CS 243: Advanced Compilers Course

Lecture 1

Introduction

I. Why Study Compilers?
II. Mathematical Abstractions: with Examples
III. Course Syllabus

Chapters 1.1-1.5, 8.4, 8.5, 9.1
Why Study Compilers?

Impact!
Techniques in compilers help all programmers
Compiler Technology: Key Programming Tool

Bridge the semantic gap between programmers and machines

Programming languages
- High-level programming languages
- Domain-specific languages
- Natural language

Computer architecture
- RISC vs CISC, Systolic arrays
- Locality: Caches, memory hierarchy
- Parallelism:
  - Instruction-level parallelism
  - Multi-processors

Pro grammers

Programming Language

Compilers

Machine

Programming Tools
- Security
- Verification
- Binary translations
Compiler Study: a Software Engineering Course

Trains Good Developers

- Reasoning about programs makes better programmers
- Tool building: there are programmers and there are tool builders …
- Excellent software engineering case study: Compilers are hard to build
  - Input: all programs
  - Objectives:

- Methodology for solving complex real-life problems
  - Build upon mathematical / programming abstractions
Compilers: Where Theory Meets Practice

- Desired solutions are often NP-complete / undecidable
- Key to success: Formulate the right abstraction / approximation
  - Can’t be solved by just pure hacking
    • theory aids generality and correctness
  - Can’t be solved by just theory
    • experimentation validates & provides feedback to problem formulation
- Tradeoffs: Generality, power, simplicity, and efficiency
Why Study Compilers?

Impact!
Techniques in compilers help all programmers

Better Programmer
Reasoning about programs
Mathematical abstractions
Course Emphasis

• **Methodology: apply the methodology to other real-life problems**
  – Design
    • Problem statement: Which problem to solve?
    • New programming abstraction through domain-specific languages
  – Theory and Algorithm
    • Theoretical frameworks
    • Algorithms
  – Experimentation: Hands-on experience
    (Weekly programming/written homeworks)

• **Compiler knowledge:**
  – Non-goal: how to build a complete optimizing compiler
  – Important algorithms
  – Exposure to new ideas
  – Background to learn existing techniques
Interactive Instruction

• Compilers are not about memorizing facts
  – Open-book examinations
• Goal: teach how to derive the concepts
  – So you can apply to new problems
  – Lectures are interactive
  – Please come to class
  – The slides may miss main points to be emphasized in class!
    • These slides supplement lectures
    • They are not self contained!
    • They may contain mistakes, corrected in class!
The Rest of this Lecture

• Goal
  – Overview the course
  – Explain why I chose the topics
  – Emphasize abstraction methodology

• For each topic:
  – Motivate its importance
  – Show an example to illustrate the complexity
  – Describe the abstraction
  – Impact
1. Optimizing Compilers for High-Level Programming Languages

• Redundancy elimination
  – High-level programming languages introduce a lot of redundancies in programs that programmers are not aware of.

• Example:
  Bubblesort program that sorts array A allocated in static storage

```c
for (i = n-2; i >= 0; i--) {
    for (j = 0; j <= i; j++) {
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```

Quiz: what is the best way to speed up this task?
Code Generated by the Front End

\[
\begin{align*}
    i & := n-2 \\
    S5: \quad & \text{if } i < 0 \text{ goto } s1 \\
    j & := 0 \\
    s4: \quad & \text{if } j > i \text{ goto } s2 \\
    t1 & = 4 * j \\
    t2 & = &A \\
    t3 & = t2 + t1 \\
    t4 & = *t3 \quad ;A[j] \\
    t5 & = j + 1 \\
    t6 & = 4 * t5 \\
    t7 & = &A \\
    t8 & = t7 + t6 \\
    t9 & = *t8 \quad ;A[j+1] \\
    \text{if } t4 & \leq t9 \text{ goto } s3 \\
    t10 & = 4 * j \\
    t11 & = &A \\
    t12 & = t11 + t10 \\
    \text{temp} & = *t12 \quad ;\text{temp}=A[j] \\
\end{align*}
\]

\[
\begin{align*}
    t13 & = j + 1 \\
    t14 & = 4 * t13 \\
    t15 & = &A \\
    t16 & = t15 + t14 \\
    t17 & = *t16 \quad ;A[j+1] \\
    t18 & = 4 * j \\
    t19 & = &A \\
    t20 & = t19 + t18 \quad ;&A[j] \\
    *t20 & = t17 \quad ;A[j]=A[j+1] \\
    t21 & = j + 1 \\
    t22 & = 4 * t21 \\
    t23 & = &A \\
    t24 & = t23 + t22 \\
    *t24 & = \text{temp} \quad ;A[j+1]=\text{temp} \\
    t10 & = 4 * j \\
    t11 & = &A \\
    t12 & = t11 + t10 \\
    \text{temp} & = *t12 \quad ;\text{temp}=A[j] \\
    s3: \quad & j = j + 1 \\
    s2: \quad & i = i - 1 \\
    s1: \quad & \text{goto } s5 \\
\end{align*}
\]

\(t4=*t3\) means read memory at address in \(t3\) and write to \(t4\): 
\(\ast t20=t17\): store value of \(t17\) into memory at address in \(t20\)
After Optimization

Result of applying:
- global common subexpression
- loop invariant code motion
- induction variable elimination
- dead-code elimination

to all the scalar and temp variables

These traditional optimizations can make a big difference!

