More Loop Optimizations
Outline

- Strength Reduction
- Loop Test Replacement
- Loop Invariant Code Motion
- SIMDization with SSE
Strength Reduction

• Replace expensive operations in an expression using cheaper ones
  – Not a data-flow problem
  – Algebraic simplification
  – Example: $a \times 4 \Rightarrow a \ll 2$
Strength Reduction

- In loops reduce expensive operations in expressions into cheaper ones by using the previously calculated value
Strength Reduction

\[ t = 202 \]
\[ \text{for } j = 1 \text{ to } 100 \]
\[ t = t - 2 \]
\[ A(j) = t \]
Strength Reduction

t = 202

for j = 1 to 100
    t = t - 2
    *(abase + 4*j) = t
Strength Reduction

t = 202
for j = 1 to 100
    t = t - 2
    *(abase + 4*j) = t

Basic Induction variable:
J = 1, 2, 3, 4, ....

Induction variable 200 - 2*j
 t = 202, 200, 198, 196, ....

Induction variable abase+4*j:
abase+4*j = abase+4, abase+8, abase+12, abase+14, ....
Strength Reduction

\[ t = 202 \]

\[ \text{for } j = 1 \text{ to } 100 \]

\[ t = t - 2 \]

\[ *(abase + 4*j) = t \]

Basic Induction variable:
\( J = 1, 2, 3, 4, \ldots \)

Induction variable 200 - 2\( j \)
\( t = 202, 200, 198, 196, \ldots \)

Induction variable \( abase+4*j \):
\( abase+4*j = abase+4, abase+8, abase+12, abase+14, \ldots \)
Strength Reduction

\[ t = 202 \]

for \( j = 1 \) to 100

\[ t = t - 2 \]

\[ \ast (abase + 4\ast j) = t \]

Basic Induction variable:
\( J = 1, \quad 2, \quad 3, \quad 4, \ldots \)

Induction variable 200 - 2*j
\( t = 202, \quad 200, \quad 198, \quad 196, \ldots \)

Induction variable abase+4*j:
\( abase+4\ast j = abase+4, \quad abase+8, \quad abase+12, \quad abase+14, \ldots \)
Strength Reduction

\[ t = 202 \]

\[ \text{for } j = 1 \text{ to } 100 \]

\[ t = t - 2 \]

\[ *(abase + 4*j) = t \]

Basic Induction variable:
\[ J = 1, 2, 3, 4, \ldots \]

Induction variable 200 - 2*j
\[ t = 202, 200, 198, 196, \ldots \]

Induction variable abase+4*j:
\[ \text{abase} + 4*j = \text{abase} + 4, \text{abase} + 8, \text{abase} + 12, \text{abase} + 14, \ldots \]
Strength Reduction Algorithm

• For a dependent induction variable \( k = a \cdot j + b \)

\[
\text{for } j = 1 \text{ to } 100 \\
*(\text{abase} + 4 \cdot j) = j
\]
Strength Reduction Algorithm

• For a dependent induction variable $k = a \cdot j + b$
• Add a pre-header $k' = a \cdot j_{init} + b$

$$t = abase + 4 \cdot 1$$
for $j = 1$ to 100

$$(abase + 4 \cdot j) = j$$
Strength Reduction Algorithm

• For a dependent induction variable $k = a \cdot j + b$
• Add a pre-header $k' = a \cdot j_{init} + b$
• Next to $j = j + c$ add $k' = k' + a \cdot c$

$t = abase + 4 \cdot 1$
for $j = 1$ to $100$
  *(abase + 4\cdot j) = j$
  $t = t + 4$
Strength Reduction Algorithm

