Introduction to CS 293S
Code Optimizations for Scalar and Parallel Programs

Yufei Ding
About Me

- Yufei Ding, Assist. Prof in CS
- addressed by Prof. Ding
- Married, one kid
Research Interest

Making computing more intelligent and efficient through software systems (compiler, runtime, library, tools, etc.)

And many other interesting research problems (e.g., Machine Learning, Artificial Intelligence)
Teaching Philosophy

Make students think…
and think critically.
Course Management

- No textbooks.
- Slides or notes won’t be posted!
  - Handouts define the scope
  - Have holes to fill in class.
Grading Policy

- 30% final term, 30% assignment (3 homework), 40% paper review and presentation.
- No cheating.
- No late submission accepted.
Communication

See course webpage for other important policy and details.
http://www.cs.ucsb.edu/~yufdeiding/cs293s

Piazza: https://piazza.com/ucsb/fall2018/cs293s/home

Email to yufdeiding@cs.ucsb.edu.
Introduction to Code Optimization

- Simple definition: Enhance the quality of a program.
- What is the metric for quality of a program?
- Why is it important to enhance the quality?
Introduction to Code Optimization

Simple definition: Enhance the quality of a program.

What is the quality of a program?

Speed, energy, power, code size, memory footprint, reliability, security, resilience, readability, extensibility, etc.
Importance of Code Optimization

- Modern humanity development is based on computing
- Code quality determines the quality of computing and hence the quality of humanity development
Importance of Code Optimization

- Scientifically: scope and precision of scientific simulation, reliability for critical missions
- Economically: “1% performance improvement saves Google millions of dollars” —Google
- Health, defense, …
Introduction to Code Optimization

- Simple definition: Enhance the quality of a program.
- Who makes the enhancement?
- How to do it?
Introduction to Code Optimization

- Simple definition: Enhance the quality of a program.
- Who makes enhancement?
  - Compiler, runtime, programmer
- How to do it? *The core of this course.*
  - Program analysis to understand programs
  - Program transformation to materialize the enhancement
Overview of Compiler
Compilers

What is a compiler?
- A program that translates a program in one language into a program in another language.
- It should improve the program, in some way.

What is an interpreter?
- A program that reads a program and produces the results of executing that program.
Compilers

- C is typically compiled, Scheme is typically interpreted
- Java is compiled to bytecodes (code for the Java VM)
  - which are then interpreted
  - Or a hybrid strategy is used
    - Just-in-time compilation
Use an intermediate representation (IR)

Front end maps legal source code into IR

Back end maps IR into target machine code

Admits multiple front ends and backends

Middle end with multiple passes for different optimizations

Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NPC
The Front End

Scanner (syntax)

Maps character stream into words & their parts of speech (e.g. id, number, +)

\[ x = x + 2 ; \]

becomes \(<\text{id},x> = <\text{id},x> + <\text{number},2> ;\)

Analogy:

Dogs are animals. => noun verb noun
The Front End

Scanner (syntax)

- Break the inputs into individual pieces
- Decide the functionality of each piece

\[ x = x + 2 ; \]

becomes

\[ <\text{id},x> = <\text{id},x> + <\text{number},2> ; \]

- Analogy:

Dogs are animals. => noun verb noun
The Front End

Parser
1. Organize the pieces back based on some predefined rules
2. Reports errors
3. Analogy:
   
   Dogs are animals.  ==>  

   ![Diagram of a parse tree with nodes labeled 'sent.', 'noun', 'verb phrase', 'Dogs', 'verb', 'are', 'noun', 'animals']
The Front End

A parser can be represented by a tree (parse tree or syntax tree)

\[ x + 2 - y \]

This contains a lot of unneeded Information.
The Front End

Compilers often use an abstract syntax tree (AST)

\[ x + 2 - y \]

The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise.

AST is one form of intermediate representation (IR)
Traditional Three-Pass Compiler

Code Improvement (or Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  - Definition of “meaning” varies
Modern optimizers are structured as a series of passes

Typical Transformations
- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
The Back End

Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been much less successful in the back end
Classification of Compilers

- **Time of compilation**
  - Offline compilation
    - e.g., GCC
  - Just-In-Time compilation (JIT)
    - e.g. JIT in Java Virtual Machine
    - e.g., Javascript compiler (V8) in Chrome

- **Unit of compilation**
  - Function (or Method)
    - e.g., GCC
  - Trace
    - e.g., Old Javascript JIT in Mozilla
# Content

<table>
<thead>
<tr>
<th>Fundamental Knowledge</th>
<th>intro</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>redundancy removal</td>
</tr>
<tr>
<td></td>
<td>data flow &amp; ssa</td>
</tr>
<tr>
<td></td>
<td>other optimizations</td>
</tr>
<tr>
<td></td>
<td>dependence theory</td>
</tr>
<tr>
<td>Exiting Research</td>
<td>parallelization</td>
</tr>
<tr>
<td></td>
<td>GPU</td>
</tr>
<tr>
<td></td>
<td>autotuning</td>
</tr>
<tr>
<td></td>
<td>general redundancy removal</td>
</tr>
</tbody>
</table>