

CSE P 501 – Compilers

Loops

Hal Perkins

Autumn 2019

Agenda

- Loop optimizations
 - Dominators – discovering loops
 - Loop invariant calculations
 - Loop transformations
- A quick look at some memory hierarchy issues
- Largely based on material in Appel ch. 18, 21; similar material in other books

Loops

Much of the execution time of programs is spent inside loops

∴ worth considerable effort to make loops go faster

∴ want to figure out how to recognize loops and figure out how to “improve” them

What's a Loop?

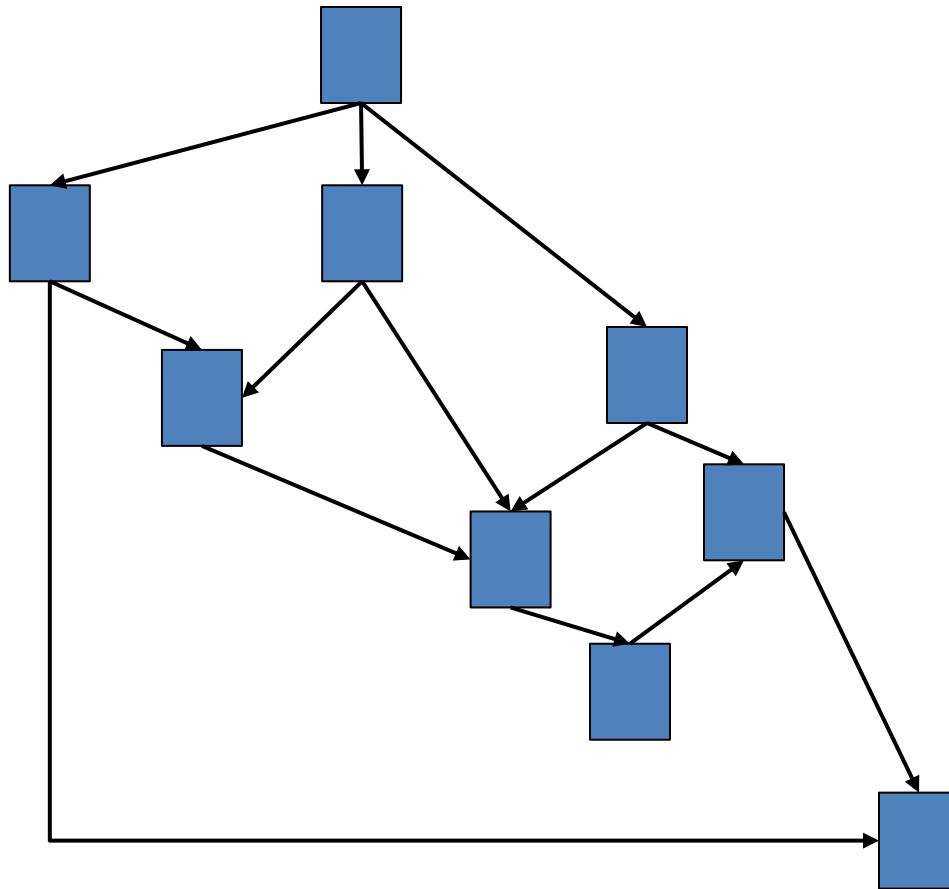
- In source code, a loop is the set of statements in the body of a **for/while** construct
- But, in a language that permits free use of **GOTOs**, how do we recognize a loop?
- In a control-flow-graph (node = basic-block, arc = flow-of-control), how do we recognize a loop?

Any Loops in this Code?

