Optimizing Program Performance

Annalise Tarhan

September 14, 2020

5 Homework Problems

5.13 Consider the function inner4, which computes the inner product of two vectors, and the corresponding assembly code.

5.13.1 Diagram how this instruction sequence would be decoded into operations and show how the data dependencies between them would create a critical path.

There are six operations in function inner4: load, mul, add, add, cmp, and jne. There are two chains of data dependencies between loop iterations, corresponding to the values stored in %rcx and %xmm0. The value in %rcx is used for both the multiplication and one of the addition operations, but it is only updated by the addition. The value in %xmm0 is used and updated exactly once, with the other addition operation. Although the multiplication is the most expensive operation, it is not on either of these paths. Both paths are critical, but the one involving %xmm0 is updated by floating point addition instead of integer addition, so it will be slower.

5.13.2 For data type double, what lower bound on the CPE is determined by the critical path?

The lower bound for the critical path involving %xmm0 is 3.00 clock cycles, the latency bound for floating-point addition.

5.13.3 Assuming similar instruction sequences for the integer code, what lower bound on the CPE is determined by the critical path for integer data?

It would be the latency lower bound for integer addition, which is 1.00 clock cycles.

5.13.4 Explain how the floating-point versions can have CPEs of 3.00, even though the multiplication operation requires 5 clock cycles.

This is hard for me to wrap my head around, since the multiplication in each iteration cannot begin until the value in %rcx is set by the previous iteration, but I suppose this is the magic of pipelining. As long as it isn't on the critical path, the compiler can make it work.

5.14 Write a version of the inner product procedure that uses 6x1 loop unrolling.

```
{
        ...
        for (i = 0; i < length-5; i += 6) {
                sum = sum + udata[i] * vdata[i] +
                        udata[i+1] * vdata[i+1] +
                        udata[i+2] * vdata[i+2] +
                        udata[i+3] * vdata[i+3] +
                        udata[i+4] * vdata[i+4] +
                        udata[i+5] * vdata[i+5];
        }
        for (; i < length; i++) {
                sum = sum + udata[i] * vdata[i];
        }
        ...
}
```

5.14.1 Explain why any (scalar) version of an inner product procedure running on our processor cannot achieve a CPE less than 1.00.

The processor only has one integer multiplier, so even if the multiplication operations could theoretically be computed in parallel, the processor can only handle one at a time. This is reflected in the fact that the throughput bound for integer multiplication is 1.00.

5.14.2 Explain why the performance for floating-point data did not improve with loop unrolling.

All of the addition operations are still happening sequentially, so they are limited by the 3.00 latency bound for floating point addition.

5.15 Write a version of the inner product procedure that uses 6x6 loop unrolling. What factor limits the performance to a CPE of 1.00?

```
{
        ...
        for (i = 0; i < length-5; i += 6) {
                acc0 = acc0 + udata[i] * vdata[i];
                acc1 = acc1 + udata[i+1] * vdata[i+1];
                acc2 = acc2 + udata[i+2] * vdata[i+2];
                acc3 = acc3 + udata[i+3] * vdata[i+3];
                acc4 = acc4 + udata[i+4] * vdata[i+4];
                acc5 = acc5 + udata[i+5] * vdata[i+5];
        }
        acc = acc0 + acc1 + acc2 + acc3 + acc4 + acc5;
        for (; i < length; i++) {
                acc = acc + udata[i] * vdata[i];
        }
        ...
}
```

The limiting factors for integer and floating-point data are the throughput bounds for integer multiplication and floating-point addition, respectively.

5.16 Implement a more efficient version of memset by using a word of data type unsigned long to pack eight copies of c, and then step through the region using word-level writes.

I'm confused by this question. It says clearly that the code should work for any size of unsigned long, which is why we use K instead of the standard eight. But, it is also very clear that we should pack eight copies of c into an unsigned long and step through doing word-level writes, writing eight bytes every clock cycle. So which is it?

```
void *efficient_memset(void *s, int c, size_t n) {
        /* c_word is the lowest byte of c repeated 8x */
        unsigned char c_byte = c & 0xff;
        unsigned long c_word = c_byte;
        for (int i = 0; i < 7; i++) {
                c_word <<= 1;
                c_word |= c_byte;
        }

        unsigned char *schar = s;
        size_t cnt = 0;
        /* First, set the bytes before the first
        address that is a multiple of eight */
        while ((cnt < n) && (*schar % 8 != 0)) {
        *schar++ = c_byte;
        cnt++;
}
/* Then, write eight bytes at a time, but only if
    at least eight bytes need to be written */
while (n >= 8 && cnt < n-8) {
        *schar = c_word;
        *schar += 8;
        cnt += 8;
}
/* Finally, set the remaining bytes */
while (cnt < n) {
        *schar++ = c_byte;
        cnt++;
}
return s;
}
```

```
void *efficient_memset(void *s, int c, size_t n) {
        size_t K = sizeof(unsigned long);
        /* c_word is lowest byte of c repeated K times */
        unsigned char c_byte = c & 0xff;
        unsigned long c_word = c_byte;
        for (int i = 0; i < K-1; i++) {
                c_word <<= 1;
                c_word |= c_byte;
        }

        unsigned char *schar = s;
        size_t cnt = 0;
        /* First, set the bytes before the first
        address that is a multiple of K */
        while ((cnt < n) && (*schar % K != 0)) {
        *schar++ = c_byte;
        cnt++;
}
/* Then, write K bytes at a time, but only if
    at least K bytes need to be written */
while (n >= K && cnt < n-K) {
        *schar = c_word;
        *schar += K;
        cnt += K;
}
/* Finally, set the remaining bytes */
while (cnt < n) {
        *schar++ = c_byte;
        cnt++;
}
return s;
}
```

5.17 Write faster versions of polynomial evaluation using the optimization techniques we've explored.

```
double fast_poly(double a[], double x, long degree) {
        long i;
        double acc1 = a[0];
        double acc2 = 0;
        double acc3 = 0;
        double xpwr = x;
        double x3pwr = x*x*x;
        double x5pwr = x3pwr*x*x;
        double x6pwr = x5pwr*x;
        for (i = 1; i < degree-5; i += 6) {
                acc1 += ((a[i] * xpwr)
                        + (a[i+1] * (x * xpwr)));
                acc2 += ((a[i+2] * x3pwr)
                        + (a[i+3] * (x * x3pwr)));
                acc3 += ((a[i+4] * x5pwr)
                        + (a[i+5] * (x * x5pwr)));
                xpwr *= x6pwr;
                x3pwr *= x6pwr;
                x5pwr *= x6pwr;
        }
        double result = acc1 + acc2 + acc3;
        for (; i <= degree; i++) {
                result += a[i] * xpwr;
                xpwr *= x;
        }
        return result;
}
```

5.18 Using a combination of loop unrolling and reassociation, write code for a faster prefix sum.

```
void fast_psum(float a[], float p[], long n) {
        long i;
        float cur, p1, p2, p3;
        float prev = a[0];
        p[0] = prev;
        for (i = 1; i < n-3; i += 4) {
                cur = a[i] + prev;
                p1 = a[i+1];
                p2 = a[i+2];
                p3 = a[i+3];
                p[i] = cur;
                p[i+1] = cur + p1;
                p[i+2] = cur + (p1 + p2);
                prev = cur + (p1 + p2 + p3);
                p[i+3] = prev;
        }
        for (; i < n; i++) {
                p[i] = p[i-1] + a[i];
        }
}
```