```c
i = n-2

t27 = 4*i

t28 = &A

t29 = t27 + t28

t30 = t28 + 4

S5: if t29 < t28 goto s1

t25 = t28

t26 = t30

s4: if t25 > t29 goto s2

t4 = *t25 ; A[j]

t9 = *t26 ; A[j+1]

if t4 <= t9 goto s3

temp = *t25 ; temp=A[j]

t17 = *t26 ; A[j+1]
*t25 = t17 ; A[j]=A[j+1]
*t26 = temp ; A[j+1]=temp

s3: t25 = t25 + 4

t26 = t26 + 4

goto S4

S2: t29 = t29 - 4

goto s5

s1:
```
DataFlow Framework

• High-level programming languages
  – Need many optimizations to be efficient

• Data flow
  – A general framework
    • Finds fixed-point solution to a set of recurrence equations
    • Monotonicity
  – Theory: prove correctness properties once and for all
  – Implementation: same code reused
## Summary

<table>
<thead>
<tr>
<th>Topic</th>
<th>Abstraction</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow optimizations</td>
<td>Graphs</td>
<td>High-level programming without loss of efficiency</td>
</tr>
<tr>
<td>1970-1980s</td>
<td>Recurrent equations Fixed-point</td>
<td></td>
</tr>
</tbody>
</table>

Consumers can automate personal and professional tasks themselves, eliminating dependence on coders.
2. High Performance Computing / Machine Learning

- Large Language Models (LLMs)
- GPT-3 in 2020
  - Trained to predict the next word
  - Unsupervised: 45 TB of Internet text
  - 175 Billion parameters
- Training
  - 10,000 V100 GPUs ($4,600,000)
  - 1287000 KWh
  - 9 days
Trends of LLMs

YaLM & GPT-4 (3/14/2023)

- Multimodal: text, images
Nvidia Volta GV100 GPU

- 21B transistors
- 815 mm\(^2\)
- 1455 Mhz
- 80 Stream Multiprocessors (SM)

https://wccftech.com/nvidia-volta-tesla-v100-cards-detailed-150w-single-slot-300w-dual-slot-gv100-powered-pcie-accelerators/
In Each SM

- 64 FP32 cores
- 64 int cores
- 32 FP64 cores
- 8 Tensor cores

Tensor Cores

\[ D = A \times B + C; \ A, B, C, D \] are 4x4 matrices

- 4 x 4 x 4 matrix processing array
- 1024 floating point ops / clock

FP32: 15 TFLOPS
FP64: 7.5 TFLOPS
Tensor: 120 TFLOPS

https://wccfte.com/nvidia-volta-tesla-v100-cards-detailed-150w-single-slot-300w-dual-slot-gv100-powered-pcie-accelerators/
Matrix Multiplication

\[
\begin{array}{ccc}
\text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ }
\end{array}
\begin{array}{ccc}
\text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ }
\end{array}
\begin{array}{c}
\text{x}
\end{array}
\begin{array}{ccc}
\text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ }
\end{array}
\begin{array}{ccc}
\text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ }
\end{array}
\]

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
        }
    }
}

- \(n^3\) computation
- \(n^2\) threads of parallelism
- 2 memory operations per multiply-add operation
- Bottleneck: memory operations
Multiprocessor Architecture

- Memory accesses are much more expensive than multiply-add
- Interconnect becomes a bottleneck – scalability!
Systolic Arrays

• Introduced by Kung and Leiserson in 1978
• Special-purpose computer architecture for specific algorithms
• Processor interconnect matches algorithm communication pattern
• Eliminates the memory bottleneck
• TPU: 128 x 128 multiply/accumulators in a systolic array
Google TPU-v4 chips, 2022

TPU v4 chip

Matrix multiplication unit: 128 x 128 multiply/accumulators in a systolic array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak compute per chip</td>
<td>275 teraflops</td>
</tr>
<tr>
<td>Min/mean/max power</td>
<td>90/170/192 W</td>
</tr>
<tr>
<td>TPU pod size</td>
<td>4096 chips</td>
</tr>
<tr>
<td>Peak compute per pod</td>
<td>1.1 exaflops</td>
</tr>
</tbody>
</table>

https://cloud.google.com/tpu/docs/system-architecture-tpu-vm
Google TPU-v4 System

Google PaLM: 540B parameters in large language models
Using 6144 TPU v4 chips (1.7 exaflops)
Principle to Successful Parallelism

• Parallel execution can be slower than sequential execution
  – Because of communication overhead!

• Goal: maximize parallelism and minimize communication

• Principles applicable to uniprocessors (caches) and multiprocessors
Blocking for Matrix Multiplication

