Pre-class Puzzle

The following code multiplies two \( N \times N \) matrices, \( a \) and \( b \). What equivalent code would likely execute faster?

```c
for (int j = 0; j < N; j++) {
    for (int i = 0; i < N; i++) {
        c[i][j] = 0;
        for (int k = 0; k < N; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```
Announcements

• Project 5 out
  • Due Tuesday Nov 13 (7 days)

• Project 6 out
  • Due Tuesday Nov 20 (14 days)

• Project 7 out
  • Due Thursday Nov 29 (23 days)
Today

- Loop optimization ctd
  - Loop-invariant removal
  - Induction variable reduction
  - Loop fusion
  - Loop fission
  - Loop unrolling
  - Loop interchange
  - Loop peeling
  - Loop tiling
  - Loop parallelization
Loop-Invariant Removal
Loop Invariants

• An assignment \( x := v_1 \text{ op } v_2 \) is invariant for a loop if for each operand \( v_1 \) and \( v_2 \) either
  • the operand is constant, or
  • all of the definitions that reach the assignment are outside the loop, or
  • only one definition reaches the assignment and it is a loop invariant
Example

L0:  \( t := 0 \)
     \( a := x \)
L1:  \( i := i + 1 \)
     \( b := 7 \)
     \( t := a + b \)
     \( *i := t \)
     if \( i < N \) goto L1 else L2

L2:  \( x := t \)
Hoisting

• We would like to **hoist** invariant computations out of the loop

• But this is trickier than it sounds:
  • We need to potentially introduce an extra node in the CFG, right before the header to place the hoisted statements (the **pre-header**)
  • Even then, we can run into trouble…
Valid Hoisting Example

L0:  \( t := 0 \)

L1:  \( i := i + 1 \)
     \( t := a + b \)
     \( *i := t \)
     \( \text{if } i<N \text{ goto L1 else L2} \)

L2:  \( x := t \)
Valid Hoisting Example

L0:  \( t := 0 \)

\[
\begin{align*}
t &:= a + b
\end{align*}
\]

L1:  \( i := i + 1 \)

\[
\begin{align*}
*i &:= t \\
\text{if } i < N \text{ goto L1 else L2}
\end{align*}
\]

L2:  \( x := t \)
Invalid Hoisting Example

L0: \( t := 0 \)

L1: \( i := i + 1 \)
\[ *i := t \]
\( t := a + b \)
if \( i < N \) goto L1 else L2

L2: \( x := t \)
Conditions for Safe Hoisting

- An invariant assignment \( d: x := v_1 \text{ op } v_2 \) is safe to hoist if:
  - \( d \) dominates all loop exits at which \( x \) is live and
  - there is only one definition of \( x \) in the loop, and
  - \( x \) is not live at the entry point for the loop (the pre-header)
Induction Variable Reduction
Induction Variables

Can express $j$ and $k$ as linear functions of $i$ where the coefficients are either constants or loop-invariant

- $j = 4i + 0$
- $k = 4i + a$