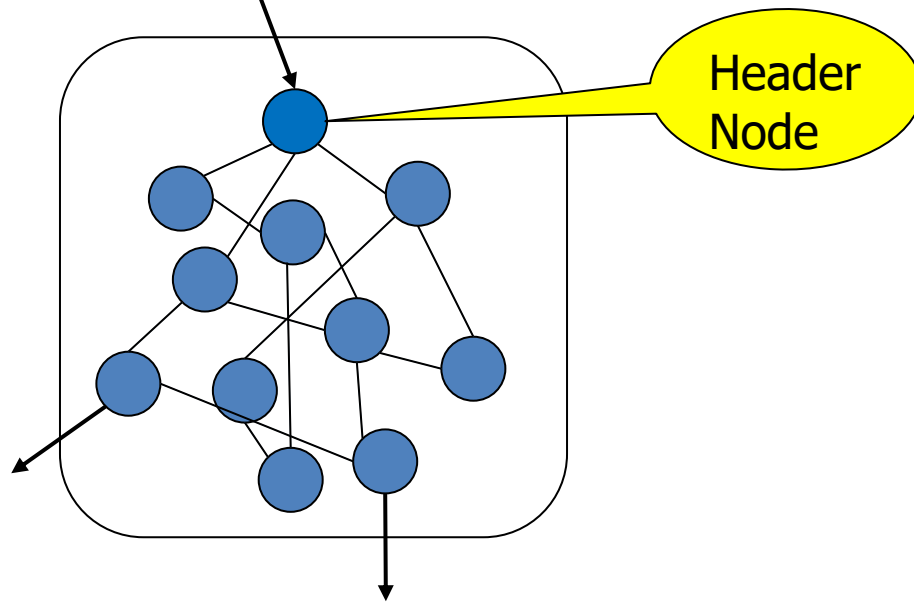
```
        i = 0
        goto L8
L7:      i++
L8:      if (i < N) goto L9
        s = 0
        j = 0
        goto L5
L4:      j++
L5:      N--
        if(j >= N) goto L3
        if (a[j+1] >= a[j]) goto L2
        t = a[j+1]
        a[j+1] = a[j]
        a[j] = t
        s = 1
L2:      goto L4
L3:      if(s != 0) goto L1 else goto L9
L1:      goto L7
L9:      return
```

Anyone recognize or
guess the algorithm?

Any Loops in this Flowgraph?



Loop in a Flowgraph: Intuition



- Cluster of nodes, such that:
- There's one node called the "header"
- I can reach all nodes in the cluster from the header
- I can get back to the header from all nodes in the cluster
- Only once entrance - via the header
- One or more exits

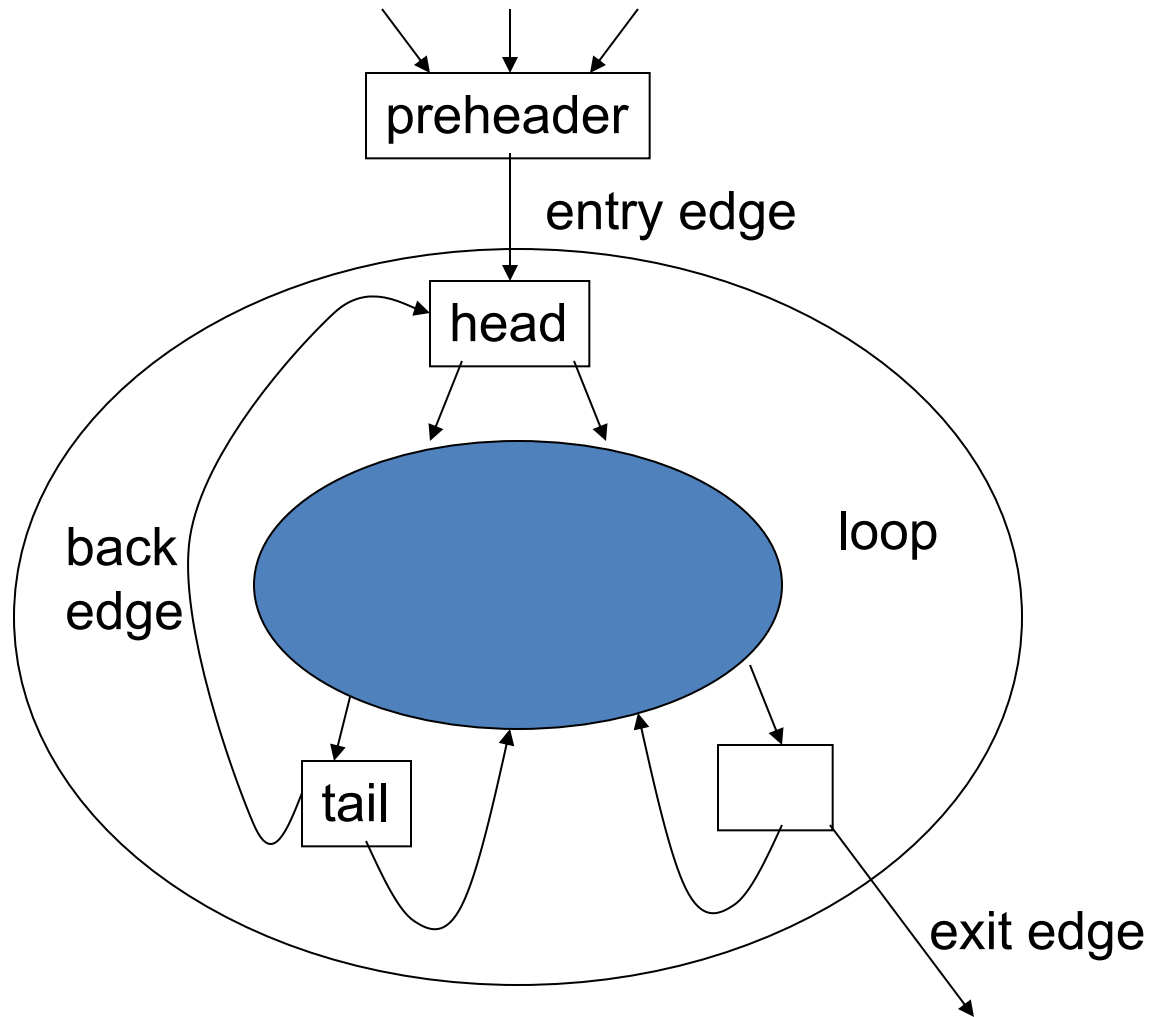
What's a Loop? (recap)

