CS 61C Fall 2017 Discussion 10

1. Data Level Parallelism

The idea central to data level parallelism is vectorized calculation. Thanks to the Intel intrinsics (Intel proprietary technology), we can use specialized instructions to deal with multiple data with one instruction.

__m128i _mm_load1_si128( )	returns 128-bit one vector
__m128i _mm_loadu_si128(__m128i *p)	returns 128-bit vector stored at pointer p
__m128i _mm_mul_ps(__m128 a, __m128 b)	returns vector (a0*b0, a1*b1, a2*b2, a3*b3)
void _mm_storeu_si128(__m128i *p, __m128i a)	stores 128-bit vector a at pointer p

1. Implement the following function, which returns the sum of two arrays:

```
static int product_naive(int n, int *a) {
        int product = 1;
        for (int i = 0; i < n; i++) {
                product *= a[i];
        }
        return product;
}
static int product_vectorized(int n, int *a)  {
         int result[4];
        __m128i prod_v = _____________________;

        for (int i = 0; i < _____; i += ___) {   // Vectorised loop
                prod_v =_____________________________________________;
        }
        __mm_storeu_si128(_____________________, _____________________);

        for (int i = _______; i < ____; i++) {  // Handle tail case
                result[0] *= _________;
        }
        return ____________________________________________;
}
```

2. Thread Level Parallelism

As powerful as data level parallelization is, it provides rather inflexible functionalities. The use of thread is much more powerful and versatile in many areas of programming. And OpenMP provides the hassle-free, plug-and-play directives to use of threads. Some examples of OpenMP directives:

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

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.

a)

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

Sometimes incorrect Always incorrect

Slower than serial Faster than serial

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

c)

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

Sometimes incorrect Always incorrect

Slower than serial Faster than serial

2.Consider the following code:

```
// Decrements 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 == myThread)
                        arr[i] *= arr[i];
        }
}
```

What potential issue can arise from this code?

3. Data race and Atomic operations.

The benefits of multi-threading programming come only after you understand concurrency. Here are two most common concurrency issues:

- **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. The can often be solved by bypassing cache and writing directly to memory, i.e. using `volatile` keyword in many languages.
- The famous **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-thread programming, the **interleaving** of R,M,W stages often produces a lot of issues.

To solve problem with Read-modify-write, we have to rely on the idea of **undisrupted execution**.

In RISC-V, we have two categories of atomic instructions:

- Load-reserve, store-conditional (undisrupted execution across multiple instructions)
- Amo.swap (single, undisrupted memory operation) and other amo operations.

Both can be used to achieve atomic primitives, here are two examples.

Test-and-set

```
Start: addi          t0 x0 1 #locked state is 1
       amoswap.w.aq t1 t0 (a0)
       bne           t1 x0 start #if the lock is not
                                  free, retry

       … #critical section

       amoswap.w.rl  x0 x0 a0#release lock
```

Compare-and-swap

```
#expect old value in a1, desired new value in a2
Start: lr      a3 (a0)
       bne     a3 a1 fail #CAS fail
       sc      a3 a2 (a0)
       bnez    a3 start #retry if store failed

        … #critical section

        amoswap.w.rl  x0 x0 a0
fail: #failed CAS
```

Instruction definitions:

- **L**oad-**r**eserve: Loads the four bytes from memory at address x[rs1], writes them to x[rd], sign-extending the result, and registers a reservation on that memory word.
- **S**tore-**c**onditional: Stores the four bytes in register x[rs2] to memory at address x[rs1], provided there exists a load reservation on that memory address. Writes 0 to x[rd] if the store succeeded, or a nonzero error code otherwise.
- **Amoswap**: Atomically, let t be the value of the memory word at address x[rs1], then set that memory word to x[rs2]. Set x[rd] to the sign extension of t.

Question: why do we need special instructions for these operations? Why can't we use normal load and store for lr and sc? Why can't we expand amoswap to a normal load and store?