Technical Document: Lab 1

Overview

Team

Role:	**Name:**
Manager	Luis Arcos
Technical Lead	Madylin Teel
Developer	Coleman Bixler
Developer	Garrett Snow
Developer	An Vu
Developer	Nathan Stephani

Conventions

Naming

Branch names	Example: {dev,techlead,manager}.<user>/<objective>/<task> i.e., dev.madyteel/obj-a/t-01
Generated output folders	Example: task##/<operation>-versions/ i.e., task02/split-versions/
C code file names	Example: var##-<operation/schedule>.x i.e., var00-ssa-schedule-blur3x3.c

Objective A: Handspun Simple Schedule Code Generator

Objective A focuses on the process of applying schedule transformation (by hand) on an operation in Static Single Assignment (SSA) form.

Static Single Assignment (SSA) – A representation used in compilers where each variable is assigned *exactly once* and defined before it is used.

Please see ` LAB-SCHEDULE-DECOUPLED/lab_inital/operations/` for all operations which need to be converted to SSA format.

Please see ` LAB-SCHEDULE-DECOUPLED/lab_inital/examples/basic-ssa/{ssa_var000.c, ssa_var001.c} for examples on how to convert the operations into SSA form.

T00: Handspun Baseline

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications in SSA form.

Implementation

Use ` ssa_var000.c and ssa_var001.c` as references for structuring loop transformations while maintaining SSA-compliance [from lab_inital/examples/basic-ssa/].

Steps:

1. Convert each individual operation into SSA form:
   a. Transform each operation file into SSA-compliant C code.
      i. Rewrite each computation such that every value is assigned exactly and only once.
      ii. Introduce new variables instead of reassigning existing ones.
   b. Ensure there is no variable reassignment!
2. Implement the SSA-compliant C code:
   a. Must write manual implementations of the SSA transformations.
   b. Can verify against `baseline.c` [from lab_initial/examples/basic-ssa].
   c. Store all ssa versions in their own folder [i.e., task00/ssa-versions/].
   d. Ensure consistent naming across versions [i.e., `ssa-var00`].
3. Verify correctness:
   a. Run `run-all-verify.sh`. (WIP)
4. Benchmark performance:
   a. Run `run-all-measure.sh`. (WIP)
5. Compare SSA implementations:
   a. Compare the performance across various operations (such as blur, conv, jacobi2d, etc.).
   b. Identify which operations benefit the most from SSA transformation.
6. Clean up generated files:
   a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

Benchmarking

T01: Handspun + Unroll

Implement, Verify, Benchmark, and Compare for Five (5) of the Operation Specifications in SSA form.

Implementation

Steps:

1. If not already done, convert each operation into SSA form (see objective-a T00 for more details):

a. Each computation must be assigned to a unique variable.
b. <u>No variable reassignment is allowed!</u>

2. **Apply loop unrolling:**
   Unrolling reduces loop overhead by manually duplicating loop iterations.
   a. Select any (5) operations [from /lab_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
   b. Manually unroll the inner loop:
      i. Identify which loop needs to be unrolled (either j0, q0, or r0 for example from slides)
      ii. Example given in the slides for this lab, specifically slide #6.

3. **Implement the SSA-compliant unrolled C code:**
   i. Store all unrolled implementations in their own folder [i.e., /task01/unrolled-versions/].
   ii. Ensure consistent naming across versions [i.e., `unroll-var00`].

4. **Verify correctness:**
   a. Run `run-all-verify.sh`. (WIP)
   b. Ensure unrolling does not introduce any unexpected behavior.

5. **Benchmark performance:**
   a. Run `run-all-measure.sh`. (WIP)
      i. Unrolled SSA vs. non-unrolled SSAx
      ii. Unrolled SSA vs. Baseline (non-SSA)

6. **Compare results:**
   a. Compare the execution time, FLOPs, and memory usage for:
      i. Unrolled SSA vs. non-unrolled SSA
      ii. Unrolled SSA vs. Baseline (non-SSA)
      iii. Which operations benefit the most from loop unrolling?
      iv. Are there any cases in which unrolling hurts performance?

7. **Clean up generated files:**
   a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

T02: Handspun + Split

Implement, Verify, Benchmark, and Compare for Five (5) of the Operation Specifications in SSA form.

Implementation

Steps:

1. If not already done, convert each operation into SSA form (see objective-a T00 for more details):
    a. Each computation must be assigned to a unique variable.
    b. No variable reassignment is allowed!
2. Apply loop splitting:
   Splitting reduces iteration overhead and improves cache locality by breaking a single loop into two nested loops.
    a. Select any (5) operations [from /lab_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
    b. Manually split the loops:
        i. Identify which loops need to be split (either j0, q0, or r0 for example from slides)
        ii. Example given in the slides for this lab, specifically slide #7.

