Outline

• Simple loop transformations

• Loop invariants based transformations

• Induction variables based transformations

• Complex loop transformations
Simple loop transformations

Simple loop transformations are used to

• Increase performance/energy savings

and/or

• Unblock other transformations
  • E.g., increase the number of constant propagations
  • E.g., Extract thread-level parallelism from sequential code
  • E.g., Generate vector instructions
Loop unrolling

\[
\text{for (a=0; a < 4; a++)\{}
\text{... // Body}
\text{
\}}
\]

Unrolling factor: 2

\[
\text{%a = 0}
\]

\[
\text{%a=cmp %a, 4}
\]

\[
\text{branch %a}
\]

\[
\text{Body}
\]

\[
\text{%a=add %a, 1}
\]

\[
\text{%a = 0}
\]

\[
\text{%a=cmp %a, 4}
\]

\[
\text{branch %a}
\]

\[
\text{Body}
\]

\[
\text{%a=add %a, 2}
\]
Loop unrolling in LLVM: requirements

- The loop you want to unroll must be in LCSSA form
Loop unrolling in LLVM: dependences

```c
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<AssumptionCacheTracker>();
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.addRequired<LoopInfoWrapperPass>();
  AU.addRequired<ScalarEvolutionWrapperPass>();
  AU.addRequired<TargetTransformInfoWrapperPass>();
}
```
Loop unrolling in LLVM: headers

```c
#include "llvm/Analysis/OptimizationRemarkEmitter.h"
#include "llvm/IR/Dominators.h"
#include "llvm/Transforms/Utils/LoopUtils.h"
#include "llvm/Transforms/Utils/UnrollLoop.h"
#include "llvm/Analysis/AssumptionCache.h"
#include "llvm/Analysis/ScalarEvolution.h"
#include "llvm/Analysis/ScalarEvolutionExpressions.h"
#include "llvm/Analysis/TargetTransformInfo.h"
```
Loop unrolling in LLVM

Get the results of the required analyses

```cpp
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
const auto &TTI = getAnalysis<TargetTransformInfoWrapperPass>().getTTI(F);
```
Fetch a loop

```c
for (auto i : LI){
    auto loop = &*i;
    ...
}
```

```c
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<AssumptionCacheTracker>();
    AU.addRequired<DominatorTreeWrapperPass>();
    AU.addRequired<LoopInfoWrapperPass>();
    AU.addRequired<ScalarEvolutionWrapperPass>();
    return ;
}
```

```c
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
const auto &TTI = getAnalysis<TargetTransformInfoWrapperPass>().getTTI(F);
```
Loop unrolling in LLVM: API

```cpp
auto tripCount = SE.getSmallConstantTripCount(loop);
```

It is 0, or the number of iterations per invocation

```cpp
UnrollLoopOptions ULO,
ULO.Count = 2;
ULO.Force = false;
ULO.Runtime = false;
ULO.AllowExpensiveTripCount = true;
ULO.UnrollRemainder = false;
ULO.ForgetAllSCEV = true;
```

Unrolling factor

```cpp
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
const auto &TTI = getAnalysis<TargetTransformInfoWrapperPass>().getTTI(F);
```

Loop to unroll

```cpp
auto unrolled = UnrollLoop(
    loop, ULO,
    &LI, &SE, &DT, &AC, &TTI, &ORE, true);
```