- In a control flow graph, a loop is a set of nodes S such that:
 - S includes a *header node* h
 - From any node in S there is a path of directed edges leading to h
 - There is a path from h to any node in S
 - There is no edge from any node outside S to any node in S other than h

Entries and Exits

- In a loop
 - An *entry node* is one with some predecessor outside the loop
 - An *exit node* is one that has a successor outside the loop
- Corollary: A loop may have multiple exit nodes, but only one entry node

Loop Terminology



Finding Loops in Flow Graphs

- We use *dominators* for this
- Recall:
 - Every control flow graph has a unique start node s_0
 - Node x dominates node y if every path from s_0 to y must go through x
 - A node x dominates itself

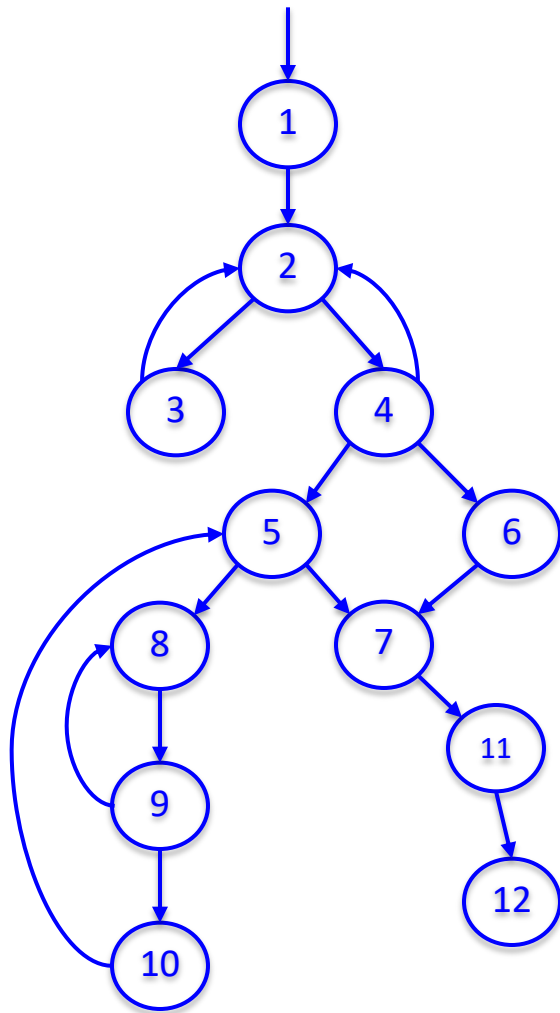
Calculating Dominator Sets

- $D[n]$ is the set of nodes that dominate n
 - $D[s_0] = \{ s_0 \}$
 - $D[n] = \{ n \} \cup (\bigcap_{p \in \text{pred}[n]} D[p])$
- Set up an iterative analysis as usual to solve this
 - Except initially each $D[n]$ must be all nodes in the graph – updates make these sets smaller if changed

