CS 61C Spring 2015
Guerrilla Section 3: VM, Parallelism & Potpourri

Problem 1 (adapted from Sp07 Final):

You run the following code on a system with the following parameter:

- 4 GiB virtual address space with 2 KiB page size, 1 GiB physical address space
- 8-entry TLB using LRU replacement
- 32 KiB data cache with 8B blocks and 4-way associative using LRU replacement

Assume that `char A[]` is both block-aligned and page-aligned, and that the code takes 1 page.

```
#define ARRAY_SIZE 33554432  // equal to 2^25
main() {
    int sum = 0, prod = 0;
    char *A = (char *) malloc(ARRAY_SIZE * sizeof(char));
    for (int i = 0; i < ARRAY_SIZE / STRETCH; i++) {
        for (j = 0; j < STRETCH; j++)      sum += A[i * STRETCH + j];
        for (j = STRETCH – 1; j >= 0; j--) prod *= A[i * STRETCH + j];
    }
}
```

a. What is the number of VPN bits? PPN bits? VPN bits: ____21____ PPN bits: ____19____

b. As we double STRETCH from 1 to 2 to 4 (…etc), we notice the number of cache misses doesn't change! What's the largest value of STRETCH before cache misses changes?

32 KiB. The idea is that we want all of the data accessed by the first inner for loop to be present when we access the second inner for loop. Thus, the largest value of STRETCH is the cache size. The hit rate will drop due to capacity misses afterwards.

c. As we double STRETCH from 1 to 2 to 4 (…etc), we notice the number of TLB misses doesn't change! What is the largest value of STRETCH before TLB misses changes?

8 KiB, since 1 page is needed for code, we have at max 7 pages (=14 KiB) for data. If the total number of pages used exceeds the number of TLB entries, we'll get capacity misses, and since STRETCH is a power of two, we can use 4 pages for data = 8 KiB.

d. For any value of STRETCH, what is the fewest number of page faults we could ever generate? Round to the nearest power of two.

16 Ki faults, since at best we would have only 1 page fault per unique page. There is 1 page of code and $2^{25}/2^{11} = 2^{14}$ pages, so after rounding we would get 2^{14} = 16 Ki faults.

e. Now suppose instead of 2 KiB pages, we had 256 B pages. All other specified parameters remain the same. If our code is 64 instructions long, how would performance change if we loop unrolled by a factor of 8? Explain.

64 instructions x 4B/instruction = 256 B, so our code fits exactly in one page. If we loop unrolled by 8x, our code would take (close to) 8 pages. Since there are only 8 entries in our TLB, we would have many capacity misses, and so performance would decrease.

Problem 2 (adapted from Sp14 Final):

In this problem, we will be parallelizing ways to compute the outer product. The outer product of two vectors is defined below. In this problem, our input vectors (x and y) will both be of length n.

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^{\mathrm{T}} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \\ u_4 v_1 & u_4 v_2 & u_4 v_3 \end{bmatrix}.$$

a. We want to parallelize the following code with OpenMP, but it is currently done incorrectly.

```
void outer_product(float* dst, float *x float *y, size_t n) {
    #pragma omp parallel
    for (size_t i = 0; i < n; i += 1) {
        for (size_t j = 0; j < n; j += 1) {
            #pragma omp critical
            dst[i * n + j] = x[i] * y[j];
        }
    }
}
```

You may only add or remove `#pragma omp` statements. What changes do we need to make the code run both quickly and correctly?

Change the #pragma omp parallel to #pragma omp parallel for
Remove the #pragma omp critical

b. Now use SSE instrinsics to optimize `outer_product()`. Assume `n` is a multiple of 4. You may find the following useful:

- `_mm_loadu_ps(__m128 *src)` loads the next four floats of `src` into the vector
- `_mm_load1_ps(float *f)` loads float `f` into each slot of the vector
- `_mm_storeu_ps(__m128 *dst, __m128 val)` stores `val` at memory location `dst`
- `_mm_mul_ps(__m128 a, __m128 b)` multiplies two vectors

```
void outer_product(float* dst, float *x, float *y, size_t n) {
    for (size_t i = 0; i < n; i+= 1) {
        for (size_t j = 0; j < n; j +=4) {
            __m128 a = _mm_load1_ps(&x[i]);
            __m128 b = _mm_loadu_ps(&y[j]);
            __m128 products = _mm_mul_ps(a, b);
            __mm_storeu_ps(&dst[i * n + j], products);
        }
    }
}
```

(continued on next page)

c. Finally, we will use Spark to generate outer products. Suppose that each input (x or y) is partitioned into equal-size chunks (there are `NUM` chunks per input), and each chunk is used as values for `map_func()`. The key for `map_func()` is a tuple `(is_x, i)` where:

- `is_x` is `True` if the associated value is from `x` and `False` if the value is from `y`
- `i` indicates that the associated value is the i-th chunk of the vector (x or y)

For example, `map_func((True, 2), [3, 4, 5])` means that `[3, 4, 5]` is the 2nd chunk of `x`.

You may assume the presence of an `outer_product(x, y)` function which returns the outer product of `x` and `y`. You may assume that data from `x` will always arrive before data from `y` during reduction.

```
def map_func((is_x, i), val):
    # YOUR CODE HERE
    output = []
    for j in range(len(NUM)):
        t = ((i, j), val) if is_x else ((j, i), val)
        output.append(t)
    return output


def reduce_func(v1, v2):
    return __outer_product(v1, v2)__

result = sc.parallelize(inputs)
         .____flatMap_________(map_func)
         .____reduceByKey_____(reduce_func)
         .reduce(join_outer_products)   # This function joins outer products
                                        # into a single outer product matrix
```

Problem 3: Potpourri

a. What's the correct data word given the following Hamming code: 0101000? _____0010_____

b. Write a MIPS function `IsNotInfinity` to return zero if and only if the input `$a0` is +/- infinity:

```
IsNotInfinity:    __sll___ $a0 $a0 1     # make -inf and inf look the same
                  __lui $t0, 0xFF00____
                  __xor $v0, $a0, $t0__
                  jr $ra
```

c. Suppose we modify the IEEE single-precision floating point format by adding one more exponent bit while removing one signifcand bit. Can we represent more or fewer real numbers? Why?
More. The non-real numbers floats represent are infinity and NaN. There are still only two infinities, but because there are fewer significand bits, there will be fewer NaNs. Thus we can represent more real numbers.

d. True/False:
__T__ For each FSM, there is an equivalent truth tables and Boolean algebra expression.
__F__ ECC provides protection from disk failures.
__F__ All RAID configurations improve reliability.
__F__ The MOESI cache coherency protocol helps prevent data races.
__F__ The MOESI cache coherency protocol helps prevent false sharing.
__T__ Polling is inefficient for disk transfers.