\[
\begin{align*}
&= \\

1000 & = \\
32 & = \\

\text{Data Accessed} & \quad 1002000 \\
& \quad 65024
\end{align*}
\]
Blocking with Matrix Multiplication

• Original program
  
  ```
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
        Z[i, j] = Z[i, j] + X[i, k] * Y[k, j];
      }
    }
  }
  ```

• Stripmine 2 outer loops
  
  ```
  for (ii = 0; ii < n; ii = ii + B) {
    for (i = ii; i < min(n, ii + B); i++) {
      for (jj = 0; jj < n; jj = jj + B) {
        for (j = jj; j < min(n, jj + B); j++) {
          Z[i, j] = Z[i, j] + X[i, k] * Y[k, j];
        }
      }
    }
  }
  ```

• Permute loops
  
  ```
  for (ii = 0; ii < n; ii = ii + B) {
    for (jj = 0; jj < n; jj = jj + B) {
      for (k = 0; k < n; k++) {
        for (i = ii; i < min(n, ii + B); i++) {
          for (j = jj; j < min(n, jj + B); j++) {
            Z[i, j] = Z[i, j] + X[i, k] * Y[k, j];
          }
        }
      }
    }
  }
  ```
Experimental Results

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
</tr>
</tbody>
</table>

- With Blocking
- Without Blocking
Affine Framework

• Many useful loop transformations for locality & parallelism
  – Loop interchange, reversal, skewing
  – Loop fusion, fission
  – Blocking

• Affine Transformations
  – Inspired by systolic arrays
  – For dense matrix computations
  – A general framework
    • Geometric transforms (linear algebra)
    • Maximizes parallelism and minimizes communication
      by solving linear inequality constraints
## Summary

<table>
<thead>
<tr>
<th>Topic</th>
<th>Abstraction</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow optimizations</td>
<td>Graphs</td>
<td>High-level programming without loss of efficiency.</td>
</tr>
<tr>
<td>1970-1980s</td>
<td>Recurrent equations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td></td>
</tr>
<tr>
<td>Parallelism and locality optimizations</td>
<td>Integer linear programming</td>
<td>Hide complexity from programmers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Systolic arrays.</td>
</tr>
</tbody>
</table>
Garbage Collection

• Automatic memory management
  – Hugely improves program robustness and developer productivity

• Original: Naïve stop-the-world garbage collection
  – Stops the program to trace the reachability of all the objects

• Key optimizations → greatly reduce pause time
  – Incremental: break up GC in time
  – Partial: break up GC in space
# Summary

<table>
<thead>
<tr>
<th>Topic</th>
<th>Abstraction</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow optimizations</td>
<td>Graphs</td>
<td>High-level programming without loss of efficiency.</td>
</tr>
<tr>
<td>1970-1980s</td>
<td>Recurrent equations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td></td>
</tr>
<tr>
<td>Parallelism and locality optimizations</td>
<td>Integer linear programming</td>
<td>Hide complexity from programmers</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>Incremental and partial GC</td>
<td>Remove manual memory management</td>
</tr>
<tr>
<td>1990-2000s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Program Analysis

• Can we build tools to help programmers with complex tasks:
  – Finding bugs
  – Program verification

• Requires reasoning about program
  – Pointer alias analysis (not covered in this class)
  – Execution paths
Example: Out-of-Bound Array Access

Program  Assume data array bound is [0, N-1]

1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5      {
6        int next;
7        next = data[i];
8        if (!(i < next && next < N)) return;
9        i = i + 1;
10       for (; i < next; i = i + 1){
11         if (data[i] == cookie)
12            i = i + 1;
13         else
14            Process(data[i]);
15       }
16     }
17  }

When is the array access in line 7 out of bound?
-- after data[i] == cookie, i = i + 1
Satisfiability Modulo Theories (SMT)

• Satisfiability
  – the problem of determining whether a formula has a model (an assignment that makes the formula true)

• SAT: Satisfiability of **propositional formulas**
