6.035

Spring 2010

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: $a*4 \Rightarrow a << 2$

Strength Reduction

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

Strength Reduction

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

Basic Induction variable:

J = 1, $\xrightarrow{1}$ 2, $\xrightarrow{1}$ 3, $\xrightarrow{1}$ 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
for j ¯ 1 to 100
  t = t - 2
  *(abase + 4*j) = t
```

Basic Induction variable:

J = 1, $\xrightarrow{1}$ 2, $\xrightarrow{1}$ 3, $\xrightarrow{1}$ 4, …..

Induction variable 200 - 2*j

t = 202, $\xrightarrow{-2}$ 200, $\xrightarrow{-2}$ 198, $\xrightarrow{-2}$ 196, …..

Induction variable abase+4*j:

abase+4*j = abase+4, abase+8, abase+12, abase+14, ….

Strength Reduction

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

Basic Induction variable:

$J = 1, \xrightarrow{1} 2, \xrightarrow{1} 3, \xrightarrow{1} 4, \ldots..$

Induction variable 200 - 2*j

$t = 202, \xrightarrow{-2} 200, \xrightarrow{-2} 198, \xrightarrow{-2} 196, \ldots..$

Induction variable abase+4*j:

$abase+4*j = abase+4, \xrightarrow{4} abase+8, \xrightarrow{4} abase+12, \xrightarrow{4} abase+14, \ldots.$

Strength Reduction Algorithm

- For a dependent induction variable k = a*j + b

```
for j = 1 to 100
  *(abase + 4*j) = j
```

Strength Reduction Algorithm

- For a dependent induction variable k = a*j+ b
- Add a pre-header k' = a*jinit + b

```
t = abase + 4*1
for j = 1 to 100
  *(abase + 4*j) = j
```

Strength Reduction Algorithm

- For a dependent induction variable k = a*j + b
- Add a pre-header k’ = a*jinit + b
- Next to j − j + c add k’ − k’ + a*c

```
t = abase + 4*1
for j = 1 to 100
  *(abase + 4*j) = j
   t = t + 4
```

Strength Reduction Algorithm

- For a dependent induction variable k = a*j + b
- Add a pre-header k’ = a*jinit + b
- Next to j − j + c add k’ − k’ + a*c
- Use k’ instead of k

```
t = abase + 4*1
for j = 1 to 100
  *(t) = j
   t = t + 4
```

Example

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


while(j>100)
   A[j] = B[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
```

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

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
   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

```
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

```
 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

```
 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

```
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

```
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

```
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

  $j > 100 \Rightarrow a > \&A + 800$

Loop Test Replacement

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

```
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

  $j > 100 \Rightarrow a > \&A + 800$
- Replace the loop condition

Loop Test Replacement

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

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

a = &A + 4
b = &B + 1028
while(a>&A+800)
    *a = *b

    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 ⇒ 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

```
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

```
if (J > X) goto L1
```

by

```
if(K' > a*X + b) goto L1
```

if a is positive

```
if(K' < a*X + b) goto L1
```

if a is negative

- If J is live at any exit from loop, recompute

```
J = (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

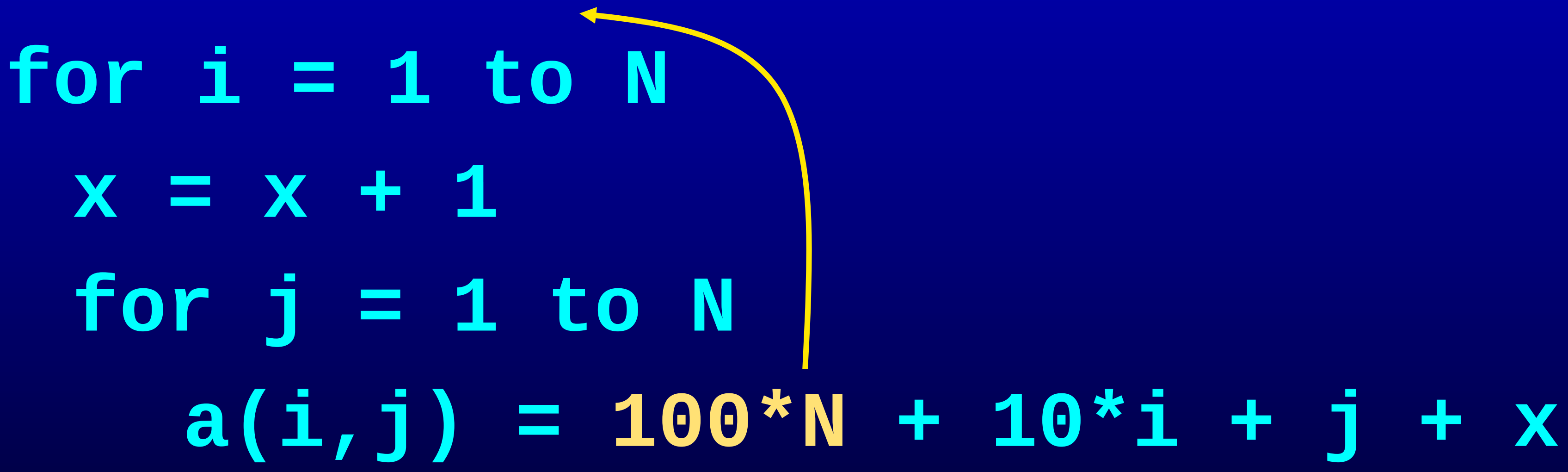
Loop Invariant Code Motion

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

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
    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