- For a dependent induction variable \( k = a \cdot j + b \)
- Add a pre-header \( k' = a \cdot j_{\text{init}} + b \)
- Next to \( j = j + c \) add \( k' = k' + a \cdot c \)
- Use \( k' \) instead of \( k \)

\[
\begin{align*}
t &= a_{\text{base}} + 4 \cdot 1 \\
\text{for } j = 1 \text{ to } 100 \\
\quad *(t) &= j \\
\quad t &= t + 4
\end{align*}
\]
Example

double A[256], B[256][256]
j = 1

while(j>100)
    A[j] = B[j][j]
    j = j + 2
Example

```c
double A[256], B[256][256]
j = 1

while(j>100)
    *(&A + 4*j) = *(&B + 4*(256*j + j))
j = j + 2
```
Example

double A[256], B[256][256]
j = 1

while(j>100)
    *(&A + 4*j) = *(&B + 4*(256*j + j))
j = j + 2

Base Induction Variable: j
Example

double A[256], B[256][256]
j = 1

while(j>100)
    *(&A + 4*j) = *(&B + 4*(256*j + j))
j = j + 2

Base Induction Variable: j
Dependent Induction Variable: a = &A + 4*j
Example

double A[256], B[256][256]
j = 1
a = &A + 4

while(j>100)
   *(&A + 4*j) = *(&B + 4*(256*j + j))
j = j + 2

Base Induction Variable: j
Dependent Induction Variable: a = &A + 4*j
Example

double A[256], B[256][256]

j = 1
a = &A + 4

while(j>100)
    *(&A + 4*j) = *(&B + 4*(256*j + j))
    j = j + 2
    a = a + 8

Base Induction Variable: j
Dependent Induction Variable: a = &A + 4*j
Example

double A[256], B[256][256]
j = 1
a = &A + 4

while(j>100)
    *a = *(&B + 4*(256*j + j))
j = j + 2
a = a + 8

Base Induction Variable:  j
Dependent Induction Variable:  a = &A + 4*j
Example

double A[256], B[256][256]
j = 1
a = &A + 4

while(j>100)
    *a = *(&B + 4*(256*j + j))
j = j + 2
a = a + 8

Base Induction Variable:  j
Dependent Induction Variable: b = &B + 4*257*j
Example

double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while (j > 100)
  *a = *(&B + 4*(256*j + j))
  j = j + 2
  a = a + 8

Base Induction Variable:  j
Dependent Induction Variable:  b = &B + 4*257*j
double A[256], B[256][256]  
j = 1  
a = &A + 4  
b = &B + 1028  
while(j>100)  
  *a = *(&B + 4*(256*j + j))  
  j = j + 2  
  a = a + 8  
  b = b + 2056

Base Induction Variable: j  
Dependent Induction Variable: b = &B + 4*257*j
Example

double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(j>100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056

Base Induction Variable: j
Dependent Induction Variable: b = &B + 4*257*j
Example

double A[256], B[256][256]

j = 1
a = &A + 4
b = &B + 1028

while (j > 100)
  *a = *b
  j = j + 2
  a = a + 8
  b = b + 2056
Outline

• Strength Reduction
• Loop Test Replacement
• Loop Invariant Code Motion
• SIMDization with SSE
Loop Test Replacement

• Eliminate basic induction variable used only for calculating other induction variables
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
j = 1
while(j>100)
    A[j] = B[j][j]
```
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(j>100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056
```
Loop Test Replacement

• Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(j > 100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056
```

• J is only used for the loop bound
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while (j > 100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056

- J is only used for the loop bound
- Use a dependent IV (a or b)
```
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]

j = 1
a = &A + 4
b = &B + 1028

while(j>100)
  *a = *b
  j = j + 2
  a = a + 8
  b = b + 2056
```

- J is only used for the loop bound
- Use a dependent IV (a or b)
- Lets choose a
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(j>100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056
```

