Optimizing Program Performance Part 2
Effect of Basic Optimizations

• 4x to 18x improvement over original unoptimized code

• To seek better performance, we must consider optimizations that exploit the microarchitecture of the processor
  • Code tuned for a specific processor

• We’ll tackle this today

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    data_t = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 Unoptimized</td>
<td>22.68</td>
<td>20.02</td>
</tr>
<tr>
<td>Combine3</td>
<td>7.17</td>
<td>9.02</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>
Exploiting Instruction-Level Parallelism

• Need general understanding of modern processor design
  • Hardware can execute multiple instructions in parallel

• Performance limited by data dependencies

• Simple transformations can yield dramatic performance improvement
  • Compilers often cannot make these transformations
Superscalar Processor

• **Superscalar processors** can issue and execute *multiple instructions in one cycle*

• Most modern CPUs are superscalar
  • Intel: since Pentium (1993)

• Instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically

• Benefit: without programming effort, superscalar processor can take advantage of a program’s *instruction level parallelism*
Haswell CPU

- 8 Total Functional Units
- Multiple instructions can execute in parallel

Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Integer/Long Divide</strong></td>
<td><strong>3-30</strong></td>
<td><strong>3-30</strong></td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Single/Double FP Divide</strong></td>
<td><strong>3-15</strong></td>
<td><strong>3-15</strong></td>
</tr>
</tbody>
</table>
Pipelined Functional Units

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage $i$ can start on new computation once values passed to $i+1$
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

```c
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>a*b</td>
<td></td>
<td></td>
<td>a*c</td>
<td></td>
<td>p1*p2</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td>a*b</td>
<td></td>
<td>a*c</td>
<td></td>
<td></td>
<td>p1*p2</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>a*b</td>
<td>a*c</td>
<td></td>
<td></td>
<td>p1*p2</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

• Inner Loop (Case: Integer Multiply)

```
.L519:
imull (%rax,%rdx,4), %ecx # t = t * d[i]
addq $1, %rdx # i++
cmpq %rdx, %rbp # Compare length:i
jg .L519 # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Loop Unrolling (2x1)

• Perform 2x more useful work per iteration

```c
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long 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;
}
```
Effect of Loop Unrolling

- Helps integer add
  - Achieves latency bound
- Others don’t improve. *Why?*
  - Sequential dependency

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>

$x = (x \ OP \ d[i]) \ OP \ d[i+1]$;
Combine4 = Serial Computation (OP = *)

• Computation (length=8)
  
  ((((1 * d[0]) * d[1]) * d[2]) * d[3])  
  * d[4]) * d[5]) * d[6]) * d[7])

• Sequential dependence
  • Performance: determined by latency of OP
Loop Unrolling with Reassociation (2x1a)

- Can this change the result of the computation?
  - Yes, for floating point numbers.  Why?
    - Floating point numbers are not associative in all cases!

```c
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long 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;
}
```
Effect of Reassociation

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

\[ x = x \text{ OP } (d[i] \text{ OP } d[i+1]); \]
Loop Unrolling with Separate Accumulators (2x2)

• Different form of reassociation

```c
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long 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;
}
```
Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- 2x speedup (over unroll2x1) for Int *, FP +, FP *
- Int + makes use of two load units

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

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

- **Overall Performance**
  - $N$ elements, $D$ cycles latency/op
  - Should be $(N/2+1)\times D$ cycles:
    - $CPE \approx D/2$
  - $CPE$ matches prediction!

*What Now?*
Unrolling & Accumulating

• Idea
  • Can unroll to any degree L
  • Can accumulate K results in parallel
  • L must be multiple of K

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

- Case
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50 (Issue: 1, Capacity 2)

<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>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
</tr>
<tr>
<td>4</td>
<td></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>
Unrolling & Accumulating: Int +

• Case
  • Intel Haswell
  • Integer addition
  • Latency bound: 1.00. Throughput bound: 0.50

<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>1.27</td>
</tr>
<tr>
<td>2</td>
<td>0.81</td>
</tr>
<tr>
<td>3</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>0.69</td>
</tr>
<tr>
<td>6</td>
<td>0.56</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>
Factors Limiting Performance

• Why where there diminishing returns for loop unrolling and association?
  • Can’t exceed the parallelism of the functional units
  • Register spilling
    • We only have a fixed number of registers that can hold temporary values in memory
    • Extra values will be stored on the stack (in memory)

• Mispredicted branches
  • Pipelined processors must guess which way a branch will go
  • If wrong, must discard the incorrect instructions and start again
  • Converting code to use conditional moves instead of branching can help
    • Good if branching is unpredictable
    • Mostly not a concern as branch prediction is very accurate
Achievable Performance

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Programming with AVX2 (Advanced Vector Extensions)

- YMM Registers: 16 total, each 32 bytes
  - 32 single-byte integers
  - 16 16-bit integers
  - 8 32-bit integers
  - 8 single-precision floats
  - 4 double-precision floats
  - 1 single-precision float
  - 1 double-precision float
SIMD (Single Instruction Multiple Data) Operations

- SIMD Operations: Single Precision
  \[ \text{vaddsd} \ \text{ymm0}, \ \text{ymm1}, \ \text{ymm1} \]

- SIMD Operations: Double Precision
  \[ \text{vaddpd} \ \text{ymm0}, \ \text{ymm1}, \ \text{ymm1} \]
Using Vector Instructions

- Make use of AVX Instructions
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>
Getting High Performance

• Good compiler and flags
• Don’t do anything silly
  • Watch out for hidden algorithmic inefficiencies
  • Write compiler-friendly code
    • Watch out for optimization blockers: procedure calls & memory references
    • Look carefully at innermost loops (where most work is done)

• Tune code for machine
  • Exploit instruction-level parallelism
  • Make code cache friendly