Immediate Dominators

- Every node n has a single *immediate dominator* $\text{idom}(n)$
 - $\text{idom}(n)$ dominates n
 - $\text{idom}(n)$ differs from n – i.e., strictly dominates
 - $\text{idom}(n)$ does not dominate any other strict dominator of n
 - i.e., strictly dominates and is nearest dominator
- Fact (er, theorem): If a dominates n and b dominates n , then either a dominates b or b dominates a
 - $\therefore \text{idom}(n)$ is unique

Example



Dominators		
n	DOM	IDOM
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

Dominator Tree

- A *dominator tree* is constructed from a flowgraph by drawing an edge from every node n to $\text{idom}(n)$
 - This will be a tree. Why?

Back Edges & Loops

- A flow graph edge from a node n to a node h that dominates n is a *back edge*
- For every back edge there is a corresponding subgraph of the flow graph that is a loop

Natural Loops

- If h dominates n and $n \rightarrow h$ is a back edge, then the *natural loop* of that back edge is the set of nodes x such that
 - h dominates x
 - There is a path from x to n not containing h
- h is the *header* of this loop
- Standard loop optimizations can cope with loops whether they are natural or not

Inner Loops

- Inner loops are more important for optimization because most execution time is expected to be spent there
- If two loops share a header, it is hard to tell which one is “inner”
 - Common way to handle this is to merge natural loops with the same header
 - Resulting loop could well not be a “natural loop”

Inner (nested) loops

- Suppose
 - A and B are loops with headers a and b
 - $a \neq b$
 - b is inside A
- Then
 - The nodes of B are a proper subset of A
 - B is nested in A, or B is the *inner loop*

Loop-Nest Tree

- Given a flow graph G
 1. Compute the dominators of G
 2. Construct the dominator tree
 3. Find the natural loops (thus all loop-header nodes)
 4. For each loop header h , merge all natural loops of h into a single loop: $\text{loop}[h]$
 5. Construct a tree of loop headers s.t. h_1 is above h_2 if h_2 is in $\text{loop}[h_1]$

Loop-Nest Tree details

- Leaves of this tree are the innermost loops
- Need to put all non-loop nodes somewhere
 - Convention: lump these into the root of the loop-nest tree

Loop Preheader

- Often we need a place to park code right before the beginning of a loop
- Easy if there is a single node preceding the loop header h
 - But this isn't the case in general
- So insert a *preheader* node p
 - Include an edge $p \rightarrow h$
 - Change all edges $x \rightarrow h$ to be $x \rightarrow p$

Loop-Invariant Computations

- Idea: If $x := a1 \text{ op } a2$ always does the same thing each time around the loop, we'd like to *hoist* it and do it once outside the loop
- But can't always tell if $a1$ and $a2$ will have the same value
 - Need a conservative (safe) approximation

Loop-Invariant Computations

- $d: x := a_1 \text{ op } a_2$ is *loop-invariant* if for each a_i
 - a_i is a constant, or
 - All the definitions of a_i that reach d are outside the loop, or
 - Only one definition of a_i reaches d , and that definition is loop invariant
- Use this to build an iterative algorithm
 - Base cases: constants and operands defined outside the loop
 - Then: repeatedly find definitions with loop-invariant operands

Hoisting

- Assume that $d: x := a1 \text{ op } a2$ is loop invariant. We can hoist it to the loop preheader if
 - d dominates all loop exits where x is live-out, and
 - There is only one definition of x in the loop, and
 - x is not live-out of the loop preheader
- Need to modify this if $a1 \text{ op } a2$ could have side effects or raise an exception

Hoisting: Possible?

- Example 1

L0: $t := 0$

L1: $i := i + 1$

d: $t := a \text{ op } b$

$M[i] := t$

 if $i < n$ goto L1

L2: $x := t$

- Example 2

L0: $t := 0$

L1: if $i \geq n$ goto L2

$i := i + 1$

d: $t := a \text{ op } b$

$M[i] := t$

 goto L1

L2: $x := t$

Hoisting: Possible?

