if your program requires C_{meas} cycles (either CPE or C(10)), then your speedup is $S_{meas} = C_{ref}/C_{meas}$, where C_{ref} is the number of cycles required by the reference implementation. Similarly, we have made our best attempt at optimizing the function, giving a cycle time of C_{best} and a speedup of $S_{best} = C_{ref}/C_{best}$. As shown in Figure 1, you will be given a score on a 100 point scale according to how well your speedup does relative to S_{best}. To get a nonzero score, you must achieve at least 10% of the speedup that we did (S_{low}). To get a score of 100, you must achieve 90% of the speedup we did (S_{full}). If you can match us, or even outperform us, you will get extra credit up to a maximum score of 120 (up to S_{high}). For your convenience, the scores for the two functions are printed when you run `poly_test`.

3

	Polynomial Evaluation		Matrix Summation	
	CPE	C(10)	Column CME	Row/Col. CME
C_{ref}	4.59	171.00	41.00	46.00
C_{best}	2.12	126.00	6.40	11.60
S_{best}	2.17	1.36	6.41	3.97
C_{low}	4.11	165.10	26.61	35.48
C_{full}	2.24	129.41	6.99	12.54
C_{high}	2.01	122.77	5.90	10.79

Figure 2: Performance Standards. C_{ref} indicates the performance (in clock cycles) of the reference implementation. C_{best} indicates the performance of the best implementation we have found. C_{low} indicates the performance required to get any performance credit (score = 20). C_{full} indicates the performance required to get full performance credit (score = 100). C_{high} indicates the performance required to get maximum performance credit (score = 120).

Figure 2 shows the actual cycle counts required to meet the thresholds low (score = 20), full (score = 100), and high (score = 120).

The code quality will be determined by looking at your code. Factors we will consider include the quality of the comments, choices of variable names, etc. Good code style is very important when you write code that does a lot of tricks for optimization. Include in your comments an explanation of the transformations you made and why they help improve the program performance. We will only look at the code for your two solution functions (best CPE and best C(10)).

Part II: Matrix Code

Consider the following definition of a square integer matrix:

```
#define N 512
/* N x N matrix */
typedef int matrix_t[N][N];

/* Vector of length N */
typedef int vector_t[N];
```

That is, data type `matrix_t` denotes a 512×512 matrix of integers, while data type `vector_t` denotes a vector of 512 integers. Given a matrix M, we wish to implement two different functions:

Column Sum: Compute a vector, where entry i is the sum of all elements in column i of M.

Row/Column Sum: Compute two vectors. In one, entry i is the sum of all elements in row i of M, while in the second, entry i is the sum of all elements in column i of M.

The following is our reference implementation for the column summation:

```
void c_sum(matrix_t M, vector_t rowsum, vector_t colsum)
{
    int i,j;
    for (j = 0; j < N; j++) {
        colsum[j] = 0;
        for (i = 0; i < N; i++)
            colsum[j] += M[i][j];
    }
}
```

Observe that the second argument to this function, rowsum, does not get used at all. We use this calling convention to enable the same testing code to be used with both types of summation functions.

The following is our reference implementation for the row/column summation:

```
void rc_sum(matrix_t M, vector_t rowsum, vector_t colsum)
{
    int i,j;
    for (i = 0; i < N; i++) {
        rowsum[i] = colsum[i] = 0;
        for (j = 0; j < N; j++) {
            rowsum[i] += M[i][j];
            colsum[i] += M[j][i];
        }
    }
}
```

We will measure the performance of the two functions in terms of the number of *clock cycles per matrix element* (CME). If the function requires C total cycles, then the performance is computed as $CME = C/512^2$. We use this form of measurement rather than CPE, since the underlying assumption of CPE is that performance scales in a simple, linear way with the number of elements. When code encounters cache effects, the performance varies nonlinearly.

On a Fish machine, our reference implementation of rc_sum requires around 46 cycles per element, while while c_sum requires around 41.

As in Part I, you can implement as many different column or row/column summation functions as you like in the file rowcol.c. You then enter the different function names and their descriptions in the table at the end of the file. Use the enumerated types COL and ROWCOL to indicate the type of summation function. Arrange the table so that your best column summation function comes first and your best row/column summation function comes second.

To test and measure your implementations, use the command "make rowcol_test" to compile the test program and then run this program with the command "./rowcol_test". It will test each function in your table for correctness and then compute its CME value.

As in Part I, your grade for each part will be determined by a combination of code quality and performance. Performance for a function will be measured by your speedup relative to that of our best implementation. The cycle thresholds are shown in Figure 2. Your performance scores are printed when you run row-col_test.

Additional Notes

- All of the code you write must be in standard, ANSI C. You may not use any embedded assembly or other features specific to GCC.

- You can develop and test your program on any IA32 machine. However, the actual performance measurements will be done on the Fish machines, so you will want to tune for these machines.

- You will find that the measured performance deviates from one run to the next. This is especially likely to happen with the memory-intensive matrix code when running on a heavily loaded system. Use our benchmark server, or try to work at a time when the machines are lightly loaded to get accurate measurements.

- Overall for this assignment, code quality counts 10% of the grade, and performance counts 90%.