  – A model is a truth assignment to Boolean variables
  – SAT solvers: check satisfiability of propositional formulas
    • Decidable, NP-complete

• SMT: Satisfiability modulo theories
  – Satisfiability of first-order formulas containing operations from background theories such as arithmetic, arrays, uninterpreted functions, etc.

• SMT Solvers:
  – check satisfiability of SMT formulas with respect to a theory
## Summary

<table>
<thead>
<tr>
<th>Topic</th>
<th>Abstraction</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow optimizations</td>
<td>Graphs</td>
<td>High-level programming without loss of efficiency.</td>
</tr>
<tr>
<td>1970-1980s</td>
<td>Recurrent equations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallelism and locality optimizations</td>
<td>Integer linear programming</td>
<td>Hide complexity from programmers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Garbage collection</td>
<td>Incremental and partial GC</td>
<td>Remove manual memory management</td>
</tr>
<tr>
<td>1990-2000s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Satisfiability modulo</td>
<td></td>
</tr>
<tr>
<td>Program analysis for correctness</td>
<td>theories (SMT)</td>
<td>Improve program robustness. Save programmers debugging time.</td>
</tr>
<tr>
<td>2000-2020s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NLP: Natural Language Processing

- Large language models (LLMs) e.g. GPT-3, chatGPT

1. Help programmers with their task
   - Writing "popular" programs from a description
   - Example: website

   “I would like you to act as a frontend web developer. For the project, you’ll code a new website using these tools: HTML, Bootstrap framework using the CDN for CSS and JavaScript. The website should be mobile-friendly and responsive. It should also include the most recent version of Twitter Bootstrap CSS classes in the site structure for layout and style. When it’s all done, there should be a single HTML file. You should also include a navigation menu with internal links to the headings within the page content. Do not provide explanations for any of the code you write.”

https://themeisle.com/blog/how-to-use-chatgpt-to-build-a-website/#gref
NLP: Natural Language Processing

- Large language models (LLMs) e.g. GPT-3, chatGPT

1. Help programmers with their task
   - Writing "popular" programs from a description.
   - Improving programmers' productivity with autocompletion
     - As you code, or from your comments
     - OpenAI codex
       - Trained to predict the next word on the internet corpus, open-source code.

https://themeisle.com/blog/how-to-use-chatgpt-to-build-a-website/#gref
NLP: Natural Language Processing

• Large language models (LLMs) e.g. GPT-3, chatGPT

1. Help programmers with their tasks (Not a focus in this course)
   – Writing "popular" programs from a description.
   – Improving programmers' productivity with autocompletion

2. End user programming (Focus in this course)
   – Let everybody program with the highest-level programming language
Example 1: Powerpoint

- Lots and lots of nested menus
- Takes a long time to make slides (even if you know what functions are available)
Multimodal Commands

– Make this text box bold in the slide master.

`Slide.Current().getSlideMaster().matching(field:.id,value:Shape.Current().id)
 .textFrame.textRange.font.setBold(bold: true)`

– Make the border of this shape with little dots.

`Shape.Current().lineFormat.setDashStyle(dashStyle:"RoundDot")`

– Make everything right aligned on this slide.

 setHorizontalAlignment(horizontalAlignment:"Right")`

– Make every shape on this slide above this yellow.

`Slide.Current().getShapes().between(field:.top,to:Shape.Current().top).
 fill.setForeGroundColor(color:"yellow")`
Demo

ReactGenie System Architecture

Semantic Parser

Runtime

Response Generator

UI Mapping

UI Update
Example 2: A YelpBot

- Yelp: API access to databases, which include reviews
- Created a new DSL: SUQL (structured and unstructured query language)