```
for i = 1 to N
  x = x + 1
  for j = 1 to N
     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

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
    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

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

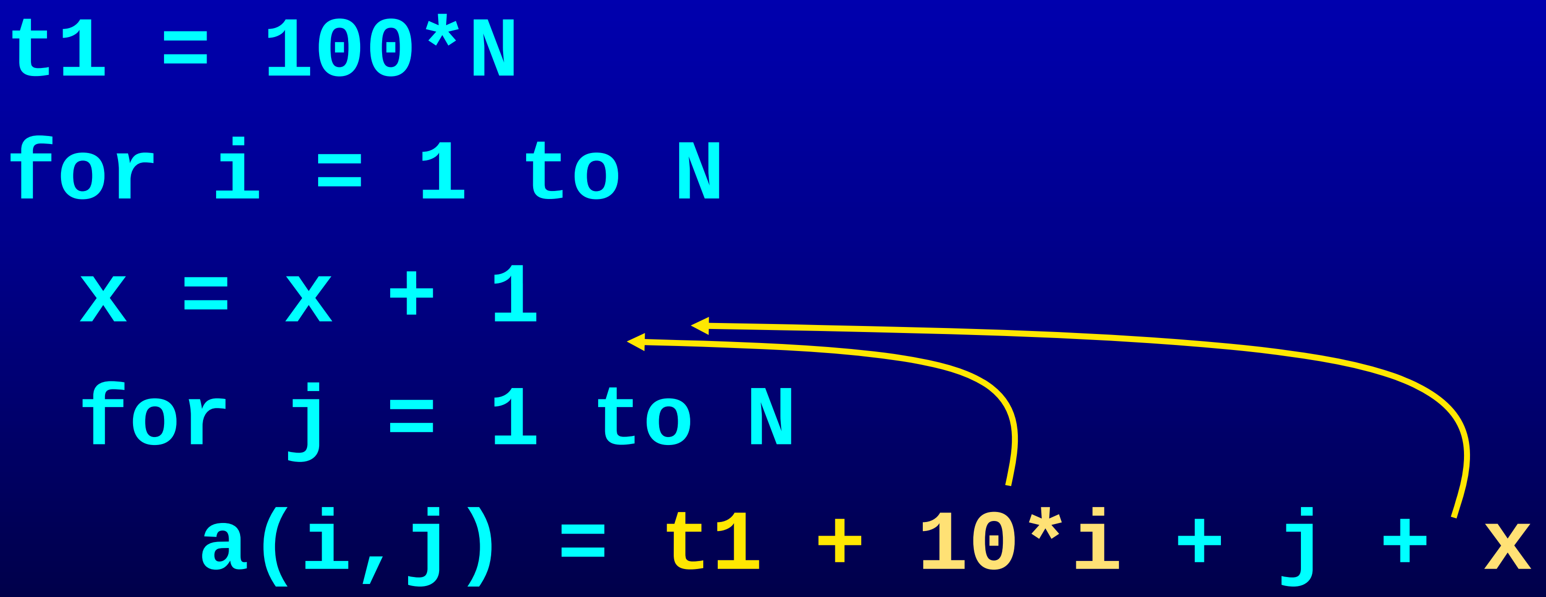
Loop Invariant Code Motion

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

```
t1 = 100*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

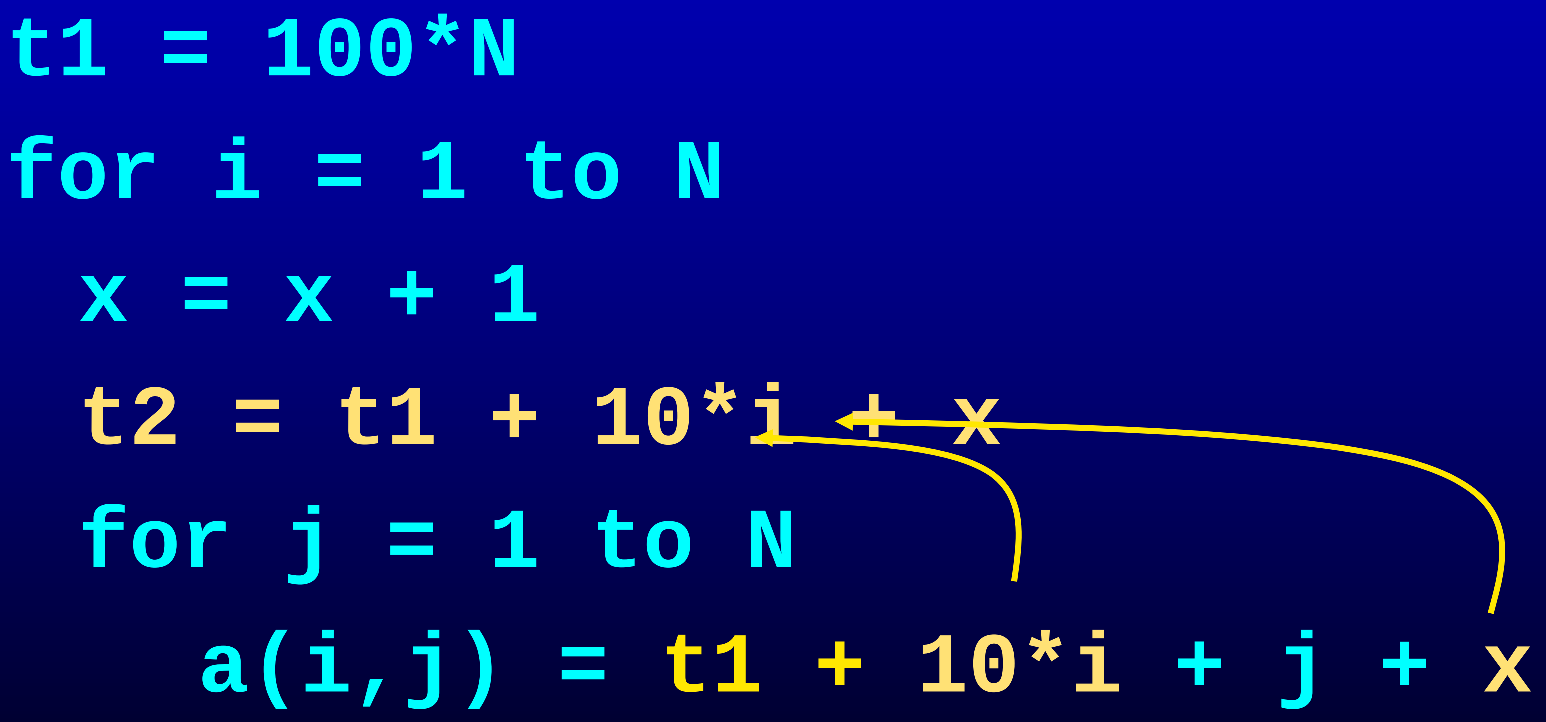
- If a computation produces the same value in every loop iteration, move it out of the loop



```
t1 = 100*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*N
for i = 1 to N
  x = x + 1
  t2 = t1 + 10*i + x
  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*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

