CSE110A: Compilers

Topics:

- *Loop optimizations*

Loop optimizations

- Regional optimization
  - We can handle multiple basic blocks
  - but only if they fit a certain pattern

For loops

- How do they look in different languages
  - C/C++
  - Python
  - Numpy
- The more constrained the for loops are, the more assumptions the compiler can make, but less flexibility for the programmer

For loops

- The compiler can optimize For loops if they fit a certain pattern

- When developing a regional optimization, we start with strict constraints and then slowly relax them and make the optimization more general.
    - Sometimes it is not worth relaxing the constraints (optimization gets too complicated. Its not the compilers job to catch every pattern!)
    - If a programmer knows the pattern, then often you can write code such that the compiler can recognize the pattern and it will do better at optimizing!
    - Thus you can write more efficient code if you write it in such a way that the compiler can recognize patterns

For loops terminology

- Loop body:
  - A series of statements that are executed each loop iteration

- Loop condition:
  - the condition that decides whether the loop body is executed

- Iteration variable:
  - A variable that is updated exactly once during the loop
  - The loop condition depends on the iteration variable
  - The loop condition is only updated through the iteration variable

Examples

iteration variable
loop body
loop condition

```
for (int i = 0; i < 1024; i++) {
  counter += 1;
}
```

```
for (; i < 1024; i+=counter) {
  counter += 1;
}
```

```
while (1) {
  i++;
  counter += 1;
  if (i < 1024) {
    break;
  }
}
```

In general, is it possible to determine if an iteration variable exists or not?

Examples

What about these?

```
for (i = 0; i < 1024; i++) {
   counter += 1;
   foo();
 }
```

```
for (i = 0; i < j; i++) {
   counter += 1;
   j = rand();
 }
```

Loop unrolling

Loop unrolling

- Executing multiple instances of the loop body without checking the loop condition.

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

unrolled by a **factor** of 2

```
for (int i = 0; i < 128; i++) {
  // body
}
```

```
for (int i = 0; i < 128; i=i+1) {
  // body
  i=i+1
  // body
}
```

could we unroll more?

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
   // body
}
```

```
for (int i = 0; i < 128; i++) {
   // body
   i++
   // body
}
```

What can go wrong?

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Validate that we actually have an iteration variable

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Validate that we actually have an iteration variable

1. **find** candidate on lhs of assignment statement

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
   // body
}
```

Validate that we actually have an iteration variable

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i=i+1) {
   // body
}
```

Validate that we actually have an iteration variable

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
   // body
}
```

Validate that we actually have an iteration variable

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a literal

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
   // body
}
```

Validate that we actually have an iteration variable

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a literal
   (i.e. a compile time constant value, e.g. i<10*12)

Do these guarantee we will find an iteration variable?
What happens if we don't find one?

Loop unrolling conditions

- Several ways to unroll
  - More constraints: Simpler to unroll in code gen, more things to check
  - Less constraints: less things to check, harder to unroll in code gen

Base constraints (required for any unrolling):

Validate that we actually have an iteration variable

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a literal
     (i.e. a compile time constant value, e.g. i<10*12)

Loop unrolling conditions

- Simple unroll
  - Most constraints
  - Easiest code generation

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations (LI)
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + (update + body)*(F-1)
- perform codegen

Loop unrolling conditions

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

how to do these steps?

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations (LI)
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + update + body
- perform codegen

Loop unrolling conditions

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

result for a factor of 2

```
for (int i = 0; i < 128; i++) {
  // body
  i++
  // body
}
```

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations (LI)
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + (update + body)*(F-1)
- perform codegen

Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations (LI)
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + update + body
- perform codegen

Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

Actually this is fine as long as i is updated with a constant addition. but we need a more complicated formula to calculate LI:

```
ceil((end - start)/update)
```

But you may want to keep your life simpler by constraining it. We will keep it simple

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + update + body
- perform codegen

Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 4; i++) {
  // body
}
```

What if we try to unroll this by a factor of 3?

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + update + body
- perform codegen

Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 4; i++) {
   // body
}
```

What if we try to unroll this by a factor of 3?

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

How many times do we execute body?

For unroll factor F

Simple unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

Simple unroll code generation:

- create a new body = body + update + body
- perform codegen

Loop unrolling conditions

Let’s examine this a bit closer?

```
for (int i = 0; i < 4; i++) {
   // body
}
```

What if we try to unroll this by a factor of 3?

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

How many times do we execute body?

Loop unrolling conditions

Let’s examine this a bit closer?

```
for (int i = 0; i < 4; i++) {
   // body
}
```

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

What if we try to unroll this by a factor of 3?

How many times do we execute body?

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = 0; i < 3; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

```
for (int i = 3; i < 4; i++) {
   // body
}
```

Loop unrolling conditions

initially the loop starts the same as the original loop

```
for (int i = 0; i < 4; i++) {
   // body
}
```

find out how many unrolled loops we can execute:
(4 / 3) * 3 = 3
This gives us the first bound

second loop is initialized with the first bound

second loop’s bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

```
for (int i = ?; i < ?; i++) {
   // body
}
```

Loop unrolling conditions

What about in the general case? For unroll factor F?

```
for (int i = x; i < y; i++) {
   // body
}
```

find out how many unrolled loops we can execute:
?
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = x; i < y/F *F; i++) {
   // body
   i++
   ...
}
```

Note that y/F * F creates a remainder
i.e. the benefit of integer arithmetic

```
for (int i = y/F *F; i < y; i++) {
   // body
}
```

Loop unrolling conditions

- general unroll

For unroll factor F

General unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations, LI

General unroll code generation:

- Create simple unrolled loop with new bound: (LI/F)*F
- Create cleanup (basic) loop with initialization: (LI/F)*F
- perform codegen

None of these numbers have to be concrete!