Fall ’22 CIS 314 Assignment 6 – 100/100 points – Due Friday, 11/18, 11:59 PM

Please submit individual source files for coding exercises (see naming conventions below) and a single solution document for non-coding exercises (.txt or .pdf only). Your code and answers need to be documented to the point that the graders can understand your thought process. Full credit will not be awarded if sufficient work is not shown.

1. [30] Consider the following C code:

```
int f(int a, int b, int c, int d, int n) {
    int result = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
           result += a * b + i * c * d + j;
        }
    }
    return result;
}
```

Rewrite the above procedure *f* to remove all multiplications from inside the loops (you may still perform multiplication outside the loops – i.e., O(1) multiplicatons).

Here are some test runs:

```
f(1, 2, 3, 4, 5): 700
f(2, 3, 4, 5, 6): 2106
f(6, 5, 4, 3, 2): 146
f(5, 4, 3, 2, 1): 20
```

Also write an int main() function to test your procedure. Name your source file 6-1.c.

2. [40] Suppose we’ve got a procedure that computes the inner product of two arrays u and v. Consider the following C code:

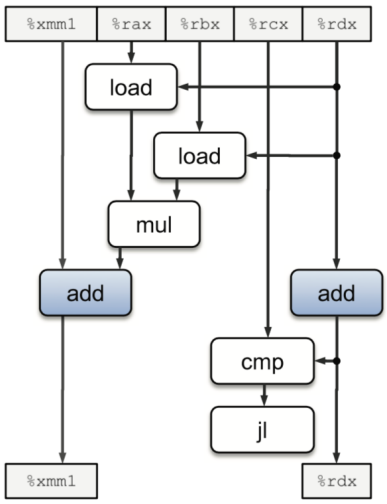
```
void inner(float *u, float *v, int length, float *dest) {
    int i;
    float sum = 0.0f;
    for (i = 0; i < length; ++i) {
        sum += u[i] * v[i];
    }
    *dest = sum;
}
```

The x86-64 assembly code for the inner loop is as follows:

```
# u in %rbx, v in %rax, length in %rcx, i in %rdx, sum in %xmm1
.L87:
    movss (%rbx, %rdx, 4), %xmm0 # Get u[i]
```

```
mulss (%rax, %rdx, 4), %xmm0 # Multiply by v[i]
adds %xmm0, %xmm1            # Add to sum
addq $1, %rdx                # Increment i
cmpq %rcx, %rdx              # Compare i to length
jl .L87                      # If <, keep looping
```

Also consider this data-flow diagram for the above procedure:



a. (10) Which operation(s) in the data-flow diagram above can NOT be parallelized? Hint: these will be the operation(s) that depend on the result of that operation from the previous loop iteration. Hint: see discussion around Figures 5.14 and 5.15. Write your answers in your solutions document.

b. (10) Given your answer from part a, what is the best-case CPE for the loop as currently written? Assume that *float* addition has a latency of 3 cycles, *float* multiplication has a latency of 5 cycles, and all integer operations have a latency of 1 cycle. Hint: the best-case CPE will be latency of the slowest of the operation(s) you identified in part a. Write your answers in your solutions document.

c. (10) Implement a procedure *inner2* that is functionally equivalent to *inner* but uses four-way loop unrolling with four parallel accumulators. Hint: see Figure 5.21. Also implement an int main() function to test your procedure. Name your source file 6-2c.c.

d. (10) Using your code from part c, collect data on the execution times of *inner* and *inner2* with varying array lengths. Summarize your findings and argue whether *inner* or *inner2* is more efficient than the other (or not). Create a graph using appropriate data points to support your argument. Include your summary and graph in your solutions document. Compile with -Og.

Hint: time how long it takes to call each function some constant number of times (e.g., 1000 – see examples from class); this will ensure that execution times aren't rounded down to 0.

3. [30] Consider the following C code:

```
float f(float *a, int n) {
      float prod = 1.0f;
      for (int i = 0; i < n; ++i) {
            if (a[i] != 0.0f) {
                  prod *= a[i];
            }
      }
      return prod;
}
```

The above code computes the product of nonzero elements in the array *a*.

```
float* createArray(int size) {
      float *a = (float *)malloc(size * sizeof(float));
      for (int i = 0; i < size; ++i) {
            // 50% chance that a[i] is 0.0f, random value on the range
            // [0.76, 1.26] otherwise.
            float r = rand()/(float)RAND_MAX;
            a[i] = r < 0.5f ? 0.0f : r + 0.26f;
      }
      return a;
}
```

The above helper function allocates an array of the specified size such that each element has a 50% chance of being zero.

a. (10) Write an int main() function that creates an array *a* using *createArray(10000)*, and then measures time taken to run *f(a)*. Print the result of *f(a)* to avoid dead-code removal. We'll cover basic instrumentation in lectures and labs. Don't forget the free the memory allocated for *a*!

b. (10) In your main() function, create a new array *b* containing the same elements as *a* but with every 0.0f replaced by 1.0f. Also write a new function *g* that is equivalent to *f* but without the zero (i.e., if a[i] != 0.0f) check. Calling *g(b)* should be functionally equivalent to calling *f(a)* because multiplication by 1.0 is a no-op. Measure the time taken to run *g(b)* - is it faster than calling *f(a)*? Explain why this might be the case in your code comments. Print the result of *g(b)* to avoid dead-code removal. Compile with -Og. Don't forget the free the memory allocated for *b*!

c. (10) In your main() function, create a new array *c* containing only the nonzero elements in *a* (i.e., *c* will almost certainly have fewer elements than *a*). Measure the time taken to run *g(c)* -

is it faster than calling *f(a)* and/or *g(b)*? Explain why this might be the case in your code comments. Print the result of *g(c)* to avoid dead-code removal. Compile with -Og. Don't forget the free the memory allocated for *c*!

Hint: time how long it takes to call each function some constant number of times (e.g., 1000 – see examples from class); this will ensure that execution times aren't rounded down to 0.

Name your source file 6-3.c.

Zip the source files and solution document (if applicable), name the .zip file <Your Full Name>Assignment6.zip (e.g., EricWillsAssignment6.zip), and upload the .zip file to Canvas (see Assignments section for submission link).