15-418/618

RECITATION 1, JANUARY 26, 2017, SPRING 2018

RECITATION MATERIALS

- /afs/cs.cmu.edu/academic/class/15418-s18/public/recw2

/PROC/CPUINFO

- model name
- cpu Mhz – speed now (Speed step)
- cache size – outermost (L3)
- siblings – number of hyperthreads
- processor – id of a hyperthread
- cpu cores
- core id
- physical id – socket
- flags – note avx, avx2, sse, etc.

WHAT ELSE (QUICK AND DIRTY) TO LOOK UP?

- Memory bandwidth – 76.8 GB/s
- Power consumption – 140W
- Intel codename: Broadwell (Shrink of Haswell)
- Functional units
  - Latency, issue time, capcacity

DEVIL IN THE DETAILS

- http://www.agner.org/optimize/microarchitecture.pdf
- CS:APP Textbook chapter 5

FUNCTIONAL UNITS

0. int arithmetic, fp multiply, int and fp division, branches
1. int arithmetic, fp addition, int multiplication, fp multiplication
2. load, address computation
3. load, address computation
4. store
5. integer arithmetic
6. integer arithmetic, branches
7. store address computation

Observations:

- 4x independent int operations (add, bitwise ops, etc)

- Takes 2 functional units to store (compute address and store)

LATENCY, ISSUE TIME, CAPACITY

	Integer			Floating-Point		
Operation	Latency	Issue	Capcacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3-30	3-30	1	3-15	3-15	1

Latency: Clock cycles required to perform the operation
Issue time: Minimum number of lock cycles between issuing independent operations
Capacity: How many can be issued simultaneously

Pipelining enables the short issue times
(Multiple instances in various states of execution concurrently)

TAYLOR SERIES APPROXIMATION OF SIN(X)

```
// Original version of sin function
void sinx_reference(int N, int terms, float * x, float *result)
{
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;
        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

What’s the inner-work loop? How many times is it executed? What are the pain point(s)?

TAYLOR SERIES APPROXIMATION OF SIN(X)

```
// Original version of sin function
void sinx_reference(int N, int terms, float * x, float *result)
{
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;
        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

What's the inner-work loop?
How many times is it executed? N*terms
What are the pain point(s)? Multiplication is expensive. So is division!

BENCHMARKING

- Time = 1062.54ms. -- Total execution time
- N = 10000 – total number of elements
- T = 15 – Number of terms/element
- r = 1000 -- repetitions
- t = 1 – number of threads
- 7.084 ns/element – time per element

SIMPLE IMPROVEMENTS

```
void sinx_better(int N, int terms, float * x, float *result)
{
    // Make some simple fixes that you think might help
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float numer = x2 * value;
        int denom = 6; // 3!
        int sign = -1;
        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x2;
            denom *= (2*j+2) * (2*j+3);
            sign = -sign;
        }
        result[i] = value;
    }
}
```

6.16 ns/element – time per element

LET'S FOCUS ON THIS

```
for (int i=0; i<N; i++) {
  …
  for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x2;
            denom *= (2*j+2) * (2*j+3);
            sign = -sign;
        }
        result[i] = value;
    }
  …
}
```

- Division is very costly
- Computation is independent of i

PRECOMPUTE RECIPROCAL FACTORIALS

```
void sinx_predenoms(int N, int terms,
float * x, float *result)
{
    float rdenom[MAXTERMS];
    int denom = 6;
    float sign = -1;
    for (int j = 1; j <= terms; j++) {
        rdenom[j] = sign/denom;
        denom *= (2*j+2) * (2*j+3);
        sign = -sign;
    }
```

```
for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float numer = x2 * value;
        for (int j=1; j<=terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

1.2ns/element, 3.6 cycles – nice!

INNER WORK LOOP -- ASM

```
.L38:
        vfmadd231ss     (%rax), %xmm0, %xmm1
        addq    $4, %rax
        vmulss  %xmm2, %xmm0, %xmm0
        cmpq    %r8, %rax
        jne     .L38
```

