CS201 2023 Fall Course Project

- Prerequisites: C++ programming, basic Linux
- Part 1: Warm up (learning LLVM basics)
  - 10 pts
  - Deadline: Nov. 6
- Part 2: Implementing a basic analysis pass (Reaching definition analysis)
  - 10 pts
  - Deadline: Nov. 20
- Part 3: Implementing a transform pass (Common subexpression elimination)
  - 10 pts
  - Deadline: Dec. 4

The project (part 1, 2, and 3) can be done in groups (up to 2 students). When submitting projects, put all member names and student numbers in the PDF report (part 1), or attach a PDF file indicating group members along with your code. (**One submission per group**)

Part 1: Warm up (learning LLVM basics)

- Objectives
  - Install and set up the LLVM 12 environments.
  - Get familiar with its basic toolchains.
    - https://clang.llvm.org/get_started.html
    - https://llvm.org/docs/CommandGuide/index.html
  - Learn how to write LLVM passes.
- Tasks:
  - Install LLVM (from source code).
  - Figure out commands to translate between different code representations (step 2-b).
  - Modify the **HelloPass** to print number of predecessors and successors (step 3).

Part 1 (Step 1)

- Install and setup LLVM 12 environment

- Detailed instructions are posted on Canvas (CS201_23F_Project_Part1.pdf)

Part 1 (Step 2a and 2b)

- Step 2a:
  - Reference: https://llvm.org/docs/CommandGuide/index.html and https://clang.llvm.org/get_started.html
  - Try these commands: clang, opt, llc, lli, llvm-link, llvm-as, llvm-dis

- Step 2b: Translate between different code representations:
  - (i) source (.c) to binary (executable)
  - (ii) source (.c) to object file (.o)
  - (iii) source (.c) to machine assembly (.s)
  - (iv) source (.c) to LLVM bitcode (.bc); source (.c) to LLVM IR (.ll)
  - (v) LLVM IR (.ll) to LLVM bitcode (.bc)
  - (vi) LLVM bitcode (.bc) to LLVM IR (.ll)
  - (vii) LLVM IR (.ll) to machine assembly (.s)

Part 1: Warm up (learning LLVM basics)

C code:

```
c = a + 1;
```

LLVM IR:

```
%0 = load i32, i32* %a, align 4

%add = add nsw i32 %0, 1

store i32 %add, i32* %c, align 4
```

Part 1 (Step 3)

<u>Structure of the working directory</u>

- **CS201-F23-Template**/
  - README.md
  - **test/**
    - **phase1/**
      - Test.c —-> Translate this code and run with the compiled pass
    - **phase2**
    - **phase3**
  - **Pass/**
    - CMakeLists.txt
    - **build/**
    - **HelloPass/**
      - CMakeLists.txt
      - HelloPass.cpp —--> Edit this file and compile
    - **ReachingDefinition**
    - **CSElimination**

LLVM basics

- The implemented Pass extends from **FunctionPass** class and overrides **runOnFunction(Function &F)** function:
    - runOnFunction(Function &F) function gets called for each function in the test code.

```
struct HelloPass : public FunctionPass {
          .......
          bool runOnFunction(Function &F) override{
                    .......
          }
}
```

- Iterate over basic blocks of the given function:

```
bool runOnFunction(Function &F) override {
          for (auto& basic_block : F)
          {
                    ...
          }
}
```

LLVM basics

- Iterate over the instructions in a basic block (BB). **Note:** instructions are in LLVM IR

```
bool runOnFunction(Function &F) override {
        for (auto& basic_block : F)
        {
                for (auto& inst : basic_block)
                {
                        ...
                }
        }
}
```

- Access the operands using getOperand(operand_index):

```
for (auto& inst : basic_block)
{
    ...
    errs() << "operand: " << inst.getOperand(0) << "\n";
    ...
}
```

LLVM basics for Part 1 (Step 3)

- Check whether instruction is a binary operation and find operator types

```
if (inst.isBinaryOp())
{
        inst.getOpcodeName(); //prints OpCode by name such as add, mul etc.
        if(inst.getOpcode() == Instruction::Add)
        {
                errs() << "This is Addition"<<"\n";
        }
        if(inst.getOpcode() == Instruction::Mul)
        {
                errs() << "This is Multiplication"<<"\n";
        }
    // See Other classes Instruction::Sub, Instruction::UDiv, Instruction::SDiv
}
```

- predecessors and successors of a basic block:

```
bool runOnFunction(Function &F) override {
        for (auto& basic_block : F)
        {
                for (auto *pred: predecessors(&basic_block)) {
                }
                for (auto *succ: successors(&basic_block)) {
                }
        }
        return false;
}
```

Part 1 (Step 4)

- Write a report that lists your experiments in step 2b and step 3 with command and input/outputs.

