Optimizing Program Performance

CMPU 224 – Computer Organization
Jason Waterman
Benchmark Example: Abstract Data Type for Vectors

```c
/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;
```

```c
/* retrieve vector element and store at val */
/* return 1 if successful, 0 otherwise */
int get_vec_element(*vec v, size_t idx, data_t *val) {
    if (idx >= v->len) {
        return 0;
    }
    *val = v->data[idx];
    return 1;
}
```

**Data Types**
- Use different declarations for `data_t`
  - int
  - long
  - float
  - double
Benchmark Computation

```c
void combine1(vec_ptr v, data_t *dest) {
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

• Data Types
  • Use different declarations for `data_t`
  • `int`
  • `long`
  • `float`
  • `double`

• Operations
  • Use different definitions of `OP` and `IDENT`
  • `+ / 0`
  • `* / 1`
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- \( T = \text{CPE} \times n + \text{Overhead} \)
  - CPE is slope of line

\[
\begin{align*}
\text{Cycles} & \quad \text{Elements} \\
0 & \quad 0 \\
500 & \quad 50 \\
1000 & \quad 100 \\
1500 & \quad 150 \\
2000 & \quad 200 \\
2500 & \quad 200
\end{align*}
\]

- \( \text{psum1} \) Slope = 9.0
- \( \text{psum2} \) Slope = 6.0
### Benchmark Performance

```c
void combine1(vec_ptr v, data_t *dest) {
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td><strong>Add</strong></td>
<td><strong>Mult</strong></td>
</tr>
<tr>
<td><strong>Add</strong></td>
<td>22.68</td>
<td>19.98</td>
</tr>
<tr>
<td><strong>Mult</strong></td>
<td>20.02</td>
<td>20.18</td>
</tr>
<tr>
<td><strong>Combine1 unoptimized</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Combine1 –O1</strong></td>
<td>10.12</td>
<td>10.17</td>
</tr>
<tr>
<td><strong>Combines1 –O1</strong></td>
<td>10.12</td>
<td>11.14</td>
</tr>
</tbody>
</table>
Eliminating Loop Inefficiencies

- Move call to `vec_length` outside of loop

- This is called code motion

```c
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

<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</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine2</td>
<td>7.02</td>
<td>9.03</td>
</tr>
</tbody>
</table>
Reducing Procedure Calls

- `get_vec_start` returns the starting address of the data array
  - This breaks the abstraction barrier!
- Remove call to `get_vec_element`
- We are no longer doing bounds checking
- No effect on performance
  - Other operations are the bottleneck
- Will help us later when we remove these bottlenecks

```c
/* Direct access to vector data*/
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

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

<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>Combine2</td>
<td>7.02</td>
<td>9.03</td>
</tr>
<tr>
<td>Combine3</td>
<td>7.17</td>
<td>9.02</td>
</tr>
</tbody>
</table>
Eliminating Unneeded Memory References

- Disassembly of inner loop for integer addition
- One memory read and one memory write per cycle
- But \*dest doesn’t need to be updated till the end of the loop

```c
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

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

.L3: # i in %rdx, length in %rbp, data in %rax
cmpq %rbp, %rdx    # i:length
jge .L1            # jump to end of loop
movq (%rax,%rdx,8), %rcx # read data[i]
addq %rcx, (%rbx)  # write *dest
addq $1, %rdx      # i++
jmp .L3
.L1:
```
Accumulate result in temporary

- Reduce to only one memory read per element using a temporary variable

```c
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

```c
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;
}
```

```asm
.L3:  # i in %rdx, data in %rax
cmpq %rbp, %rdx    # i:length
jge .L1            # jump to end of loop
movq (%rax,%rdx,8), %rcx   # read data[i]
addq %rcx, (%rbx)  # write *dest
addq $1, %rdx     # i++
jmp .L3           
.L1:               
```

