CSC D70: Compiler Optimization

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
CSC D70: Compiler Optimization
Introduction, Logistics

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
Summary

• Syllabus
  – Course Introduction, Logistics, Grading

• Information Sheet
  – Getting to know each other

• Assignments

• Learning LLVM

• Compiler Basics
Syllabus: Who Are We?
Gennady (Gena) Pekhimenko

Assistant Professor, Instructor
pekhimenko@cs.toronto.edu
http://www.cs.toronto.edu/~pekhimenko/
Office: BA 5232 / IC 454
PhD from Carnegie Mellon
Worked at Microsoft Research, NVIDIA, IBM
Research interests: computer architecture, systems, machine learning, compilers, hardware acceleration

Vector Institute
EcoSystem Group
Bojian Zheng

PhD Student, TA
bojian@cs.toronto.edu

Office: BA 5214 D02
BSc. from UofT ECE
Research interests: computer architecture, GPUs, machine learning

Vector Institute
EcoSystem Group
Course Information: Where to Get?

• Course Website:  
  http://www.cs.toronto.edu/~pekhimenko/courses/cscd70-w20/  
  – Announcements, Syllabus, Course Info, Lecture Notes, Tutorial Notes, Assignments

• Piazza:  
  https://piazza.com/utoronto.ca/winter2020/cscd70/home  
  – Questions/Discussions, Syllabus, Announcements

• Quercus  
  – Emails/announcements

• Your email
Useful Textbook

Compilers
Principles, Techniques, & Tools
Second Edition

Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman
CSC D70: Compiler Optimization
Compiler Introduction

Prof. Gennady Pekhinmenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
Why Computing Matters (So Much)?
WHAT IS THE DIFFERENCE BETWEEN THE COMPUTING INDUSTRY AND THE PAPER TOWEL INDUSTRY?
Industry of replacement

1971  2020
CAN WE CONTINUE BEING AN INDUSTRY OF NEW POSSIBILITIES?

Personalized healthcare  Virtual reality  Real-time translators
Moore’s Law
Or, how we became an industry of new possibilities

Every 2 Years

- Double the number of transistors
- Build higher performance general-purpose processors
  - Make the transistors available to masses
  - Increase performance (1.8×↑)
  - Lower the cost of computing (1.8×↓)
What is the catch?
Powering the transistors without melting the chip
Looking back

Evolution of processors

Single-core Era

1971

740 KHz

Multicore Era

2004

Dennard scaling broke

2003

3.4 GHz

2013

3.5 GHz
Any Solution Moving Forward?

Hardware accelerators:

- GPUs (Graphics Processing Units)
- FPGAs (Field Programmable Gate Arrays)
- TPUs (Tensor Processing Units)
Heterogeneity and Specialization

- CPUs
- Low Power CPUs
- GPU
- FPGAs
- Custom Logic
- OpenCL
- Halide
- Catapult
- Hadoop
- Java
- GraphLab
- Catapult
- CUDA
- OpenCL
Programmability versus Efficiency

We need compilers!
Introduction to Compilers

• What would you get out of this course?
• Structure of a Compiler
• Optimization Example
What Do Compilers Do?

1. Translate one language into another
   - e.g., convert C++ into x86 object code
   - difficult for “natural” languages, but feasible for computer languages

2. Improve (i.e. “optimize”) the code
   - e.g., make the code run 3 times faster
     • or more energy efficient, more robust, etc.
   - driving force behind modern processor design
How Can the Compiler Improve Performance?

Execution time = Operation count * Machine cycles per operation

- **Minimize the number of operations**
  - arithmetic operations, memory accesses
- **Replace expensive operations with simpler ones**
  - e.g., replace 4-cycle multiplication with 1-cycle shift
- **Minimize cache misses**
  - both data and instruction accesses
- **Perform work in parallel**
  - instruction scheduling within a thread
  - parallel execution across multiple threads
What Would You Get Out of This Course?