- Submit your report (in PDF) along with your work folder, which includes the temporary files you used and generated in step 2 and step 3

- PDF format: **CS201-23Fall-Part1-StudentNumbers(each group member).pdf**

Part 2: Implementing reaching definition analysis pass

- Forward May Problem
  - IN[B]: Definitions that reach B’s entry.
  - OUT[B]:Definitions that reach B’s exit.
  - GEN[B], KILL[B]: …
- Your implementation should use **iterative** algorithm or the **worklist** one.
- Implementation: finish *Pass/ReachingDefinition/ReachingDefinition.cpp*
- Compiling:

```
cd ../build/
cmake -DCMAKE_BUILD_TYPE=Release ../ReachingDefinition
make
```

- Testing:
  - Scripts provided in test folder
    - `bash create_input.sh Test`: transform Test.c to Test.ll
    - `bash test.sh Test.ll`: run the pass and save output to Test.ll.out

Part 3: Implementing Common Subexpression Elimination

- Implement **Available expression pass**
- Use the results of available expression analysis and reaching definition analysis to transform the input program
- The transformation algorithm:

For each statement S: A = B op C st B op C is available at entry of S's basic block and neither B or C are redefined prior to S do the following:

1. Find definitions that reach S's block that have B op C on the right hand side.
2. Create a new name T.
3. Replace each statement D = B op C found in step 1 by: T = B op C; D = T;
4. Replace statement S by A = T.

Part 3: Implementing Common Subexpression Elimination

- A new variable ***tmp*** is created and placed at the beginning of **entry** block

<table>
<tr><th>Input Program</th><th>Output Program (transformed)</th></tr>
<tr><td>

```
define dso_local void @test() #0 {
entry:
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %e = alloca i32, align 4
  %f = alloca i32, align 4
  %0 = load i32, i32* %f, align 4
  store i32 %0, i32* %c, align 4
  %1 = load i32, i32* %e, align 4
  %cmp = icmp sgt i32 %1, 0
  br i1 %cmp, label %if.then, label %if.else
```

</td><td>

```
define dso_local void @test() #0 {
entry:
  %tmp = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %e = alloca i32, align 4
  %f = alloca i32, align 4
  %0 = load i32, i32* %f, align 4
  store i32 %0, i32* %c, align 4
  %1 = load i32, i32* %e, align 4
  %cmp = icmp sgt i32 %1, 0
  br i1 %cmp, label %if.then, label %if.else
```

</td></tr>
</table>

Part 3: Implementing Common Subexpression Elimination

- **e = b + c** is replaced by two instructions: **tmp = b + c; e = tmp;**

Input Program	Output Program (transformed)

<pre>if.else:
  %6 = load i32, i32* %b, align 4
  %7 = load i32, i32* %c, align 4
  %add1 = add nsw i32 %6, %7
  store i32 %add1, i32* %e, align 4
  br label %if.end
if.end:
  %8 = load i32, i32* %b, align 4
  %9 = load i32, i32* %c, align 4
  %add2 = add nsw i32 %8, %9
  store i32 %add2, i32* %a, align 4
  ret void</pre>
	<pre>if.else:
	  %7 = load i32, i32* %b, align 4
	  %8 = load i32, i32* %c, align 4
	  %add1 = add nsw i32 %7, %8
	  store i32 %add1, i32* %tmp, align 4
	  %9 = load i32, i32* %tmp, align 4
	  store i32 %9, i32* %e, align 4
	  br label %if.end
	if.end:
	  %10 = load i32, i32* %b, align 4
	  %11 = load i32, i32* %c, align 4
	  %12 = load i32, i32* %tmp, align 4
	  store i32 %12, i32* %a, align 4
	  ret void</pre>

Contact Information

- Email: cmama002@ucr.edu

Office Hours

- Tuesday: 1:00-2:00 PM
- Friday: 9:30 - 10:30 AM