```
loop:
  value += numer * redenom[j];
  j++;
  numer *= x2;
  test = i != N;
  if (test) goto loop;
```

Vfmadd – fused multiply and add

3 cycle latency of multiplication is limiting

LOOP UNROLLING

- Reduce the number of “tests” and increase the amount of work done
- Avoid branch penalities from jumping
- Improve the potential for ILP by removing tests and jumps and leaving parallelism
- If taken to extremes…
  - Binary size can be a factor (Especially for IoT applications, etc)
  - Instruction caches can be swamped
  - More temporary variables versus registers (less of a current concern)

LOOP UNROLLING

```
.L49:
        vmovaps %xmm1, %xmm3
        vmulss  4(%rax), %xmm2, %xmm1
        addq    $8, %rax
        vfmadd231ss     -8(%rax), %xmm0, %xmm3
        cmpq    %r10, %rax
        vfmadd132ss     %xmm0, %xmm3, %xmm1
        vmulss  %xmm4, %xmm0, %xmm0
        jne     .L49
```

```
loop:
  t = numer * rdenom[j+1];
  j+=2;
  value += numer * rdenom[j-2];
  value += numer * t;
  test = j==terms-1;
  numer *= x4;
  if (test) goto loop;
```

Critical path has two fp ops ==> 6 clock cycles for 2 elements.
Should be the same as regular code, but one less multiply might be helping it run faster.

REASSOCIATION (ASSUME FP IS DISTRIBUTIVE AND ASSOCIATIVE)

```
.L64:
        vmovss  4(%rax), %xmm3
        addq    $8, %rax
        vfmadd213ss     -8(%rax), %xmm2,
%xmm3
        cmpq    %r10, %rax
        vfmadd231ss     %xmm0, %xmm3,
%xmm1
        vmulss  %xmm4, %xmm0, %xmm0
        jne     .L64
```

```
loop:
        v = rdenom[j+1];
        j+= 2;
        t = rdenom[j-2] + x2 * v;
        test = j != terms;
        value += numer * t;
        numer *= x4;
        if (test) goto loop;
```

Updating of value and numer both induce delay of 3 cycles, but compute 2 elements in that time. CPE=1.5

LIMITS OF UNROLLING
(CONSIDER AN UNROLLING FACTOR OF *K*)

- As keep unrolling more, would be limited by addition to update value. ~3.0/k.
- Limited by pair of FMA units once k exceeds 6. E.g., for k = 6
  - would have 7 operations, requiring 3.5 cycles to compute 6 elements = 3.5/6.
  - In general would have k+1 operations performed by two
- functional units to produce k values. CPE = (k+1)/12.
- We'd also be limited by the overhead of setting up the loop, things not dividing evenly into iterations, etc.

LIMITS OF UNROLLING

2x: 0.70 ns/element
3x: 0.50
4x: 0.76
5x: 0.48

16 terms:

2x: 0.676
3x: 0.610
4x: 0.498 (16 evenly divides 4)
5x: 0.575

ISPC VECTORIZATION OF UNROLLED CODE

```
  uniform float rdenom[MAXTERMS];
  uniform int denom = 6;
  uniform float sign = -1;
  …

foreach (i=0 ... N) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float x6 = x2 * x4;
        float numer = x2 * value;

        uniform int j;
```

```
for (j=1; j<=terms-2; j+=3) {
        value +=
          numer * (rdenom[j] +
                x2 * rdenom[j+1] +
                x4 * rdenom[j+2]);
        numer *= x6;
    }

    for (; j <= terms; j++) {
        value += numer * rdenom[j];
        numer *= x2;
    }
    result[i] = value;
}
```

ISPC PERFORMANCE

- The ISPC code ran in 0.63ns/element, versus 7.16 for the unvectorized code, a speedup of 9.83x.
- A speedup of more than 8x seems surprising
- But, *demon* and *sign* are uniform, so only computed once.

WHAT'S THE SCORE?

- Vectorizing was easy: 5.40x speedup
- Conventional optimization was painful: 15x improvement
- Total improvement: 82x