- Example 3

L0: $t := 0$

L1: $i := i + 1$

d: $t := a \text{ op } b$

$M[i] := t$

$t := 0$

$M[j] := t$

if $i < n$ goto L1

L2: $x := t$

- Example 4

L0: $t := 0$

L1: $M[j] := t$

$i := i + 1$

d: $t := a \text{ op } b$

$M[i] := t$

if $i < n$ goto L1

L2: $x := t$

Induction Variables

- Suppose inside a loop
 - Variable i is incremented or decremented
 - Variable j is set to $i*c+d$ where c and d are loop-invariant
- Then we can calculate j 's value without using i
 - Whenever i is incremented by a , increment j by $a*c$

Example

- Original

s := 0

i := 0

L1: if $i \geq n$ goto L2

j := $i * 4$

k := $j + a$

x := $M[k]$

s := $s + x$

i := $i + 1$

goto L1

L2:

- To optimize, do...

- Induction-variable analysis to discover i and j are related induction variables
- Strength reduction to replace $*4$ with an addition
- Induction-variable elimination to replace $i \geq n$
- Assorted copy propagation

Result

- Original

s := 0

i := 0

L1: if $i \geq n$ goto L2

j := $i * 4$

k := j+a

x := M[k]

s := s+x

i := i+1

goto L1

L2:

- Transformed

s := 0

k' = a

b = $n * 4$

c = a+b

L1: if $k' \geq c$ goto L2

x := M[k']

s := s+x

k' := k'+4

goto L1

L2:

Details are somewhat messy – see your favorite compiler book

Basic and Derived Induction Variables

- Variable i is a *basic induction variable* in loop L with header h if the only definitions of i in L have the form $i := i \pm c$ where c is loop invariant
- Variable k is a *derived induction variable* in L if:
 - There is only one definition of k in L of the form $k := j * c$ or $k := j + d$ where j is an induction variable and c, d are loop-invariant, *and*
 - if j is a derived variable in the family of 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

Optimizing Induction Variables

- Strength reduction: if a derived induction variable is defined with $j := i * c$, try to replace it with an addition inside the loop
- Elimination: after strength reduction some induction variables are not used or are only compared to loop-invariant variables; delete them
- Rewrite comparisons: If a variable is used only in comparisons against loop-invariant variables and in its own definition, modify the comparison to use a related induction variable

Loop Unrolling

- If the body of a loop is small, much of the time is spent in the “increment and test” code
- Idea: reduce overhead by *unrolling* – put two or more copies of the loop body inside the loop

Loop Unrolling

- Basic idea: Given loop L with header node h and back edges $s_i \rightarrow h$
 1. Copy the nodes to make loop L' with header h' and back edges $s_i' \rightarrow h'$
 2. Change all back edges in L from $s_i \rightarrow h$ to $s_i \rightarrow h'$
 3. Change all back edges in L' from $s_i' \rightarrow h'$ to $s_i' \rightarrow h$

Unrolling Algorithm Results

- Before

L1: $x := M[i]$

$s := s + x$

$i := i + 4$

if $i < n$ goto L1 else L2

L2:

- After

L1: $x := M[i]$

$s := s + x$

$i := i + 4$

if $i < n$ goto L1' else L2

L1': $x := M[i]$

$s := s + x$

$i := i + 4$

if $i < n$ goto L1 else L2

L2:

Hmmmm....

- Not so great – just code bloat
- But: use induction variables and various loop transformations to clean up

After Some Optimizations

- Before

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i < n goto L1' else L2
L1': x := M[i]
    s := s + x
    i := i + 4
    if i < n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i < n goto L1 else L2
L2:
```

Still Broken...

- But in a different, better(?) way
- Good code, but only correct if original number of loop iterations was even
- Fix: add an epilogue to handle the “odd” leftover iteration

Fixed

- Before

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i < n goto L1 else L2
L2:
```

- After

```
    if i < n - 8 goto L1 else L2
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i < n - 8 goto L1 else L2
L2: x := M[i]
    s := s + x
    i := i + 4
    if i < n goto L2 else L3
L3:
```