Unrolling options
Loop unrolling in LLVM: result

```c
switch (unrolled){
    case LoopUnrollResult::FullyUnrolled :
        errs() << " Fully unrolled\n";
        return true ;
    case LoopUnrollResult::PartiallyUnrolled :
        errs() << " Partially unrolled\n";
        return true ;
    case LoopUnrollResult::Unmodified :
        errs() << " Not unrolled\n";
        break ;
    default:
        abort();
}
```

```c
auto unrolled = UnrollLoop(  
    loop, ULO,  
    &LI, &SE, &DT, &AC, &TTI, &ORE, true  
);
```
Loop unrolling in LLVM: example

```c
%2:     br label %4

%4:
%5 = phi i32 [ 0, %2 ], [ %8, %4 ]
%6 = phi i32 [ 0, %2 ], [ %7, %4 ]
%7 = tail call i32 @myF(i32 %6)
%8 = add nsw nsw i32 %5, 1
%9 = icmp eq i32 %8, 10
br i1 %9, label %3, label %4
```

```
%4:
%5 = phi i32 [ 0, %2 ], [ %10, %4 ]
%6 = phi i32 [ 0, %2 ], [ %9, %4 ]
%7 = tail call i32 @myF(i32 %6)
%8 = add nsw nsw i32 %5, 1
%9 = tail call i32 @myF(i32 %7)
%10 = add nsw nsw i32 %8, 1
%11 = icmp eq i32 %10, 10
br i1 %11, label %3, label %4
```

CFG for 'main' function

```
%3:     ret i32 %7

%3:     ret i32 % lessa
% lessa = phi i32 [ %9, %4 ]
```

CFG for 'main' function
Loop unrolling in LLVM: Demo

- **Detail**: LLVM_loops/README
- **Pass**: LLVM_loops/llvm/7
- **C program**: LLVM_loops/code/12
- **C program**: LLVM_loops/code/0
Loop unrolling in LLVM: example 2

There is still the same amount of loop overhead!
Loop unrolling in LLVM: the runtime checks

```c
UnrollLoopOptions ULO;
ULO.Count = 2;
ULO.Force = false;
ULO.Runtime = false;
ULO.AllowExpensiveTripCount = true;
ULO.UnrollRemainder = false;
ULO.ForgetAllSCEV = true;
```
Loop unrolling in LLVM: example 3

```c
int main (int argc, char *argv[]){
    auto r = 0;
    for (auto i=0; i < argc; i++){
        r = myF(r);
    }
    return r;
}
```

```
if (argc > 0)
    i = 0
    if (argc > 0)
        i++
    if (i == argc)
        return r
```

```
auto n=0
for (;n<i_mul; n+=4){
    Body
    Body
    Body
    Body
}
for(auto m=0;m<i_rest;m++){
    Body
}
```

```
body = i_mul
i_rest = i & 3
i_mul = i - i_rest
if (i_mul > 0)
    return r
```

Runtime checks

CFG for 'main' function
Loop unrolling in LLVM: API

```c++
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
const auto &TTI = getAnalysis<TargetTransformInfoWrapperPass>().getTTI(F);

auto unrolled = UnrollLoop(
  loop, ULO,
  &LI, &SE, &DT, &AC, &TTI, &ORE, true
);
```

OptimizationRemarkEmitter ORE(&F);

Normalize the generated loop to LCSSA
Loop peeling

%a = cmp %a, 10
branch %a

Body

%a = add %a, 1

Body

Peeling factor: 1

%a = cmp %a, %10
branch %a

Body

%a = add %a, 1
Loop peeling in LLVM

• API

```cpp
#include "llvm/Transforms/Utils/LoopPeel.h"

auto peeled = peelLoop(
    loop, peelingCount,
    &LI, &SE, &DT, &AC,
    true);
```

• No trip count
• No flags
• (almost) always possible
• To check if you can peel, invoke the following API: bool canPeel(Loop *loop)
Loop peeling in LLVM: example
Fetching analyses outputs from a module pass

• From a function pass

```c++
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache();
```

• From a module pass

```c++
auto& LI = getAnalysis<LoopInfoWrapperPass>(F).getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>(F).getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>(F).getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```
Outline

• Simple loop transformations

• Loop invariants based transformations

• Induction variables based transformations

• Complex loop transformations
Optimizations in small, hot loops

• Most programs: 90% of time is spent in few, small, hot loops
while (){ 
  statement 1
  statement 2
  statement 3
}
• Deleting a single statement from a small, hot loop might have a big impact
  (100 seconds -> 70 seconds)
Loop example

```c
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
3: do {
4:   a = 1;
5:   y = x + N;
6:   b = k + z;
7:   c = a * 3;
8:   if (N < 0){
9:     m = 5;
10:    break;
11: }
12: x++;  
13:} while (x < N);
```

- **Observation**: each statement in that loop will contribute to the program execution time.
- **Idea**: what about moving statements from inside a loop to outside it?
- Which statements can be moved outside our loop?
- How to identify them automatically? (code analysis)
- How to move them? (code transformation)
Hoisting code

• In order to “hoist” a loop-invariant computation out of a loop, we need a place to put it

• We could copy it to all immediate predecessors of the loop header...

```cpp
for (auto pBB : predecessors(H)){
  p = pBB->getTerminator();
  inv->moveBefore(p);
}
```

Is it correct?

• ...But we can avoid code duplication (and bugs) by taking advantage of loop normalization that guarantees the existence of the pre-header
Hoisting code

• In order to “hoist” a loop-invariant computation out of a loop, we need a place to put it

• We could copy it to all immediate predecessors of the loop header...

```c
pBB = loop->getLoopPreheader();
p = pBB->getTerminator();
inv->moveBefore(p);
```

• ...but we can avoid code duplication (and bugs) by taking advantage of loop normalization that guarantees the existence of the pre-header
Can we hoist all invariant instructions of a loop L in the pre-header of L?

```c
for (inv : invariants(loop)) {
  pBB = loop->getLoopPreheader();
  p = pBB->getTerminator();
  inv->moveBefore(p);
}
```
Hoisting conditions

- For a loop-invariant definition

(d) \( t = x \text{ op } y \)

- Assuming no SSA, we can hoist \( d \) into the loop’s pre-header if
  1. \( d \) dominates all loop exits at which \( t \) is live-out, and
  2. there is only one definition of \( t \) in the loop, and
  3. \( t \) is not live-out of the pre-header

\[
\begin{align*}
L0: & \quad t = 0 \\
L1: & \quad i = i + 1 \\
& \quad t = a \times b \\
& \quad M[i] = t \\
& \quad \text{if } i < N \text{ goto } L1 \\
L2: & \quad x = t \\
L0: & \quad t = 0 \\
L1: & \quad i = i + 1 \\
& \quad t = a \times b \\
& \quad M[i] = t \\
& \quad \text{goto } L1 \\
L2: & \quad x = t \\
L0: & \quad t = 0 \\
L1: & \quad i = i + 1 \\
& \quad t = a \times b \\
& \quad M[i] = t \\
& \quad t = 0 \\
& \quad M[j] = t \\
& \quad \text{if } i < N \text{ goto } L1 \\
L2: & \quad x = t
\end{align*}
\]
Hoisting conditions

• For a loop-invariant definition
  (d) \( t = x \text{ op } y \)

• Assuming SSA, we can hoist d into the loop’s pre-header if
  1. d dominates all loop exits at which t is live-out, and
  2. there is only one definition of t in the loop, and
  3. t is not live-out of the pre-header
Outline

• Simple loop transformations

• Loop invariants based transformations

• Induction variables based transformations

• Complex loop transformations
Loop example

1: if (N>5) { k = 1; z = 4; }
2: else { k = 2; z = 3; }
3: do {
4:   a = 1;
5:   y = x + N;
6:   b = k + z;
7:   c = a * 3;
8:   if (N < 0) {
9:     m = 5;
10:    break;
11:  }
12:  x++;  
13: } while (x < N);

Assuming a, b, c, m are used after our code

Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?
Loop example

1: if (N > 5) { k = 1; z = 4; } 
2: else { k = 2; z = 3; } 

do 

3:   a = 1; 
4:   y = x + N; 
5:   b = k + z; 
6:   c = a * 3; 
7:   if (N < 0) { 
8:     m = 5; 
9:     break; 
}
10:  x++; 
11: } while (x < N);

Do we have to execute 4 for every iteration? 

Compute manually values of x and y for every iteration 

What do you see?

Do we have to execute 10 for every iteration?
Loop example

1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
3: do {
4:   a = 1;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
10:   }
11: x++;y++;
12: } while (x < N);

Do we have to execute 10 for every iteration?

Do we have to execute 4 for every iteration?
Loop example

1: if (N>5) { k = 1; z = 4; }
2: else { k = 2; z = 3; }

3: do {
4:   a = 1;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0) {
8:     m = 5;
9:     break;
10:   }
11:   x++; y++
12: } while (y < (2*N));

Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?
Loop example

1: if (N>5) { k = 1; z = 4; }
2: else { k = 2; z = 3; }
do {
3:   a = 1;
4:   b = k + z;
5:   c = a * 3;
6:   if (N < 0) {
7:     m = 5;
8:     break;
9:   }
10:  y++;
11:} while (y < (2*N));

Do we have to execute 10 for every iteration?
Do we have to execute 4 for every iteration?
Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?

x, y are induction variables
Is the code transformation worth it?

1: if (N>5) { k = 1; z = 4;}
2: else { k = 2; z = 3;}

A : y = N; tmp = 2*N;

```
1: if (N>5) { k = 1; z = 4;}
2: else { k = 2; z = 3;}

1: if (N>5) { k = 1; z = 4;}
2: else { k = 2; z = 3;}
```

```
do {
3:   a = 1;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0) {
8:     m = 5;
9:     break;
}
10:  y++;
11:} while (y < tmp);

3: a = 1;
4: y = x + N;
5: b = k + z;
6: c = a * 3;
7: if (N < 0) {
8: m = 5;
9: break;
}
10: x++;
11:} while (x < N);
```

**Induction variable elimination**
... and after Loop Invariant Code Motion ...

1. if (N>5) { k = 1; z = 4; }
2. else { k = 2; z = 3; }

A : y = N; tmp = 2 * N;
3. a = 1;
5. b = k + z;
6. c = a * 3;

d{ 
7. if (N < 0) {
8. m = 5;
9. break;
}
10. y ++;
11. } while (y < tmp);

1. if (N>5) { k = 1; z = 4; }
2. else { k = 2; z = 3; }

d {
3. a = 1;
4. y = x + N;
5. b = k + z;
6. c = a * 3;
7. if (N < 0) {
8. m = 5;
9. break;
}
10. x ++;
11. } while (x < N);
... and with a better Loop Invariant Code Motion ...

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
A : y = N; tmp = 2*N;
3 : a = 1;
5 : b = k + z;
6: c = a * 3;
7: if (N < 0){
8:    m = 5;
9:    break;
}
10: x++; 
11:} while (x < N);
```
... and after dead code elimination ...

1: if (N>5) {k = 1; z = 4;}  
2: else {k = 2; z = 3;}

3: a=1;  
5: b=k+z;  
6: c=a*3;  
7: if (N < 0) {
8:   m=5;  
}  

Assuming a,b,c,m are used after our code

1: if (N>5) {k = 1; z = 4;}  
2: else {k = 2; z = 3;}

3: a = 1;  
4: y = x + N;  
5: b = k + z;  
6: c = a * 3;  
7: if (N < 0) {
8:   m = 5;  
9:   break;  
}  
10: x++;  
11: } while (x < N);
Induction variable elimination

• Suppose we have a loop variable
  • \(i\) initially set to \(i_0\); each iteration \(i = i + 1\)

• and a variable that linearly depends on it
  • \(x = i \cdot c_1 + c_2\)

• We can
  • Initialize \(x = i_0 \cdot c_1 + c_2\)
  • Increment \(x\) by \(c_1\) each iteration
Is it faster?

On some hardware, adds are faster than multiplies
  • Strength reduction
Induction variable elimination: step 1

Run induction variable identification

① Iterate over IVs
   k = j * c1 + c2
   • where the IV j = (i, a, b), and
   • this is the only def of k in the loop, and
   • there is no def of i between the def of j and the def of k

② Record as k = (i, a*c1, b*c1+c2)
Induction variable elimination: step 2

For an induction variable $k = (i, c1, c2)$

① Initialize $k = i \times c1 + c2$ in the pre-header

② Replace $k$’s def in the loop by $k = k + c1$
   • Make sure to do this after $i$’s definition
Outline

• Simple loop transformations

• Loop invariants based transformations

• Induction variables based transformations

• Complex loop transformations
Loop transformations

• Restructure a loop to expose more optimization opportunities and/or transform the “loop overhead”
  • Loop unrolling, loop peeling, ...