3. Implement the SSA-compliant split C code:
    i. Store all split implementations in their own folder [i.e., /task02/split-versions/].
    ii. Ensure consistent naming across versions [i.e., `split-var00`].
4. Verify correctness:
    a. Run `run-all-verify.sh`. (WIP)
    b. Ensure loop splitting does not introduce any unexpected behavior.
5. Benchmark performance:
    a. Run `run-all-measure.sh`. (WIP)

i. Split SSA vs. non-split SSA
ii. Split SSA vs. Baseline (non-SSA)

6. Compare results:
    a. Compare the execution time, FLOPs, and memory usage for:
        i. Split SSA vs. non-split SSA
        ii. Split SSA vs. Baseline (non-SSA)
        iii. Which operations benefit the most from loop splitting?
        iv. Are there any cases in which splitting hurts performance perhaps by introducing overhead?
7. Clean up generated files:
    a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

T03: Handspun + Interchange

Implement, Verify, Benchmark, and Compare for One (1) Operation against six (6) different ways to interchange the loops in SSA form.

Implementation

Steps:

1. If not already done, convert the operation into SSA form (see objective-a T00 for more details):
    a. Each computation must be assigned to a unique variable.
    b. <u>No variable reassignment is allowed!</u>
2. Apply loop interchange in (6) different ways:
    Loop interchange affects both cache locality and execution efficiency by changing the order of nested loops.
    a. Select any (1) operation [from /lab_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
    b. Manually interchange the loops:
        i. Identify which loops need to be interchanged.
        ii. Example given in the slides for this lab, specifically slide #8.

3. Implement the SSA-compliant split C code:
    i. Store all interchange implementations in their own folder [i.e., /task03/interchange-versions/].
    ii. Ensure consistent naming across versions [i.e., `ssa-interchange-var00`].
4. Verify correctness:
    a. Run `run-all-verify.sh`. (WIP)
    b. Ensure loop interchanges do not introduce any unexpected behavior.
5. Benchmark performance:
    a. Run `run-all-measure.sh`. (WIP)
        i. Each interchanged version vs. baseline SSA
        ii. Each interchanged version vs. other interchanged versions
6. Compare results:
    a. Compare the execution time, FLOPs, and memory usage for:
        i. Each interchanged version vs. baseline SSA
        ii. Which interchanges give the most optimal results?
        iii. Are there any cases in which interchanging worsens performance?
7. Clean up generated files:
    a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

T04: Handspun + Complex Schedule (3 Commands)

Implement, Verify, Benchmark, and Compare for three (3) of the Operation Specifications using the same schedule in SSA form.

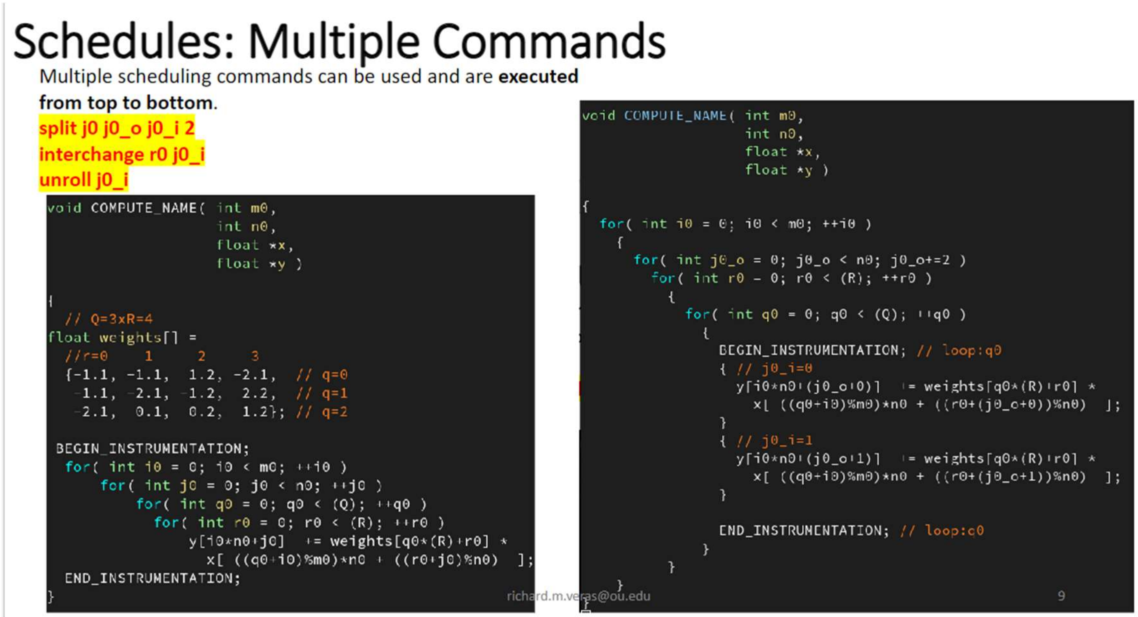
Implementation

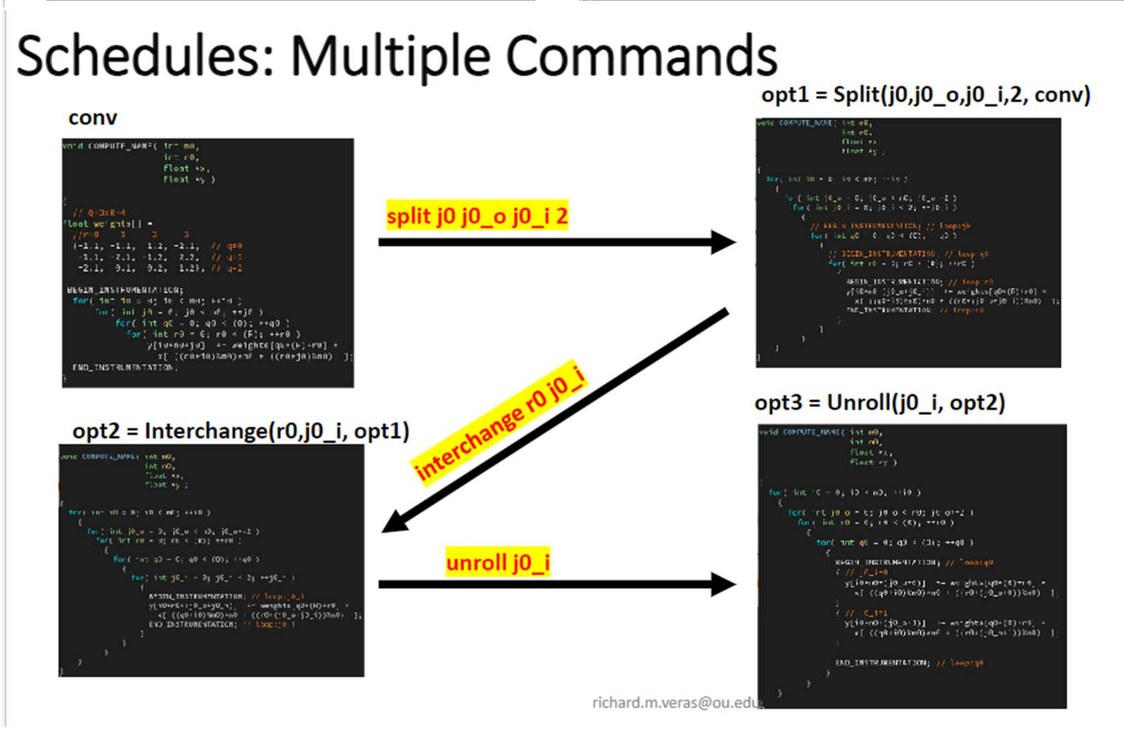
Steps:

1. Before applying complex scheduling, ensure the (3) desired operations are in SSA form (see objective-a T00 for more details):
    a. Each computation must be assigned to a unique variable.
    b. <u>No variable reassignment is allowed!</u>
2. Either define or utilize premade complex schedule:
   A complex schedule applies (in this case, 3) transformations to the loop structure (which could be any combination of: unrolling, splitting, interchanging, etc.).
   Predefined schedules can be found in /lab_initial/schedules/complex_mix/. *The first two options in this folder are (3) command schedules.*

   Example of a (3) command complex schedule:
    a. Apply loop splitting: split j0 into j0_outer and j0_inner
    b. Apply loop interchange: swap q0 and j0_outer
    c. Apply loop unrolling: unroll j0_inner by a factor of 2
3. Apply the complex schedule to (3) operations:
    a. Select any (3) operations [from /lab_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
    b. Apply the same transformation sequence across all (3) operations.

c. Example given in the slides for this lab, specifically slide #9 and slide #10.