• Basic knowledge of existing compiler optimizations

• Hands-on experience in constructing optimizations within a fully functional research compiler

• Basic principles and theory for the development of new optimizations
Structure of a Compiler

- Optimizations are performed on an “intermediate form”
  - similar to a generic RISC instruction set
- Allows easy portability to multiple source languages, target machines
Ingredients in a Compiler Optimization

• **Formulate optimization problem**
  – Identify opportunities of optimization
    • applicable across many programs
    • affect key parts of the program (loops/recursions)
    • amenable to “efficient enough” algorithm

• **Representation**
  – Must abstract essential details relevant to optimization
Ingredients in a Compiler Optimization
Ingredients in a Compiler Optimization

• **Formulate optimization problem**
  – Identify opportunities of optimization
    • applicable across many programs
    • affect key parts of the program (loops/recursions)
    • amenable to “efficient enough” algorithm

• **Representation**
  – Must abstract essential details relevant to optimization

• **Analysis**
  – Detect when it is desirable and safe to apply transformation

• **Code Transformation**

• **Experimental Evaluation** (and repeat process)
Representation: Instructions

• Three-address code
  \[ A := B \text{ op } C \]
  - LHS: name of variable e.g. \( x \), \( A[t] \) (address of \( A \) + contents of \( t \))
  - RHS: value

• Typical instructions
  \[ A := B \text{ op } C \]
  \[ A := \text{unaryop } B \]
  \[ A := B \]
  GOTO \( s \)
  IF \( A \) relop \( B \) GOTO \( s \)
  CALL \( f \)
  RETURN
Optimization Example

- **Bubblesort** program that sorts an array $A$ that is allocated in static storage:
  - an element of $A$ requires four bytes of a byte-addressed machine
  - elements of $A$ are numbered 1 through $n$ ($n$ is a variable)
  - $A[j]$ is in location $\&A+4*(j-1)$

```pascal
FOR i := n-1 DOWNTO 1 DO
  FOR j := 1 TO i DO
      temp := A[j];
      A[j] := A[j+1];
      A[j+1] := temp
    END
```
Translated Code

\[
i := n-1
\]

S5: if i<1 goto s1

\[
j := 1
\]

s4: if j>i goto s2

\[
t1 := j-1
t2 := 4*t1
t3 := A[t2] \ ; A[j]
t4 := j+1
t5 := t4-1
t6 := 4*t5
t7 := A[t6] \ ; A[j+1]
\]

if t3<=t7 goto s3

\[
t8 := j-1
t9 := 4*t8
temp := A[t9] \ ; A[j]
t10 := j+1
t11 := t10-1
t12 := 4*t11
t13 := A[t12] \ ; A[j+1]
t14 := j-1
t15 := 4*t14
t16 := j+1
t17 := t16-1
t18 := 4*t17
\]

s3: j := j+1
goto S4

S2: i := i-1
goto s5

s1:
Representation: a Basic Block

• **Basic block** = a sequence of 3-address statements
  – only the first statement can be reached from outside the block (no branches into middle of block)
  – all the statements are executed consecutively if the first one is (no branches out or halts except perhaps at end of block)

• **We require basic blocks to be** *maximal*
  – they cannot be made larger without violating the conditions

• **Optimizations within a basic block are** *local* optimizations
Flow Graphs

• **Nodes:** basic blocks

• **Edges:** $B_i \rightarrow B_j$, iff $B_j$ can follow $B_i$ immediately in some execution
  
  – Either first instruction of $B_j$ is target of a goto at end of $B_i$
  
  – Or, $B_j$ physically follows $B_i$, which does not end in an unconditional goto.

• The block led by first statement of the program is the *start*, or *entry* node.
Find the Basic Blocks

\[ i := n-1 \]

S5: \text{if } i<1 \text{ goto s1}
\[ j := 1 \]

s4: \text{if } j>i \text{ goto s2}
\[ t1 := j-1 \]
\[ t2 := 4*t1 \]
\[ t3 := A[t2] ; A[j] \]
\[ t4 := j+1 \]
\[ t5 := t4-1 \]
\[ t6 := 4*t5 \]
\[ t7 := A[t6] ; A[j+1] \]
\text{if } t3<=t7 \text{ goto s3}

\[ t8 := j-1 \]
\[ t9 := 4*t8 \]
\text{temp := A[t9] ; A[j]} \]
\[ t10 := j+1 \]
\[ t11 := t10-1 \]
\[ t12 := 4*t11 \]
\[ t14 := j-1 \]
\[ t15 := 4*t14 \]
\[ t16 := j+1 \]
\[ t17 := t16-1 \]
\[ t18 := 4*t17 \]

s3: \text{j := j+1}
goto S4

S2: \text{i := i-1}
goto s5

s1:
Basic Blocks from Example

in

B1

B2

B3

B4

B5

B6

B7

B8

out
Partitioning into Basic Blocks

• Identify the leader of each basic block
  – First instruction
  – Any target of a jump
  – Any instruction immediately following a jump

• Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)
\[ 
\begin{align*}
1) & \quad i = 1 \\
2) & \quad j = 1 \\
3) & \quad t_1 = 10 \times i \\
4) & \quad t_2 = t_1 + j \\
5) & \quad t_3 = 8 \times t_2 \\
6) & \quad t_4 = t_3 - 88 \\
7) & \quad a[t_4] = 0.0 \\
8) & \quad j = j + 1 \\
9) & \quad \text{if } j \leq 10 \text{ goto (3)} \\
10) & \quad i = i + 1 \\
11) & \quad \text{if } i \leq 10 \text{ goto (2)} \\
12) & \quad i = 1 \\
13) & \quad t_5 = i - 1 \\
14) & \quad t_6 = 88 \times t_5 \\
15) & \quad a[t_6] = 1.0 \\
16) & \quad i = i + 1 \\
17) & \quad \text{if } i \leq 10 \text{ goto (13)} \\
\end{align*} \]
Sources of Optimizations

• Algorithm optimization

• Algebraic optimization
  \[ A := B + 0 \quad \Rightarrow \quad A := B \]

• Local optimizations
  – within a basic block -- across instructions

• Global optimizations
  – within a flow graph -- across basic blocks

• Interprocedural analysis
  – within a program -- across procedures (flow graphs)
Local Optimizations

• Analysis & transformation performed within a basic block
• No control flow information is considered
• Examples of local optimizations:
  – local common subexpression elimination
    analysis: same expression evaluated more than once in b.
    transformation: replace with single calculation
  – local constant folding or elimination
    analysis: expression can be evaluated at compile time
    transformation: replace by constant, compile-time value
  – dead code elimination
Example

\[ i := n-1 \]

S5: if i<1 goto s1
j := 1

s4: if j>i goto s2
t1 := j-1
t2 := 4*t1
t3 := A[t2] ; A[j]
t4 := j+1
t5 := t4-1
t6 := 4*t5
t7 := A[t6] ; A[j+1]
if t3<=t7 goto s3

\[ t8 := j-1 \]
\[ t9 := 4*t8 \]
\[ \text{temp := A[t9]} ; A[j] \]
\[ t10 := j+1 \]
\[ t11 := t10-1 \]
\[ t12 := 4*t11 \]
\[ t14 := j-1 \]
\[ t15 := 4*t14 \]
\[ t16 := j+1 \]
\[ t17 := t16-1 \]
\[ t18 := 4*t17 \]
\[ A[t18] := \text{temp} ; A[j+1] := \text{temp} \]

s3: j := j+1
goto S4

S2: i := i-1
goto s5

s1:
Example

B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
   t2 := 4*t1
   t3 := A[t2] ; A[j]
   \color{blue}{t6 := 4*j}
   t7 := A[t6] ; A[j+1]
   if t3<=t7 goto B8

B7: t8 := j-1
   t9 := 4*t8
   t12 := 4*j
   A[t9]:= t13 ; A[j]:=A[j+1]
   A[t12]:=temp ; A[j+1]:=temp
B8: j := j+1
   goto B4
B5: i := i-1
   goto B2
out:
(Intraprocedural) Global Optimizations

- **Global versions of local optimizations**
  - global common subexpression elimination
  - global constant propagation
  - dead code elimination

- **Loop optimizations**
  - reduce code to be executed in each iteration
  - code motion
  - induction variable elimination

- **Other control structures**
  - Code hoisting: eliminates copies of identical code on parallel paths in a flow graph to reduce code size.
Example

B1: \( i := n-1 \)
B2: if \( i < 1 \) goto out
B3: \( j := 1 \)
B4: if \( j > i \) goto B5
B6: \( t1 := j-1 \)
\( t2 := 4*t1 \)
\( t6 := 4*j \)
\( t7 := A[t6] ; A[j+1] \)
if \( t3 <= t7 \) goto B8

B7: \( t8 := j-1 \)
\( t9 := 4*t8 \)
\( t12 := 4*j \)

B8: \( j := j+1 \)
goto B4
B5: \( i := i-1 \)
goto B2
out:
Example (After Global CSE)

B1: $i := n-1$
B2: if $i<1$ goto out
B3: $j := 1$
B4: if $j>i$ goto B5
B6: $t1 := j-1$
    $t2 := 4*t1$
    $t3 := A[t2];A[j]$
    $t6 := 4*j$
    $t7 := A[t6];A[j+1]$
    if $t3<=t7$ goto B8
B5: $i := i-1$
    goto B2
B7: $A[t2] := t7$
    $A[t6] := t3$
B8: $j := j+1$
    goto B4
    goto B2
out:
Induction Variable Elimination

• Intuitively
  – Loop indices are induction variables (counting iterations)
  – Linear functions of the loop indices are also induction variables (for accessing arrays)

