How to Write Fast Numerical Code
Spring 2017

Lecture: Optimization for Instruction-Level Parallelism

Instructor: Markus Püschel
TA: Alen Stojanov, Geoerg Ofenbeck, Gagandeep Singh

Abstracted Microarchitecture: Example Core i7 Haswell (2013) and Sandybridge (2011)

Throughput ($\text{tp}$) is measured in doubles/cycle. For example: $4 (2)$
Latency (lat) is measured in cycles
1 double floating point (FP) = 8 bytes
fma = fused multiply add
Rectangles not to scale

Memory hierarchy:
- Registers
- L1 cache
- L2 cache
- L3 cache
- Main memory
- Hard disk

Mapping of execution units to ports

<table>
<thead>
<tr>
<th>Port 0</th>
<th>Port 1</th>
<th>Port 2</th>
<th>Port 3</th>
<th>Port 4</th>
<th>Port 5</th>
<th>Port 6</th>
<th>Port 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp fma</td>
<td>fp fma</td>
<td>load</td>
<td>load</td>
<td>store</td>
<td>SIMD log</td>
<td>Int ALU</td>
<td>st addr</td>
</tr>
<tr>
<td>fp mul</td>
<td>fp mul</td>
<td>st addr</td>
<td>st addr</td>
<td>shuffle</td>
<td>fp mov</td>
<td>Int ALU</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>fp add</td>
<td>SIMD log</td>
<td>SIMD log</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Int ALU</td>
<td>Int ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Every port can issue one instruction/cycle
Gap = 1/throughput

Intel calls gap the throughput!

Same units for scalar and vector flops

How To Make Code Faster?

- It depends!
- Memory bound: Reduce memory traffic
  - Reduce cache misses, register spills
  - Compress data
- Compute bound: Keep floating point units busy
  - Reduce cache misses, register spills
  - Instruction level parallelism (ILP)
  - Vectorization
- Next: Optimizing for ILP (an example)

Part of these slides are adapted from the course associated with this book
Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- **Benefit:** Superscalar processors can take advantage of *instruction level parallelism (ILP)* that many programs have.

- Most CPUs since about 1998 are superscalar
- Intel: since Pentium Pro
- Simple embedded processors are usually not superscalar

---

**ILP**

**Code**

\[
\begin{align*}
t_2 &= t_0 + t_1 \\
t_5 &= t_4 \times t_3 \\
t_6 &= t_2 + t_5 \\
\end{align*}
\]

**Dependencies**

\[
\begin{align*}
t_6 &= t_2 + t_5 \\
t_2 &= t_0 + t_1 \\
t_5 &= t_4 \times t_3 \\
\end{align*}
\]

can be executed in parallel and in any order
### Hard Bounds: Pentium 4 vs. Core 2

- **Pentium 4 (Nocona)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

- **Core 2**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>18/50</td>
<td>18/50</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>18/32</td>
<td>18/32</td>
</tr>
</tbody>
</table>

\[
\text{put on blackboard}
\]

\[
1/\text{throughput} = 2 \text{ cycles}
\]

- **Single/Double FP Multiply**

\[
\text{ops}
\]

\[
\text{gap} = 1/\text{throughput} = 2 \text{ cycles}
\]

\[
\text{cycles}
\]
Hard Bounds (cont’d)

- How many cycles at least if
  - Function requires $n$ float adds?
  - Function requires $n$ int mults?

Example Computation (on Pentium 4)

```c
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
    {
        t = t OP d[i];
        *dest = t;
    }
}
```


data_t: float or double or int

OP:  + or *
IDENT: 0 or 1
Runtime of Combine4 (Pentium 4)

- Use cycles/OP

```c
void combine4(vec_ptr v, data_t *dest) {
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

- Questions:
  - Explain red row
  - Explain gray row

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>10.0</td>
<td>7.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Cycles per element (or per OP)

- Sequential dependence = no ILP!

Hence: performance determined by latency of OP!
Loop Unrolling

```c
void unroll2(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2)
        x = (x OP d[i]) OP d[i+1];
    /* Finish any remaining elements */
    for (; i < length; i++)
        x = x OP d[i];
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer sum
- Others don’t improve. Why?
  - Still sequential dependency
    ```c
    x = (x OP d[i]) OP d[i+1];
    ```
Loop Unrolling with Separate Accumulators

```c
void unroll2_sa(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++)
        x0 = x0 OP d[i];
    *dest = x0 OP x1;
}
```

- Can this change the result of the computation?
- *Floating point: yes!*

Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-sa</td>
<td>1.50</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Almost exact 2x speedup (over unroll2) for Int *, FP +, FP *
  - Breaks sequential dependency
    ```c
    x0 = x0 OP d[i];
    x1 = x1 OP d[i+1];
    ```
Separate Accumulators

\[ x_0 = x_0 \text{ OP } d[i]; \]
\[ x_1 = x_1 \text{ OP } d[i+1]; \]

- **What changed:**
  - Two independent “streams” of operations

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be \((N/2+1)*D\) cycles:
    \[ \text{cycles per OP} \approx D/2 \]

What Now?

Unrolling & Accumulating

- **Idea**
  - Use K accumulators
  - Increase K until best performance reached
  - Need to unroll by L, K divides L

- **Limitations**
  - Diminishing returns:
    Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths: Finish off iterations sequentially
Unrolling & Accumulating: Intel FP *

- **Case**
  - Pentium 4
  - FP Multiplication
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>7.00</td>
</tr>
<tr>
<td>2</td>
<td>3.50</td>
</tr>
<tr>
<td>3</td>
<td>2.34</td>
</tr>
<tr>
<td>4</td>
<td>2.01</td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

**Why 4?**

Latency: 7 cycles

Those have to be independent

Based on this insight: \( K = \#\text{accumulators} = \text{ceil(latency/cycles per issue)} \)
### Unrolling & Accumulating: Intel FP +

#### Case
- Pentium 4
- FP Addition
- Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>Unrolling Factor L</th>
<th>( K )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP +</td>
<td></td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2.5</td>
<td>2.5</td>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.0</td>
</tr>
</tbody>
</table>

### Unrolling & Accumulating: Intel Int *

#### Case
- Pentium 4
- Integer Multiplication
- Theoretical Limit: 1.00

<table>
<thead>
<tr>
<th>Unrolling Factor L</th>
<th>( K )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int *</td>
<td></td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2.5</td>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1.67</td>
<td>1.67</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>1.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.14</td>
</tr>
</tbody>
</table>
## Unrolling & Accumulating: Intel Int +

- **Case**
  - Pentium 4
  - Integer addition
  - Theoretical Limit: 1.00

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K)</td>
<td>1</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>2.2</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
</tr>
<tr>
<td>4</td>
<td>1.1</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

### Pentium 4

### Core 2

*FP * is fully pipelined*
Summary (ILP)

- Instruction level parallelism may have to be made explicit in program

- Potential blockers for compilers
  - Reassociation changes result (FP)
  - Too many choices, no good way of deciding

- Unrolling
  - By itself does often nothing (branch prediction works usually well)
  - But may be needed to enable additional transformations
    (here: reassociation)

- How to program this example?
  - Solution 1: program generator generates alternatives and picks best
  - Solution 2: use model based on latency and throughput