```
s := 0
i := 0
L1: if i >= n goto L2
    j := i*4
    k := j+a
    x := *k
    s := s+x
    i := i+1
L2: ...
```
Induction Variables

s := 0
i := 0
L1: if i >= n goto L2
j := i\times 4
k := j+a
x := k
s := s+x
i := i+1
L2: ...

• Note that $i$ only changes by the same amount each iteration of the loop
• We say that $i$ is a **linear induction variable**
• It’s easy to express the change in $j$ and $k$
  • Since $j = 4\times i + 0$ and $k = 4\times i + a$, if $i$ changes by $c$, $j$ and $k$ change by $4\times c$
Detecting Induction Variables

**Definition:** \( i \) is a **basic induction variable** in a loop \( L \) if the only definitions of \( i \) within \( L \) are of the form \( i := i + c \) or \( i := i - c \) where \( c \) is loop invariant.

**Definition:** \( k \) is a **derived induction variable** in loop \( L \) if:

1. There is only one definition of \( k \) within \( L \) of the form \( k := j \cdot c \) or \( k := j + c \) where \( j \) is an induction variable and \( c \) is loop invariant; and
2. If \( j \) is an induction variable in the family of \( i \) (i.e., linear in \( i \)) then:
   - the only definition of \( j \) that reaches \( k \) is the one in the loop; and
   - there is no definition of \( i \) on any path between the definition of \( j \) and the definition of \( k \)

If \( k \) is a derived induction variable in the family of \( j \) and \( j = a \cdot i + b \) and, say, \( k := j \cdot c \), then \( k = a \cdot c \cdot i + b \cdot c \).
Strength Reduction

• For each derived induction variable $j$ where $j = e_1 \times i + e_0$ make a fresh temp $j'$
• At the loop pre-header, initialize $j'$ to $e_0$
• After each $i := i + c$, define $j' := j' + (e_1 \times c)$
  • note that $e_1 \times c$ can be computed in the loop header (i.e., it’s loop invariant)
• Replace the unique assignment of $j$ in the loop with $j := j'$
Example

\begin{align*}
  &s := 0 \\
  &i := 0 \\
  &L1: \text{ if } i &\geq n \text{ goto } L2 \\
  &j := i*4 \\
  &k := j+a \\
  &x := *k \\
  &s := s+x \\
  &i := i+1 \\
  &L2: \ldots
\end{align*}

• \textbf{i} is basic induction variable
• \textbf{j} is derived induction variable in family of \textbf{i}
  \begin{itemize}
    \item \textbf{j} = 4*i + 0
  \end{itemize}
• \textbf{k} is derived induction variable in family of \textbf{j}
  \begin{itemize}
    \item \textbf{k} = 4*i + a
  \end{itemize}
Example

\[ s := 0 \]
\[ i := 0 \]
\[ j' := 0 \]
\[ k' := a \]

L1:  if \( i \geq n \) goto L2
\[ j := i \times 4 \]
\[ k := j + a \]
\[ x := *k \]
\[ s := s + x \]
\[ i := i + 1 \]

L2:  ...

- \( i \) is basic induction variable
- \( j \) is derived induction variable in family of \( i \)
  - \( j = 4 \times i + 0 \)
- \( k \) is derived induction variable in family of \( j \)
  - \( k = 4 \times i + a \)
Example

i is basic induction variable

j is derived induction variable in family of i

k is derived induction variable in family of j

L1: 
    if i >= n goto L2
    j := i*4
    k := j+a
    x := *k
    s := s+x
    i := i+1

    j' := j' + 4
    k' := k' + 4

L2: ...

s := 0
i := 0
j' := 0
k' := a

j := i*4
k := j+a
x := *k
s := s+x
i := i+1

j' := j' + 4
k' := k' + 4
Example

\begin{itemize}
\item $i$ is basic induction variable
\item $j$ is derived induction variable in family of $i$
  \begin{itemize}
  \item $j = 4 \cdot i + 0$
  \end{itemize}
\item $k$ is derived induction variable in family of $j$
  \begin{itemize}
  \item $k = 4 \cdot i + a$
  \end{itemize}
\end{itemize}

\begin{verbatim}
s := 0
i := 0
j' := 0
k' := a

L1:  if i >= n goto L2
j := j'
k := k'
x := *k
s := s+x
i := i+1
j' := j'+4
k' := k'+4

L2:  ...
\end{verbatim}
Example

\[ s := 0 \]
\[ i := 0 \]
\[ j' := 0 \]
\[ k' := a \]

L1:
\[ \text{if } i \geq n \text{ goto } L2 \]
\[ x := *k' \]
\[ s := s+x \]
\[ i := i+1 \]
\[ j' := j'+4 \]
\[ k' := k'+4 \]

L2:
\[ ... \]

• \( i \) is basic induction variable
• \( j \) is derived induction variable in family of \( i \)
  • \( j = 4*i + 0 \)
• \( k \) is derived induction variable in family of \( j \)
  • \( k = 4*i + a \)
Useless Variables

A variable is **useless** for $L$ if it is dead at all exits from $L$ and its only use is in a definition of itself

- E.g., $j'$ is useless

- Can delete useless variables

```
s := 0
i := 0
j' := 0
k' := a

L1: if i >= n goto L2
x := *k'
s := s+x
i := i+1
j' := j'+4
k' := k'+4

L2: ...
```
Useless Variables

- A variable is **useless** for L if it is dead at all exits from L and its only use is in a definition of itself
  - E.g., \( j' \) is useless
- Can delete useless variables

```
s := 0
i := 0
j' := 0
k' := a

L1: if i >= n goto L2
    x := *k'
    s := s+x
    i := i+1
    k' := k'+4

L2: ...
```
Useless Variables

A variable is **useless** for $L$ if it is dead at all exits from $L$ and its only use is in a definition of itself

- E.g., $j'$ is useless
- Can delete useless variables

s := 0
i := 0
k' := a

L1: if i >= n goto L2
x := *k'
s := s+x
i := i+1
k' := k'+4

L2: ...
Almost Useless Variables

A variable is almost useless for \( L \) if it is used only in comparison against loop invariant values and in definitions of itself, and there is some other non-useless induction variable in same family.

- E.g., \( i \) is useless

An almost-useless variable may be made useless by modifying comparison.
- E.g., See Appel for details