• Analysis: detection of induction variable

• Optimizations
  – strength reduction:
    • replace multiplication by additions
  – elimination of loop index:
    • replace termination by tests on other induction variables
Example

B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2];A[j]
    t6 := 4*j
    t7 := A[t6];A[j+1]
    if t3<=t7 goto B8
B7: A[t2] := t7
    A[t6] := t3
    B8: j := j+1
        goto B4
B5: i := i-1
    goto B2
out:
Example (After IV Elimination)

B1:   i := n-1
B2:   if i<1 goto out
B3:   t2 := 0
t6 := 4
B4:   t19 := 4*I
       if t6>t19 goto B5
B6:   t3 := A[t2]
t7 := A[t6] ;A[j+1]
       if t3<=t7 goto B8
B7:   A[t2] := t7
       A[t6] := t3
B8:   t2 := t2+4
t6 := t6+4
goto B4
B5:   i := i-1
goto B2
out:
Loop Invariant Code Motion

• **Analysis**
  – a computation is done within a loop and
  – result of the computation is the same as long as we keep going around the loop

• **Transformation**
  – move the computation outside the loop
Machine Dependent Optimizations

- Register allocation
- Instruction scheduling
- Memory hierarchy optimizations
- etc.
Local Optimizations (More Details)

• Common subexpression elimination
  – array expressions
  – field access in records
  – access to parameters
Example 1:
- grammar (for bottom-up parsing):
  \[ E \rightarrow E + T \mid E - T \mid T, \quad T \rightarrow T*F \mid F, \quad F \rightarrow (E) \mid \text{id} \]
- expression: \( a + a*(b-c) + (b-c)*d \)
Graph Abstractions

Example 1: an expression
\[ a + a \times (b - c) + (b - c) \times d \]

Optimized code:
\[
\begin{align*}
    t1 &= b - c \\
    t2 &= a \times t1 \\
    t3 &= a + t2 \\
    t4 &= t1 \times d \\
    t5 &= t3 + t4
\end{align*}
\]
How well do DAGs hold up across statements?

• Example 2

\[
\begin{align*}
a &= b+c; \\
b &= a-d; \\
c &= b+c; \\
d &= a-d;
\end{align*}
\]

Is this optimized code correct?

\[
\begin{align*}
a &= b+c; \\
d &= a-d; \\
c &= d+c;
\end{align*}
\]
Critique of DAGs

• **Cause of problems**
  – Assignment statements
  – Value of variable depends on TIME

• **How to fix problem?**
  – build graph in order of execution
  – attach variable name to latest value

• **Final graph created is not very interesting**
  – Key: variable->value mapping across time
  – loses appeal of abstraction
Value Number: Another Abstraction

• More explicit with respect to VALUES, and TIME

  • each value has its own “number”
    – common subexpression means same value number
  • var2value: current map of variable to value
    – used to determine the value number of current expression
      \[ r1 + r2 \Rightarrow \text{var2value}(r1) + \text{var2value}(r2) \]
Algorithm

Data structure:
VALUES = Table of
expression     // [OP, valnum1, valnum2]
var            // name of variable currently holding expression

For each instruction (dst = src1 OP src2) in execution order

valnum1 = var2value(src1); valnum2 = var2value(src2);

IF [OP, valnum1, valnum2] is in VALUES
  v = the index of expression
  Replace instruction with CPY dst = VALUES[v].var
ELSE
  Add
  expression = [OP, valnum1, valnum2]
  var        = dst
  to VALUES
  v = index of new entry; tv is new temporary for v
  Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                           dst = tv;

set_var2value (dst, v)
More Details

• What are the initial values of the variables?
  – values at beginning of the basic block

• Possible implementations:
  – Initialization: create “initial values” for all variables
  – Or dynamically create them as they are used

• Implementation of VALUES and var2value: hash tables
Example

Assign: a→r1, b→r2, c→r3, d→r4

a = b+c;  
ADD t1 = r2,r3
CPY r1 = t1

b = a-d;  
SUB t2 = r1,r4
CPY r2 = t2

c = b+c;  
ADD t3 = r2,r3
CPY r3 = t3

d = a-d;  
SUB t4 = r1,r4
CPY r4 = t4
Conclusions

• Comparisons of two abstractions
  – DAGs
  – Value numbering

• Value numbering
  – VALUE: distinguish between variables and VALUES
  – TIME
    • Interpretation of instructions in order of execution
    • Keep dynamic state information
CSC D70:
Compiler Optimization
Introduction, Logistics

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons