Recitation 1:
ILP, SIMD, and Thread Parallelism

15-418 Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2019
Goals for today

- Topic is parallelism models: ILP, SIMD, threading
- Solve some exam-style problems
- Walk through example code
- Most of all, ANSWER YOUR QUESTIONS!
Recall: Taylor expansion of $\sin(x)$

void sinx(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;
    }
}

- How fast is this code?
- Where should we focus optimization efforts?
- What is the bottleneck?
Recall: Taylor expansion of $\sin(x)$

```c
void sinx(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;
    }
}
```

- How fast is this code?
- On ghc machines:
  - $7.2$ ns / element $\approx 23$ cycles / element
- Not very good 😞
Recall: Taylor expansion of $\sin(x)$

void sinx(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;
    }
}

- Where should we focus optimization efforts?
- A: Where most of the time is spent
Recall: Taylor expansion of $\sin(x)$

```c
void sinx(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 is the bottleneck?
Dataflow for a single iteration

OK, but how does this perform on a real machine?
Superscalar OOO Processor

- What in microarchitecture should we worry about?
GHC Machine Microarchitecture

What in microarchitecture should we worry about?

- Fetch & Decode?
  
  NO. Any reasonable machine will have sufficient frontend throughput to keep execution busy + all branches in this code are easy to predict (not always the case!).

- Execution?
  
  YES. This is where dataflow + most structural hazards will limit our performance.

- Commit?
  
  NO. Again, any reasonable machine will have sufficient commit throughput to keep execution busy.
## Intel Broadwell (GHC machines) Execution Microarchitecture

<table>
<thead>
<tr>
<th></th>
<th><strong>Integer</strong></th>
<th></th>
<th><strong>Floating Point</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Pipelined?</td>
<td>Number</td>
<td>Latency</td>
</tr>
<tr>
<td>Add</td>
<td>1</td>
<td>✓</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Multiply</td>
<td>3</td>
<td>✓</td>
<td>1</td>
<td>5 3</td>
</tr>
<tr>
<td>Divide</td>
<td>3-30</td>
<td>×</td>
<td>1</td>
<td>3-15</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>✓</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
What is our **throughput bound**?

\[
\text{sign} \times \text{numer} \times \ldots \\
\text{denom} \times \ldots \\
\text{value} \times \ldots \\
\text{value}' \times \ldots \\
\text{denom}' \times \ldots \\
\text{sign}' \\
\text{numer}' \\
\text{LD} \\
\text{Op} \quad \# \text{ Code} \quad \mu\text{Arch} \quad \text{Thput bound}
\]

- Int Add
- Int Mul
- Int Div
- FP Add
- FP Div
- Load

**THPUT BOUND!**
What is our latency bound?

- Find the critical path in the dataflow graph

![Dataflow Diagram]

```
3 + (3/15) + 3
= 9 to 21
```

```
3 + 1 + 3 + 3 = 10
```

```
1 + 3 + 3 = 7
```
Takeaways

▪ Observe performance of 23 cycles / element

▪ Latency bound dominates throughput bound
  ➔ We are latency bound!

▪ Notes
  ▪ This analysis can often be “eyeballed” w/out full dataflow
  ▪ Actual execution is more complicated, but latency/thput bounds are good approximation
  ▪ (Also, avoid division!!!)
Speeding up \( \sin(x) \): Attempt #1

- What if we eliminate unnecessary work?

```c
void sinx_better(int N, int terms, float * x, 
                  float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = x[i]*x[i];
        float numer = x2*x[i];
        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;
    }
}
```

A: Small improvement.

6ns / element \( \approx \)
18 cycles / element

Why not better?
What is our **latency bound**?

- Find the critical path in the dataflow graph

![Diagram of a dataflow graph with nodes and operations labeled with values. The critical path is highlighted in red.](image)
Attempt #1 Takeaways

▪ First attempt didn’t change latency bound

▪ To get real speedup, we need to focus on the performance bottleneck

▪ Q: Why did we get any speedup at all?
  ▪ A: Actual dynamic scheduling is complicated; would need to simulate execution in more detail
Speeding up $\sin(x)$: Attempt #2

Let's focus on that pesky division...

```c
void sinx_predenom(int N, int terms, float * x, float *result) {
  float rdenom[MAXTERMS];
  int denom = 6;
  for (int j = 1; j <= terms; j++) {
    rdenom[j] = 1.0/denom;
    denom *= (2*j+2) * (2*j+3);
  }
  for (int i=0; i<N; i++) {
    float value = x[i];
    float x2 = value * value;
    float numer = x2 * value;
    int sign = -1;
    for (int j=1; j<=terms; j++) {
      value += sign * numer * rdenom[j];
      numer *= x2;
      sign = -sign;
    }
    result[i] = value;
  }
}
```

A: Big improvement!

2.4ns / element $\approx$ 7.7 cycles / element
What is our latency bound?

- Find the critical path in the dataflow graph

\[ j' \quad \text{value} \quad \text{rdenom}[j] \quad \text{sign} \quad \text{numer} \]

\[ j + 1 \quad \text{LD} \quad \times \quad \text{NEG} \quad \times \]

\[ \text{value}' \quad \text{sign}' \quad \text{numer}' \]

\[ j' + 1 \quad 3 + 3 = 6 \quad 1 \quad 3 \]

\[ \text{LATENCY BOUND!} \]
Attempt #2 Takeaways

- Here we go! Attacking the bottleneck got nearly 3×!

- …But performance is still near the latency bound, can we do better?
Speeding up \( \sin(x) \): Attempt #3

- Don’t need sign in inner-loop either

```c
void sinx_predenoms(int N, int terms, float * x, float *result) {
    float rdenom[MAXTERMS];
    int denom = 6;
    float sign = -1.0;
    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;
    }
}
```

\(1.1 \text{ns / element} \approx 3.5 \text{ cycles / element}\)
What is our latency bound?

- Find the critical path in the dataflow graph

\[ \text{numer} \times \text{denom}[j] \]

3 (LD will be executed speculatively, only depends on \( j \))
Attempt #3 Takeaways

- We’re down to the latency of a single, fast operation per iteration
- Observed performance is very close to this latency bound, so throughput isn’t limiting
- ➔ We’re done optimizing individual iterations

How to optimize multiple iterations?
- Eliminate dependence chains across iterations
  - A) Loop unrolling (ILP)
  - B) Explicit parallelism (SIMD, threading)
Speeding up $\sin(x)$: Loop unrolling

- Compute multiple elements per iteration

```c
void sinx_unrollx2(int N, int terms, float * x, float *result) {
    // same predom stuff as before...
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        for (int j=1; j<=terms; j+=2) {
            value += numer * rdenom[j];
            value += numer * x2 * rdenom[j+1];
            numer *= x4;
        }
        result[i] = value;
    }
}
```

Correct? Not yet...
Speeding up $\sin(x)$: Loop unrolling

- Compute multiple elements per iteration

```c
void sinx_unrollx2(int N, int terms, float * x, float *result) {
    // same predom stuff as before...
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        int j;
        for (j=1; j<=terms-1; j+=2) {
            value += numer * rdenom[j];
            value += numer * x2 * rdenom[j+1];
            numer *= x4;
        }
        for (; j<=terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

0.99 ns / element \(\approx\) 3.2 cycles / element

Didn’t change 😞
What is our latency bound?

- Find the critical path in the dataflow graph:

\[
\begin{align*}
  & j \quad \text{value} \quad \text{rdenom}[j] \quad \text{rdenom}[j+1] \quad x_2 \quad \text{numer} \\
  & 2 + j' = 2 + \frac{1}{2} = 0.5 \\
  & 3/2 = 1.5 \\
\end{align*}
\]

- LD will be executed speculatively, only depends on j.
Speeding up \( \sin(x) \): Loop unrolling #2

- What if floating point associated + distributed?

```c
void sinx_unrollx2(int N, int terms, float * x, float *result) {
    // same predom stuff as before...
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        int j;
        for (j=1; j<=terms-1; j++) {
            value += numer * (rdenom[j] + x2 * redom[j+1]);
            numer *= x4;
        }
        for (; j<=terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

0.69 ns / element \( \approx \) 2.2 cycles / element
What is our latency bound?

- Find the critical path in the dataflow graph

\[
\begin{align*}
\text{j} & \quad \text{value} \quad \text{rdenom}[j] \quad \text{rdenom}[j+1] \quad \times 2 \quad \text{numer} \quad \times 4 \\
& \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\text{j'} & \quad \text{value}' \quad \text{rdenom}[j+1] \quad \times 2 \quad \text{numer}' \\
& \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
1/2=0.5 & \quad 3/2=1.5 \quad (\text{LD will be executed speculatively, only depends on } j) \quad 3/2=1.5
\end{align*}
\]
Loop unrolling takeaways

- Need to break dependencies across iterations to get speedup
  - Unrolling by itself doesn’t help

- We are now seeing throughput effects
  - Latency bound = 1.5 vs. observed = 2.2

- Can unroll loop 3x, 4x to improve further, but...
  - …Diminishing returns (1.65 cycles / element at 4x)
Speeding up \( \sin(x) \): Going parallel (explicitly)

- Use ISPC to vectorize the code

```c
export void sinx_reference(uniform int N, uniform int terms, uniform float x[], uniform float result[]) {
    foreach (i=0 ... N) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform 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;
    }
}
```

1.0 ns / element \( \approx \) 3.2 cycles / element
Speeding up $\sin(x)$: Going parallel (explicitly) + optimize

```c
export void sinx_unrollx2a(uniform int N, uniform int terms,
    uniform float x[],
    uniform float result[]) {
    uniform float rdenom[MAXTERMS];
    uniform int denom = 6;
    uniform float sign = -1;
    for (uniform int j = 1; j <= terms; j++) {
        rdenom[j] = sign/denom;
        denom *= (2*j+2) * (2*j+3);
        sign = -sign;
    }
    foreach (i=0 ... N) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        uniform int j;
        for (j=1; j<=terms-1; j+=2) {
            value +=
                numer * (rdenom[j] +
                     x2 * rdenom[j+1]);
            numer *= x4;
        }
        for (; j <= terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
```

$0.14 \text{ ns} / \text{element} \approx 0.45 \text{ cycles} / \text{element}$
**SIMD takeaways**

- Well, that was easy! (Thanks ISPC)

- Cycles per element:

- Speedup

  Maximum speedup requires hand tuning + explicit parallelism!
What if? #1
Impact of structural hazards

Q: What would happen to $\sin(x)$ if we only had a single, unpipelined floating-point multiplier?

- A1: Performance will be much worse
- A2: We will hit throughput bound much earlier
- A3: Loop unrolling will help by reducing multiplies
What if? #2

Impact of structural hazards

Q: What would happen to $\sin(x)$ if LDs (cache hits) took 2 cycles instead of 1 cycle?

A: Nothing. This program is latency bound, and LDs are not on the critical path.
Loads do not limit \( \sin(x) \)

- Consider just the slice of the program that generates the subexpression: \( (\text{rdenom}[j] + x2 \times \text{rednom}[j+1]) \)

- What is this program’s latency + throughput bound?

  - **Latency bound:** 1 cycle / iteration!
    - Through \( j' \) computation, not the subexpression computation – there is no cross-iteration dependence in the subexpression!

  - **Throughput bound:** also 1 cycle / iteration
    - 1 add / 4 adders; 2 LDs / 2 LD units; 1 FP FMA / 1 FP unit
    - (This will change to 2 cycles if we add the value FMA)
Loads do not limit \( \sin(x) \):
Visualization

- Consider just the slice of the program that generates the subexpression:
  \((\text{rdenom}[j] + x2 \times \text{rednom}[j + 1])\)

- Subexpressions are off the critical path + we have enough throughput to produce next subexpression each cycle (excluding value FMA)
Loads do not limit \(\sin(x)\): Example execution

Note: Throughput limit is 2 cycles / iteration once we add \texttt{value} FMA, but this is dominated by the latency bound of 3 cycles / iteration (also from \texttt{value} FMA). Regardless, 2-cycle LDs are not the bottleneck.
What if? #3  
Vector vs. multicore

- Q: What would happen to \( \sin(x) \) if the vector width was doubled?
  - A1: If we’re using ISPC, we would expect roughly 2× performance (slightly less would be realized in practice).

- Q: Can we do this forever & expect same results?
  - A: No. Computing \( \text{rdenom} \) will limit gains (Amdahl’s Law).

- Q: For this \( \sin(x) \) program, would you prefer larger vector or more cores?
  - A: Either should give speedup, but this program maps easily to SIMD, and adding vector lanes is much cheaper (area + energy) than adding cores. (Remember GPU vs CPU pictures.)
What if? #4

Benefits(?) of SMT

- Q: How should we schedule threads on a dual-core processor with SMT, running these two apps, each of which have 2 threads?
  - The $\sin(x)$ function
  - A program that is copying large amounts of data with very little computation

- (Note: There are four “cores” and four threads)

- A: We want to schedule one $\sin(x)$ thread and one memcpy() thread on each core, since SMT is most beneficial when threads use different execution units
What if? #5
Limits of speculation

Q: What will limit the “performance” of this (silly) program on a superscalar OOO processor?

```c
int foo() {
    int i = 0;
    while (i < 100000) {
        // assume single-cycle rand instruction
        if (rand() % 2 == 0) {
            i++;
        } else {
            i--;
        }
    }
}
```

A: Unpredictable branch in if-else will cause frequent pipeline flushes
What if? #6
Benefits(?) of SMT

- Q: Would the previous program benefit from running on multiple SMT threads on a single core?

- A: Yes! Its performance is limited by the CPU frontend, which is replicated in SMT