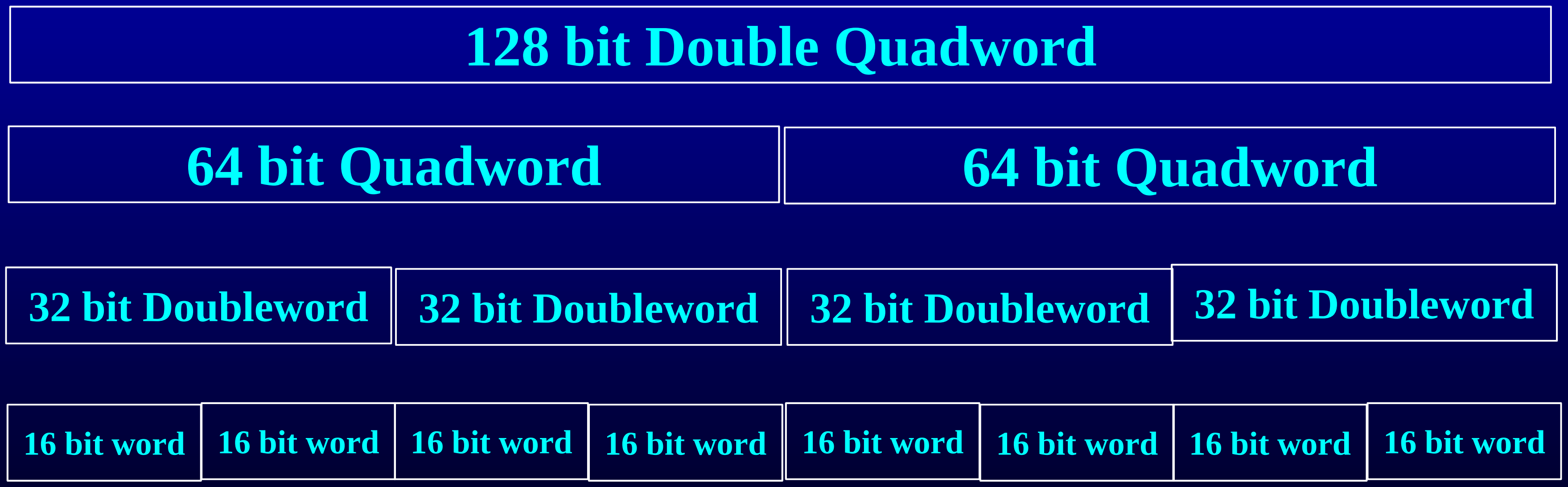
- Strength Reduction
- Loop Test Replacement
- Loop Invariant Code Motion
- **SIMDization with SSE**

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



128 bit Double Quadword

64 bit Quadword **64 bit Quadword**

32 bit Doubleword **32 bit Doubleword** **32 bit Doubleword** **32 bit Doubleword**

16 bit word **16 bit word** **16 bit word** **16 bit word** **16 bit word** **16 bit word** **16 bit word** **16 bit word**