```plaintext
s := 0
i := 0
k′ := a

L1: if i >= n goto L2
    x := *k′
    s := s + x
    i := i + 1
    k′ := k′ + 4

L2: ...
```
Loop Fusion and Loop Fission

- Fusion: combine two loops into one
- Fission: split one loop into two
Loop Fusion

• Before
  ```
  int acc = 0;
  for (int i = 0; i < n; ++i) {
    acc += a[i];
    a[i] = acc;
  }
  for (int i = 0; i < n; ++i) {
    b[i] += a[i];
  }
  ```

• After
  ```
  int acc = 0;
  for (int i = 0; i < n; ++i) {
    acc += a[i];
    a[i] = acc;
    b[i] += acc;
  }
  ```

• What are the potential benefits? Costs?

• Locality of reference
Loop Fission

• Before

```c
for (int i = 0; i < n; ++i) {
    a[i] = e1;
    b[i] = e2;  // e1 and e2 independent
}
```

• After

```c
for (int i = 0; i < n; ++i) {
    a[i] = e1;
}
for (int i = 0; i < n; ++i) {
    b[i] = e2;
}
```

• What are the potential benefits? Costs?
• Locality of reference
Loop Unrolling

- Make copies of loop body
- Say, each iteration of rewritten loop performs 3 iterations of old loop
Loop Unrolling

• Before

```c
for (int i = 0; i < n; ++i) {
    a[i] = b[i] * 7 + c[i] / 13;
}
```

• After

```c
for (int i = 0; i < n % 3; ++i) {
    a[i] = b[i] * 7 + c[i] / 13;
}
for (; i < n; i += 3) {
    a[i] = b[i] * 7 + c[i] / 13;
    a[i + 1] = b[i + 1] * 7 + c[i + 1] / 13;
    a[i + 2] = b[i + 2] * 7 + c[i + 2] / 13;
}
```

• What are the potential benefits? Costs?
• Reduce branching penalty, end-of-loop-test costs
• Size of program increased
Loop Unrolling

• If fixed number of iterations, maybe turn loop into sequence of statements!
• Before
  
  ```
  for (int i = 0; i < 6; ++i) {
    if (i % 2 == 0) foo(i); else bar(i);
  }
  ```

• After
  
  ```
  foo(0);
  bar(1);
  foo(2);
  bar(3);
  foo(4);
  bar(5);
  ```
Loop Interchange

- Change order of loop iteration variables
Loop Interchange

• Before

```cpp
for (int j = 0; j < n; ++j) {
    for (int i = 0; i < n; ++i) {
        a[i][j] += 1;
    }
}
```

• After

```cpp
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        a[i][j] += 1;
    }
}
```

• What are the potential benefits? Costs?
  • Locality of reference
Loop Peeling

- Split first (or last) few iterations from loop and perform them separately
Loop Peeling

• Before

```c
for (int i = 0; i < n; ++i) {
    b[i] = (i == 0) ? a[i] : a[i] + b[i-1];
}
```

• After

```c
b[0] = a[0];
for (int i = 1; i < n; ++i) {
    b[i] = a[i] + b[i-1];
}
```

• What are the potential benefits? Costs?
Loop Tiling

- For nested loops, change iteration order
Loop Tiling

• Before

```c
for (i = 0; i < n; i++) {
    c[i] = 0;
    for (j = 0; j < n; j++) {
        c[i] = c[i] + a[i][j] * b[j];
    }
}
```

• After:

```c
for (i = 0; i < n; i += 4) {
    c[i] = 0;
    c[i + 1] = 0;
    for (j = 0; j < n; j += 4) {
        for (x = i; x < min(i + 4, n); x++) {
            for (y = j; y < min(j + 4, n); y++) {
                c[x] = c[x] + a[x][y] * b[y];
            }
        }
    }
}
```

• What are the potential benefits? Costs?
Loop Parallelization

• Before

```c
for (int i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];  // a, b, and c do not overlap
}
```

• After

```c
for (int i = 0; i < n % 4; ++i) a[i] = b[i] + c[i];
for (; i < n; i = i + 4) {
    __some4SIMDadd(a+i,b+i,c+i);
}
```

• What are the potential benefits? Costs?