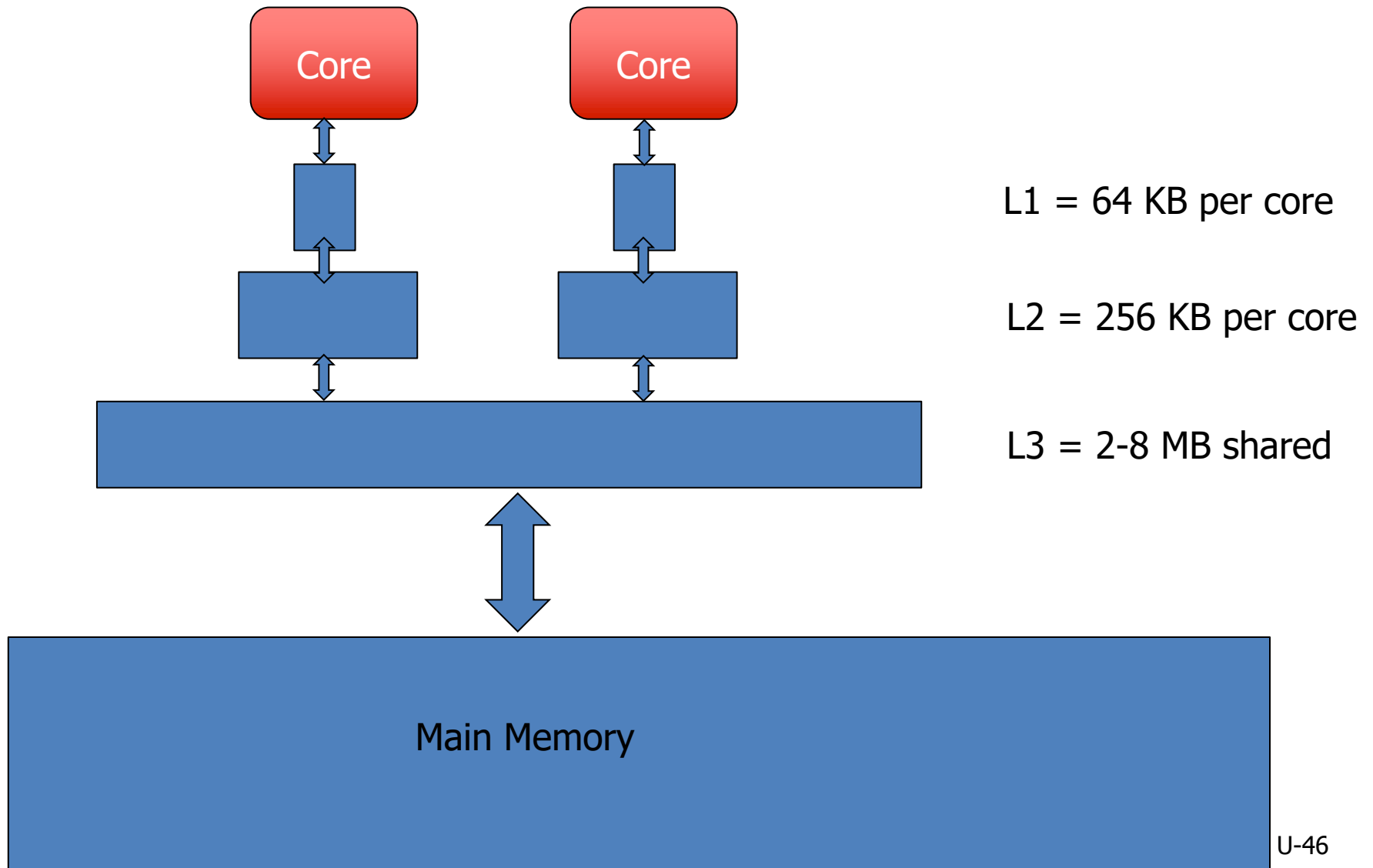
Postscript

- This example only unrolls the loop by a factor of 2
- More typically, unroll by a factor of K
 - Then need an epilogue that is a loop like the original that iterates up to $K-1$ times

Memory Heirarchies

- One of the great triumphs of computer design
- Effect is a large, fast memory
- Reality is a series of progressively larger, slower, cheaper stores, with frequently accessed data automatically staged to faster storage (cache, main storage, disk)
- Programmer/compiler typically treats it as one large store. (but not always the best idea)
- Hardware maintains cache coherency – most of the time

Intel Haswell Caches



Just How Slow *is* Operand Access?