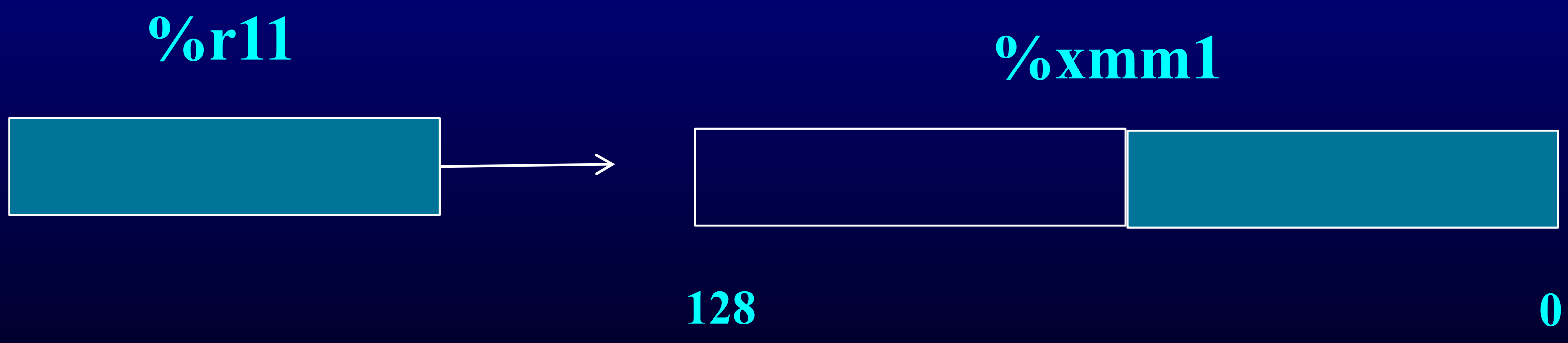
Data Transfer

- 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

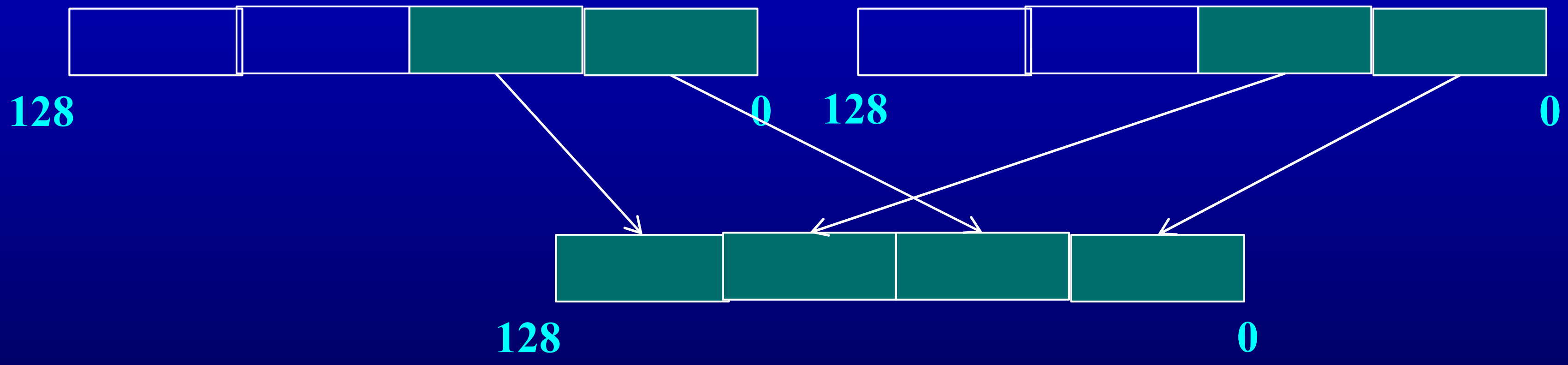
- Moving Data From 64-bit registers
  - MOVQ *OP1*, *OP2* Move Double Quadword
    - Can move from 64 bit register to xmm register or viceversa
    - Writes to/Reads from the lower 64 bits of xmm register
    - Can also be used to read a 64-bit chunk to/from memory

MOVQ %r11, %xmm1



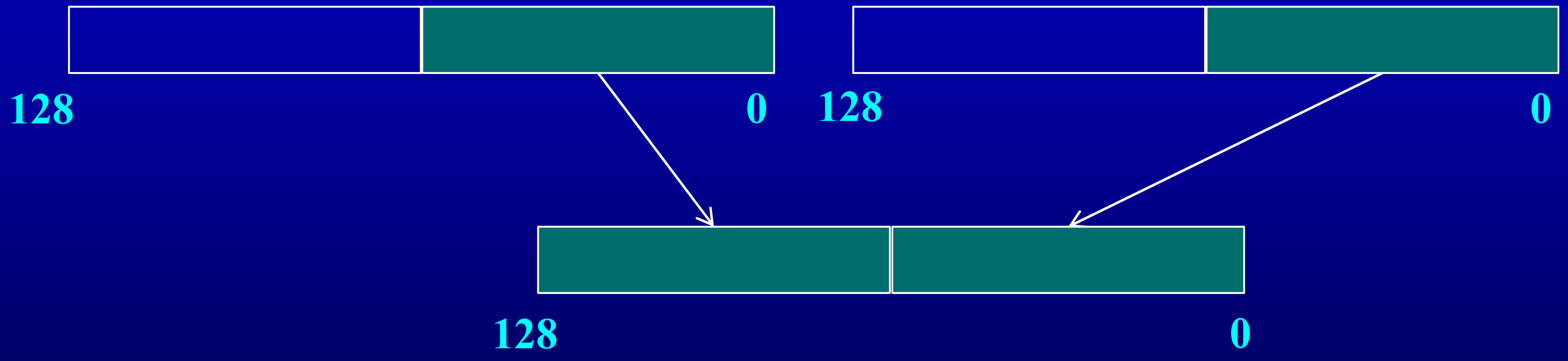
Data Reordering

- 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 sufix
- 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

```
for i = 1 to N
   A[i] = A[i] * b
```

- After Unrolling

```
loop:
    mov     (%rdi,%rax), %r10
    mov      (%rdi,%rbx), %rcx
    imul    %r11, %r10
    imul    %r11, %rcx
    mov     %r10, (%rdi,%rax)
    sub     $8, %rax
    mov     %rcx, (%rdi,%rbx)
    sub     $8, %rbx
    jz      loop
```

Reading from consecutive addresses

Mult by a loop invariant

Writing to consecutive addresses

Putting it all together

Original Version

```
loop:
    mov      (%rdi,%rax), %r10
    mov      (%rdi,%rbx), %rcx
    imul     %r11, %r10
    imul     %r11, %rcx
    mov      %r10, (%rdi,%rax)
    sub      $8, %rax
    mov      %rcx, (%rdi,%rbx)
    sub      $8, %rbx
    jz       loop
```

SSE Version

```
    movq      %r11, %xmm2
    punpckldq %xmm2, %xmm2
loop:
    movdqa    (%rdi,%rax), %xmm0
    pmuludq   %xmm2, %xmm0
    movdqa    %xmm0, (%rdi, %rax)
    sub       $8, %rax
    jz        loop
```

Putting it all together

SSE Version

```
    movq      %r11, %xmm2
    punpckldq %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

6.035 Computer Language Engineering
Spring 2010