• Reorganize a loop to improve memory utilization
  • Cache blocking, skewing, loop reversal

• Distribute a loop over cores/processors
  • DOACROSS, DOALL, DSWP, HELIX
• How many clock cycles will it take?

...  

`varX = array[5];`

...
• How many clock cycles will it take?
Loop transformations for memory optimizations

• How many clock cycles will it take?

... varX = array[5];
...

![Diagram of memory hierarchy]
Goal: improve cache performance

• **Temporal locality**
  A resource that has just been referenced will more likely be referenced again in the near future

• **Spatial locality**
  The likelihood of referencing a resource is higher if a resource near it was just referenced

• Ideally, a compiler generates code with high temporal and spatial locality for the target architecture
  • What to minimize: bad replacement decisions
What a compiler can do

• Time:
  • When is an object accessed?

• Space:
  • Where does an object exist in the address space?
  • What is the data layout of an object in memory?

• These are the two “knobs” a compiler can manipulate
First understand cache behavior ...

• When do cache misses occur?
  • Use locality analysis

• Can we change the visitation order to produce better behavior?
  • Evaluate costs

• Does the new visitation order still produce correct results?
  • Use dependence analysis
... and then rely on loop transformations

- loop interchange
- cache blocking
- loop fusion
- loop reversal
- ...

...
Code example

double A[N][N], B[N][N];
...
for i = 0 to N-1{
    for j = 0 to N-1{
        ... = A[i][j] ...
    }
}

Iteration space for A
Loop interchange

for i = 0 to N-1
for j = 0 to N-1
... = A[j][i] ...

Assumptions: N is large; A is row-major; 2 elements per cache line

For j = 0 to N-1
for i = 0 to N-1
... = A[j][i] ...

A[][] in C? Java?
Java (similar in C)

To create a matrix:

```java
double [][] A = new double[3][3];
```

A is an array of arrays
A is not a 2 dimensional array!
Java (similar in C)

To create a matrix:

double [][] A = new double[3][];
A[0] = new double[3];
Java (similar in C)

To create a matrix:
```java
double [][] A = new double[3][];
A[0] = new double[10];
A[1] = new double[5];
```

A is a jagged array
C#: [][] vs. [,]

double [][] A = new double[3][];
A[0] = new double[3];

double [,] A = new double[3,3];

The compiler can easily choose between raw-major vs. column-major
```c
#include <stdio.h>

int main (){
    int a[2][4];

    printf("0x%p\n", &a[0][0]);
    printf("0x%p\n", &a[0][1]);
    printf(" Distance: %d bytes\n", ((unsigned int)(&a[0][1])) - ((unsigned int)(&a[0][0])));

    printf("0x%p\n", &a[0][0]);
    printf("0x%p\n", &a[1][0]);
    printf(" Distance: %d bytes\n", ((unsigned int)(&a[1][0])) - ((unsigned int)(&a[0][0])));

    return 0;
}
```
Cache blocking (a.k.a. tiling)

for $i = 0$ to $N-1$
  for $j = 0$ to $N-1$
    $f(A[i], A[j])$

for $JJ = 0$ to $N-1$ by $B$
  for $i = 0$ to $N-1$
    for $j = JJ$ to $\min(N-1, JJ+B-1)$
      $f(A[i], A[j])$
Loop fusion

for $i = 0$ to $N-1$

$C[i] = A[i] \times 2 + B[i]$

for $i = 0$ to $N-1$

$D[i] = A[i] \times 2$

• Reduce loop overhead
• Improve locality by combining loops that reference the same array
• Increase the granularity of work done in a loop
Loop transformations

• They manipulate the order of memory accesses

• They can change both temporal and spatial localities

• They can enable or disable parallelism