- Instruction ~5 per cycle
- Register 1 cycle
- L1 CACHE ~4 cycles
- L2 CACHE ~10 cycles
- L3 CACHE (unshared line) ~40 cycles
- DRAM ~100 ns

Implications

- CPU speed increases have out-paced increases in memory access times
- Memory access now often determines overall execution speed
- “Instruction count” is not the only performance metric for optimization

Memory Issues

- Byte load/store is often slower than whole (physical) word load/store
 - Unaligned access is often extremely slow
- **Temporal locality**: accesses to recently accessed data will usually find it in the (fast) cache
- **Spatial locality**: accesses to data near recently used data will usually be fast
 - “near” = in the same cache block
- But – alternating accesses to blocks that map to the same cache block will cause thrashing

Data Alignment

- Data objects (structs) often are similar in size to a cache block (≈ 64 bytes)
 - ∴ Better if objects don't span blocks
- Some strategies
 - Allocate objects sequentially; bump to next block boundary if useful
 - Allocate objects of same common size in separate pools (all size-2, size-4, etc.)
- Tradeoff: speed for some wasted space

Instruction Alignment

- Align frequently executed basic blocks on cache boundaries (or avoid spanning cache blocks)
- Branch targets (particularly loops) may be faster if they start on a cache line boundary
 - Often see multi-byte nops in optimized code as padding to align loop headers
 - How much depends on architecture (typical 16 or 32 bytes)
- Try to move infrequent code (startup, exceptions) away from hot code
- Optimizing compiler can perform basic-block ordering

Loop Interchange

- Watch for bad cache patterns in inner loops; rearrange if possible

- Example

```
for (i = 0; i < m; i++)
```

```
  for (j = 0; j < n; j++)
```

```
    for (k = 0; k < p; k++)
```

```
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

- $b[i,j+1,k]$ is reused in the next two iterations, but will have been flushed from the cache by the k loop

Loop Interchange

- Solution for this example: interchange j and k loops

```
for (i = 0; i < m; i++)
```

```
  for (k = 0; k < p; k++)
```

```
    for (j = 0; j < n; j++)
```

```
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

- Now $b[i,j+1,k]$ will be used three times on each cache load
- Safe here because loop iterations are independent

Loop Interchange

- Need to construct a data-dependency graph showing information flow between loop iterations
- For example, iteration (j,k) depends on iteration (j',k') if (j',k') computes values used in (j,k) or stores values overwritten by (j,k)
 - If there is a dependency and loops are interchanged, we could get different results – so can't do it

Blocking

- Consider matrix multiply

```
for (i = 0; i < n; i++)  
  for (j = 0; j < n; j++) {  
    c[i,j] = 0.0;  
    for (k = 0; k < n; k++)  
      c[i,j] = c[i,j] + a[i,k]*b[k,j]  
  }
```

- If a, b fit in the cache together, great!
- If they don't, then every b[k,j] reference will be a cache miss
- Loop interchange (i<->j) won't help; then every a[i,k] reference would be a miss

Blocking

- Solution: reuse rows of A and columns of B while they are still in the cache
- Assume the cache can hold $2 \cdot c \cdot n$ matrix elements ($1 < c < n$)
- Calculate $c \times c$ blocks of C using c rows of A and c columns of B

Blocking

- Calculating $c \times c$ blocks of C
for ($i = i_0; i < i_0+c; i++$)
 for ($j = j_0; j < j_0+c; j++$) {
 $c[i,j] = 0.0;$
 for ($k = 0; k < n; k++$)
 $c[i,j] = c[i,j] + a[i,k]*b[k,j]$
 }
}

Blocking

- Then nest this inside loops that calculate successive $c \times c$ blocks

```
for (i0 = 0; i0 < n; i0+=c)
  for (j0 = 0; j0 < n; j0+=c)
    for (i = i0; i < i0+c; i++)
      for (j = j0; j < j0+c; j++) {
        c[i,j] = 0.0;
        for (k = 0; k < n; k++)
          c[i,j] = c[i,j] + a[i,k]*b[k,j]
      }
}
```

Parallelizing Code

- There is a large literature about how to rearrange loops for better locality and to detect parallelism
- Some starting points
 - Latest edition of *Dragon book*, ch. 11
 - Allen & Kennedy *Optimizing Compilers for Modern Architectures*
 - Wolfe, *High-Performance Compilers for Parallel Computing*