```
"SELECT *, summary(reviews) FROM restaurants WHERE 'szechuan' = ANY (cuisines) AND location = "Palo Alto" AND answer(reviews, 'does this restaurant serve spicy food?') = 'Yes' LIMIT 1;"
```

```
"SELECT answer(reviews, 'does this restaurant have non-spicy options?') FROM restaurants WHERE name ILIKE 'Taste' AND location = 'Palo Alto';"
```
End-User Natural Language Programming

• New DSL designed for natural language programming

• Neural semantic parser: Use LLMs (with fine-tuning)

• Examples
  – ReactGenieDSL: Multimodal programming
    • Developer add a few annotations to React code
  – SUQL: Structured and Unstructured Data Queries
    • Using LLMs as a subroutine
    • Optimizing compiler
## Summary

<table>
<thead>
<tr>
<th>Topic</th>
<th>Abstraction</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recurrent equations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallelism and locality optimizations 1980-1990s</td>
<td>Integer linear programming</td>
<td>Hide complexity from programmers</td>
</tr>
<tr>
<td></td>
<td>Linear algebra</td>
<td>Machine independence code. Systolic arrays.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Garbage collection 1990-2000s</td>
<td>Incremental and partial GC</td>
<td>Remove manual memory management</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program analysis for correctness 2000-2020s</td>
<td>Satisfiability modulo theories (SMT)</td>
<td>Improve program robustness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Save programmers debugging time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End user programming in natural language 2010-2020s</td>
<td>DSL for NL translation</td>
<td>New generation of natural, powerful user interfaces</td>
</tr>
<tr>
<td></td>
<td>Neural semantic parser</td>
<td></td>
</tr>
</tbody>
</table>

Best of time-tested concepts in compilers!
# Tentative Course Schedule

<table>
<thead>
<tr>
<th>No.</th>
<th>Topic</th>
<th>Subtopics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Course Introduction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Data Flow Optimizations</td>
<td>Data-flow analysis: introduction</td>
</tr>
<tr>
<td>3</td>
<td>Data Flow Optimizations</td>
<td>Data-flow analysis: theoretic foundation</td>
</tr>
<tr>
<td>4</td>
<td>Optimization: constant propagation</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Optimization: redundancy elimination</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Machine Dependent Optimizations</td>
<td>Register allocation</td>
</tr>
<tr>
<td>7</td>
<td>Instruction Scheduling</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Software pipelining</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Loop Transformations</td>
<td>Parallelization</td>
</tr>
<tr>
<td>10</td>
<td>Loop Transformations</td>
<td>Loop transformations</td>
</tr>
<tr>
<td>11</td>
<td>Loop Transformations</td>
<td>Pipelined parallelism</td>
</tr>
<tr>
<td>12</td>
<td>Loop Transformations</td>
<td>Locality + parallelism</td>
</tr>
<tr>
<td>13</td>
<td>Satisfiability Modulo Theories</td>
<td>Introduction</td>
</tr>
<tr>
<td>14</td>
<td>Satisfiability Modulo Theories</td>
<td>Examples</td>
</tr>
<tr>
<td>15</td>
<td>Natural Language Programming</td>
<td>Structured and unstructured query lang</td>
</tr>
<tr>
<td>16</td>
<td>Natural Language Programming</td>
<td>Multimodal programming</td>
</tr>
<tr>
<td>17</td>
<td>Garbage Collection</td>
<td>Advanced Techniques</td>
</tr>
</tbody>
</table>
Homework

- **Due Wednesday (no need to hand in)**

- Read Chapter 9.1 for introduction of the optimizations

- Work out the example on pages 10-12 in this handout.