```asm
.L3:  # data in %rax, i in %rdx, acc in %rcx
cmpq %rbp, %rdx    # i:length
jge .L1            # jump to end of loop
movq (%rax,%rdx,8), %rcx   # read data[i]
addq (%rcx, (%rbx))  # write *dest
addq $1, %rdx     # i++
jmp .L3           
.L1:               
```
Accumulate result in temporary

void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
data_t *data = get_vec_start(v)
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}

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;
}

• Why doesn’t the compiler perform this transformation automatically?
  • Memory aliasing
  • The dest could be one of the elements in the vector (e.g., the last element)
    • combine3 and combine4 could have different results in this case
  • Using a temporary register tells the compiler not to check for memory aliasing
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

<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>

```c
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;
}
```
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*
Modern CPU Design

Instruction Control

- Fetch
- Fetch Control
- Instruction Decode
- Instruction Cache
- Operations
- Instructions
- Address
- Register File
- Retirement Unit
- Prediction OK?
- Register Updates

Execution

- Branch
- Arith
- Arith
- Arith
- Load
- Store
- Functional Units
- Operation Results
- Addr.
- Data
- Addr.
- Data

Memory

Data Cache

5/2/2022

CMPU 224 -- Computer Organization
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>a*c</td>
<td></td>
<td>p1*p2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>a*b</td>
<td>a*c</td>
<td></td>
<td>p1*p2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>a*b</td>
<td>a*c</td>
<td></td>
<td>p1*p2</td>
<td></td>
<td></td>
<td></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

- Reduces overhead to integer add
  - Achieves latency bound
- Others don’t improve. *Why?*
  - Still sequential dependency

<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>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>

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

• Computation (length=8)
  \(((1 \times d[0]) \times d[1]) \times d[2]) \times d[3])
  \times d[4]) \times d[5]) \times d[6]) \times 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;
}
```

Compare to before

```c
x = (x OP d[i]) OP d[i+1];
```
Effect of Reassociation

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

\[ x = x \text{ OP} (d[i] \text{ OP} d[i+1]); \]

<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>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>

2 func. units for FP *
2 func. units for load
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

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

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```
Separate Accumulators

• What changed:
  • Two independent “streams” of operations

\[
x_0 = x_0 \text{ OP } d[i]; \\
x_1 = x_1 \text{ OP } d[i+1];
\]
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></td>
<td>K 1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td></td>
<td>1    5.01 5.01 5.01 5.01 5.01 5.01 5.01</td>
</tr>
<tr>
<td></td>
<td>2    2.51 2.51 2.51</td>
</tr>
<tr>
<td></td>
<td>3    1.67</td>
</tr>
<tr>
<td></td>
<td>4    1.25 1.26</td>
</tr>
<tr>
<td></td>
<td>6    0.84 0.88</td>
</tr>
<tr>
<td></td>
<td>8    0.63</td>
</tr>
<tr>
<td></td>
<td>10   0.51</td>
</tr>
<tr>
<td></td>
<td>12   0.52</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></td>
<td>K 1 2 3 4 6 8 10 12</td>
</tr>
<tr>
<td>1</td>
<td>1.27 1.01 1.01 1.01 1.01 1.01</td>
</tr>
<tr>
<td>2</td>
<td>0.81 0.69 0.54</td>
</tr>
<tr>
<td>3</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>0.69 1.24</td>
</tr>
<tr>
<td>6</td>
<td>0.56 0.56</td>
</tr>
<tr>
<td>8</td>
<td>0.56</td>
</tr>
<tr>
<td>10</td>
<td>0.54</td>
</tr>
<tr>
<td>12</td>
<td>0.56</td>
</tr>
</tbody>
</table>

Accumulators
### 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></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td>0.54</td>
<td>1.01</td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</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
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
End of the Semester Logistics

• Cache Assignment due: Sunday May 8\textsuperscript{th} 11:59pm
• Other Labs and Assignments
  • Check to make sure you have submitted your assignments
  • If not, come talk to me
• Final: Friday May 13\textsuperscript{th} 9:00 AM SP 309 (this room)
  • Review next week
  • Practice problems will be posted to the course website
• CEQs
  • https://go.vassar.edu/course/evals