CS 61C Summer 2016 Discussion 10 – SIMD and OpenMP

SIMD PRACTICE

A.SIMDize the following code:

```
void count (int n, float *c) {
        for (int i= 0 ; i<n; i++)
                c[i] = i;
}
```

Enter your solution by filling in the spaces provided. Assume n is a multiple of 4. (__mm__set1__ps(x) returns a __m128 with all four elements set to x.)

```
Void countfast( int n, float *c) {
        float m[4] = { ___0___, ____1___, ___2____, ___3___};
        __m128 iterate = __mm_loadu_ps( m );
        for (int i= 0; i< ____n/4_____; i++)  {
                __mm__storeu__ps ( _______c+i * 4_____________, iterate);
                iterate  = __mm__add__ps( iterate, __mm__set1__ps(____4___));
                }
        }
```

b. Horner's rule is an efficient way to find the value of polynomial $p(x) = c_0x^{n-1} + c_1x^{n-2}+\ldots +c_{n-2}x + c_{n-1}$:

```
float poly( int n, float *c, float x) {
        float p = 0;
        for (int i = 0; i < n; i++ )
                p = p *x + c[i];
        return p;
}
```

Complete the following SIMD solution by filling in the blanks. Assume n is a multiple of 4.

```
Float fastpoly( int n, float *c, float x) {
        __m128 p = __mm__setzero__ps( );
        for  (int i= 0 ; i< n; i+=4) {
                p = __mm__mul__ps ( p, __mm__set1__ps (_____x*x*x*x_____));
                p = __mm__add__ps ( p, __mm__loadu__ps( _______c+i_________));
        }
        float m[4] = { ____x*x*x_______, _____x*x_____, _____x_____, _____1_____);
        p = __mm__mul__ps( p, __mm__loadu__ps(m));
        __mm__storeu__ps(m, p);
        return _____m[0] + m[1] + m[2]+ m[3]__ _____;
}
```

Thread Level Parallelism

```
#pragma omp parallelism
{
        /* code here */
}

#pragma omp parallel for
for (int i = 0; i < n; i++) {
        /* code here */
}
```

*Each thread runs a copy of code within the block
*Thread scheduling is non-deterministic

Same as:

```
#pragma omp parallel
{
        #pragma  omp for
        for (int i =0; i < n; i++) {...}
}
```

1. For the following snippets of code
below, circle one of the following to indicate what issue, if any, the code will experience. Then provide a short justification. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume arr is an int array with length n.

OPENMP Practice

a)

```
//Set element i to arr to i
#pragma omp parallel
for (int i = 0; i < n; i ++)
        arr[i] = I;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

Slower than serial – there is no for directive, so every thread executes this loop in its entirety. n threads running n loops at the same time will actually execute in the same time as 1 thread running 1 loop. Despite the possibility of false sharing, the values should all be correct at the end of the loop. Furthermore, the existence of parallel overhead due to the extra number of threads could slow down the execution time.

b)

```
//Set arr  to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i ++)
        arr[i] = arr[i-1] + arr[i-2];
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

Always incorrect (if n>4) – Loop has data dependencies, so the calculation of all threads but the first one will depend on data from the previous thread. Because we said "assume no thread will complete before another thread starts executing," then this code will always be wrong from reading incorrect values.

c)

```
//Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (int i = 0; i < n; i ++)
        arr[i] = 0;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

Faster than serial – the for directive actually automatically makes loop variables (such as the index) private, so this will work properly. The for directive splits up the iterations of the loop into continuous chunks for each thread, so no data dependencies or false sharing.

2. Consider the following code:

```
// Square element i of arr. n is a multiple of omp_get_num_threads()
```

```
#pragma omp parallel
{
        int threadCount = omp_get_num_threads();
        int myThread = omp_get_thread_num();
        for (int i = 0; i < n; i++) {
                if (i% threadCount  == my Thread)
                        arr[i] * = arr[i];
        }
}
```

What potential issue can arise from this code?

False sharing arises because different threads can modify elements located in the same memory block simultaneously. This is a problem because some threads may have incorrect values in their cache block when they modify the value arr[i], invalidating the cache block.

3.Consider the following function:

```
void transferFunds(struct account *from, struct account *to, long cents) {
        from  -> cents  -= cents;
        to-> cents += cents;
}
```

a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: If the problem isn't obvious, translate the function into MIPS first)

Each thread needs to read the "current" value, perform an add/sub, and store a value for from- >cents and to->cents. Two threads could read the same "current" value and the later store essentially overwrites the other transaction at either line.

b. How could you fix or avoid these races? Can you do this without hardware support?

Wrap transferFunds in a critical section, or divide up the accounts array and for loop in a way that you can have separate threads work on different accounts

OPEN MP Final Questions:

1.Which of the following choices correctly describes the given code? (Circle one). Explain your choice.

```
omp_set_num_threads(8);
#pragma omp parallel

{
     int thread_id = omp_get_thread_num();
     for(int count=0; count < ARR_SIZE/8*8; count+=8){
          arr[(thread_id%4)*2 + (thread_id/4) + count] = thread_id;
     }
```

}

A) Always correct, slower than serial
B) Always correct, speed depends on caching scheme
C) Always correct, faster than serial
D) Sometimes incorrect
E) Always incorrect

We have false sharing.

2.Suppose we have `int *A` that points to the head of an array of length `len`. Below are 3 different attempts to set each element to its index (i.e. `A[i]=i`) using OpenMP with `n>1` threads. Determine which statement (A)-(E) correctly describes the code execution and provide a one or two sentence justification.

a) #pragma omp parallel for

```
for (int x = 0; x < len; x ++) {
        *A = x;
        A++;
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always**Correct**,FasterthanSerial

The for loop work is split across threads, but there is a data race to increment the pointer A. However, if the threads happen to complete work in disjoint time intervals and in-order, we may get the correct result.

b) #pragma omp parallel {

```
for (int x = 0; x < len; x ++) {
        *(A+x) = x;
        }
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always**Correct**,FasterthanSerial

Our code computes the correct result but will be slower than the serial equivalent due to duplication of work.

c) #paragma omp parallel

```
{   for (int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {
```

```
        A[x] = x;
}}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always **Correct**, Faster than Serial

Here we will encounter false sharing. Although our result will be correct, the speedup will depend on how well our caching scheme handles processors working closely in memory and the order of thread execution.