- J is only used for the loop bound
- Use a dependent IV (a or b)
- Lets choose a
  
  \[
  j > 100 \implies a > &A + 800
  \]
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(a>&A+800)
  *a = *b
  j = j + 2
  a = a + 8
  b = b + 2056
```

- J is only used for the loop bound
- Use a dependent IV (a or b)
- Lets choose a
  
  \[ j > 100 \implies a > &A + 800 \]
- Replace the loop condition
Loop Test Replacement

• Eliminate basic induction variable used only for calculating other induction variables

\[
\text{double } A[256], B[256][256]
\]

\[
a = &A + 4 \\
b = &B + 1028 \\
\text{while}(a >&A+800) \\
\quad \ast a = \ast b \\
\quad a = a + 8 \\
b = b + 2056
\]

• J is only used for the loop bound
• Use a dependent IV (a or b)
• Lets choose a
  \[
  j > 100 \Rightarrow a > &A + 800
  \]
• Replace the loop condition
• Get rid of j
Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```c
double A[256], B[256][256]
a = &A + 4
b = &B + 1028
while(a>&A+800)
  *a = *b
  a = a + 8
  b = b + 2056
```
Loop Test Replacement Algorithm

• If basic induction variable J is only used for calculating other induction variables

• Select an induction variable k in the family of J (K = a*J + b)

• Replace a comparison such as
  \[ \text{if } (J > X) \text{ goto } L1 \]
  by
  
  \[ \text{if} (K' > a*X + b) \text{ goto } L1 \] if a is positive
  
  \[ \text{if} (K' < a*X + b) \text{ goto } L1 \] if a is negative

• If J is live at any exit from loop, recompute
  
  \[ J = \frac{(K' - b)}{a} \]
Outline

• Strength Reduction
• Loop Test Replacement
• Loop Invariant Code Motion
• SIMDization with SSE
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop
- Same idea as with induction variables
  - Variables not updated in the loop are loop invariant
  - Expressions of loop invariant variables are loop invariant
  - Variables assigned only loop invariant expressions are loop invariant
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[
\text{for } i = 1 \text{ to } N \\
\quad x = x + 1 \\
\text{for } j = 1 \text{ to } N \\
\quad a(i,j) = 100*N + 10*i + j + x
\]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[
\begin{align*}
\text{for } i = 1 \text{ to } N \\
x &= x + 1 \\
\text{for } j = 1 \text{ to } N \\
a(i,j) &= 100*N + 10*i + j + x
\end{align*}
\]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[ t1 = 100 \times N \]
\[ \text{for } i = 1 \text{ to } N \]
\[ x = x + 1 \]
\[ \text{for } j = 1 \text{ to } N \]
\[ a(i,j) = 100 \times N + 10 \times i + j + x \]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[ t1 = 100*N \]

\[ \text{for } i = 1 \text{ to } N \]
\[ x = x + 1 \]
\[ \text{for } j = 1 \text{ to } N \]
\[ a(i,j) = t1 + 10*i + j + x \]
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

\[ t_1 = 100 \times N \]

\[
\text{for } i = 1 \text{ to } N \\
\quad x = x + 1 \\
\quad \text{for } j = 1 \text{ to } N \\
\quad \quad a(i,j) = t_1 + 10 \times i + j + x
\]
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

\[ t1 = 100*N \]

for i = 1 to N

\[ x = x + 1 \]

for j = 1 to N

\[ a(i,j) = t1 + 10*i + j + x \]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[ t1 = 100 \times N \]

\[ \text{for } i = 1 \text{ to } N \]

\[ x = x + 1 \]

\[ t2 = t1 + 10 \times i + x \]

\[ \text{for } j = 1 \text{ to } N \]

\[ a(i,j) = t1 + 10 \times i + j + x \]
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```plaintext
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = t1 + 10*i + x
for j = 1 to N
  a(i,j) = t2 + j
```
Outline

• Strength Reduction
• Loop Test Replacement
• Loop Invariant Code Motion
• **SIMDization with SSE**
SIMD Through SSE extensions

• Single Instruction Multiple Data
  – Compute multiple identical operations in a single instruction
  – Exploit fine grained parallelism
SSE Registers

- 16 128-bit registers: %xmm0 to %xmm16
  - Multiple interpretations for each register
  - Each arithmetic operation comes in multiple versions
Data Transfer

- Moving Data From Memory or xmm registers
  - MOVDQA $OP1$, $OP2$ Move *aligned* Double Quadword
    - Can read or write to memory in 128 bit chunks
    - If OP1 or OP2 are registers, they must be xmm registers
    - Memory locations in OP1 or OP2 must be multiples of 16
  - MOVDQU $OP1$, $OP2$ Move *unaligned* Double Quadword
    - Same as MOVDQA but
    - Memory addresses don’t have to be multiples of 16
Data Transfer

- Moving Data From 64-bit registers
  - **MOVQ** *OP1, OP2*  Move Double Quadword
    - Can move from 64 bit register to xmm register or vice versa
    - Writes to/Reads from the lower 64 bits of xmm register
    - Can also be used to read a 64-bit chunk to/from memory

**MOVQ** %r11, %xmm1

%r11

%xmm1

128

0
Data Reordering

• Unpack and Interleave
  – PUNPCKLDQ Low Doublewords
Data Reordering

• Unpack and Interleave
  – PUNPCKLQDQ Low Quadwords
Arithmetic

• Arithmetic operations come in many flavors
  – based on the datatype of the register
  – specified in the instruction suffix

• Example: Addition
  – PADDQ Add 64-bit Quadwords
  – PADDD Add 32-bit Doublewords
  – PADDW Add 16-bit words

• Example: Subtraction
  – PSUBQ Subtract 64-bit Quadwords
  – PSUBD Subtract 32-bit Doublewords
  – PSUBW Subtract 16-bit words
Putting It All Together

• Source Code

\[
\text{for } i = 1 \text{ to } N \\
A[i] = A[i] \times b
\]

• After Unrolling

\[
\text{loop:}
\]

\[
\begin{align*}
\text{mov} &\quad (%rdi,%rax), %r10 \\
\text{mov} &\quad (%rdi,%rbx), %rcx \\
\text{imul} &\quad %r11, %r10 \\
\text{imul} &\quad %r11, %rcx \\
\text{mov} &\quad %r10, (%rdi,%rax) \\
\text{sub} &\quad $8, %rax \\
\text{mov} &\quad %rcx, (%rdi,%rbx) \\
\text{sub} &\quad $8, %rbx \\
\text{jz} &\quad \text{loop}
\end{align*}
\]

Reading from consecutive addresses

Mult by a loop invariant

Writing to consecutive addresses

Saman Amarasinghe
Putting it all together

<table>
<thead>
<tr>
<th>Original Version</th>
<th>SSE Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop:</strong></td>
<td><strong>loop:</strong></td>
</tr>
<tr>
<td>mov (%rdi,%rax), %r10</td>
<td>movq %r11, %xmm2</td>
</tr>
<tr>
<td>mov (%rdi,%rbx), %rcx</td>
<td>punpckldq %xmm2, %xmm2</td>
</tr>
<tr>
<td>imul %r11, %r10</td>
<td>movdqa (%rdi,%rax), %xmm0</td>
</tr>
<tr>
<td>imul %r11, %rcx</td>
<td>pmuludq %xmm2, %xmm0</td>
</tr>
<tr>
<td>mov %r10, (%rdi,%rax)</td>
<td>movdqa %xmm0, (%rdi,%rax)</td>
</tr>
<tr>
<td>sub $8, %rax</td>
<td>sub $8, %rax</td>
</tr>
<tr>
<td>mov %rcx, (%rdi,%rbx)</td>
<td>jz loop</td>
</tr>
<tr>
<td>sub $8, %rbx</td>
<td></td>
</tr>
<tr>
<td>jz loop</td>
<td></td>
</tr>
</tbody>
</table>
Putting it all together

SSE Version

```assembly
movq %r11, %xmm2
punpckldqw %xmm2, %xmm2

loop:
  movdqa (%rdi,%rax), %xmm0
  pmuludq %xmm2, %xmm0
  movdqa %xmm0, (%rdi, %rax)
  sub $8, %rax
  jz loop
```

Populate xmm2 with copies of %r11

Only one index is needed
Conditions for SIMDization

- Consecutive iterations reading and writing from consecutive locations
- Consecutive iterations are independent of each other
- The easiest thing is to pattern match at the basic block level after unrolling loops