4. Implement SSA-compliant scheduled code:
    a. Store all scheduled implementations in their own folder [i.e., /task04/scheduled-versions/].
    b. Ensure consistent naming across versions [i.e., `ssa-schedule-blur3x3.c`].
5. Verify correctness:
    a. Run `run-all-verify.sh`. (WIP)
    b. Ensure complex schedule does not introduce any unexpected behavior.
6. Benchmark performance:
    a. Run `run-all-measure.sh`. (WIP)

            i. Scheduled SSA vs. non-scheduled SSA
            ii. Scheduled SSA vs. baseline (non-SSA)

7. **Compare results:**
    a. Compare the execution time, FLOPs, and memory usage for:
        i. Scheduled SSA vs. non-scheduled SSA
        ii. Scheduled SSA vs. baseline (non-SSA)
        iii. Which combinations of transformations performs the best?
        iv. Are there any cases in which a specific operation benefits more from the schedule?
8. **Clean up generated files:**
    a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

Optional: For Additional Points

T01-Opt00: Handspun + Unroll(r) vs Unroll(q) [Optional]

Implement, Verify, Benchmark, and Compare for One (1) Operation, but two (2) different loops in SSA form.

T02-Opt00: Handspun + Split(i) vs Split(j) vs Split(q) vs Split(r) [Optional]

Implement, Verify, Benchmark, and Compare for One (1) Operation, but four (4) different loops to be split in SSA form.

T02-Opt01: Handspun + Split(2) vs Split(4) vs Split(8) [Optional]

Implement, Verify, Benchmark, and Compare for One (1) Operation, One (1) loop, but four (4) different split factors) in SSA form.

T04-Opt00: Handspun + Complex Schedule (3 Commands) [Optional]

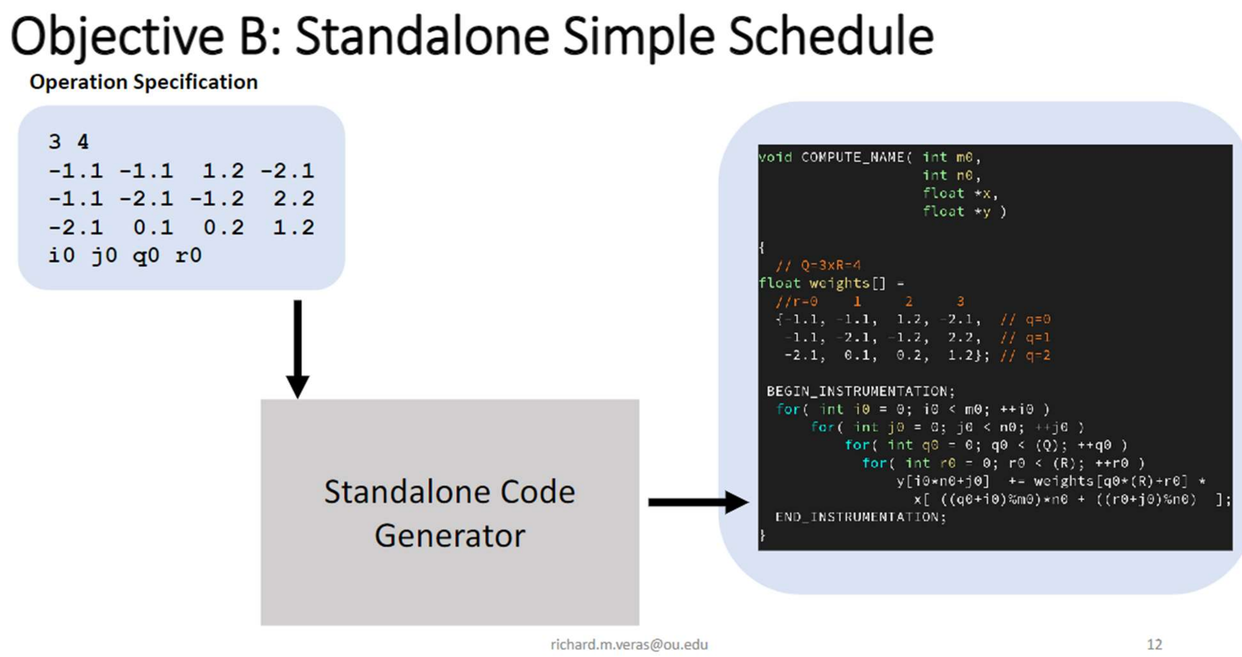
Implement, Verify, Benchmark, and Compare for one (1) of the Operation Specification using three (3) different schedules in SSA form.

Objective B: Standalone Simple Schedule Code Generator

The goal of this objective is to focus on automating what was previously done manually in objective-a.

T00: Simple Standalone Baseline

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications in SSA form.



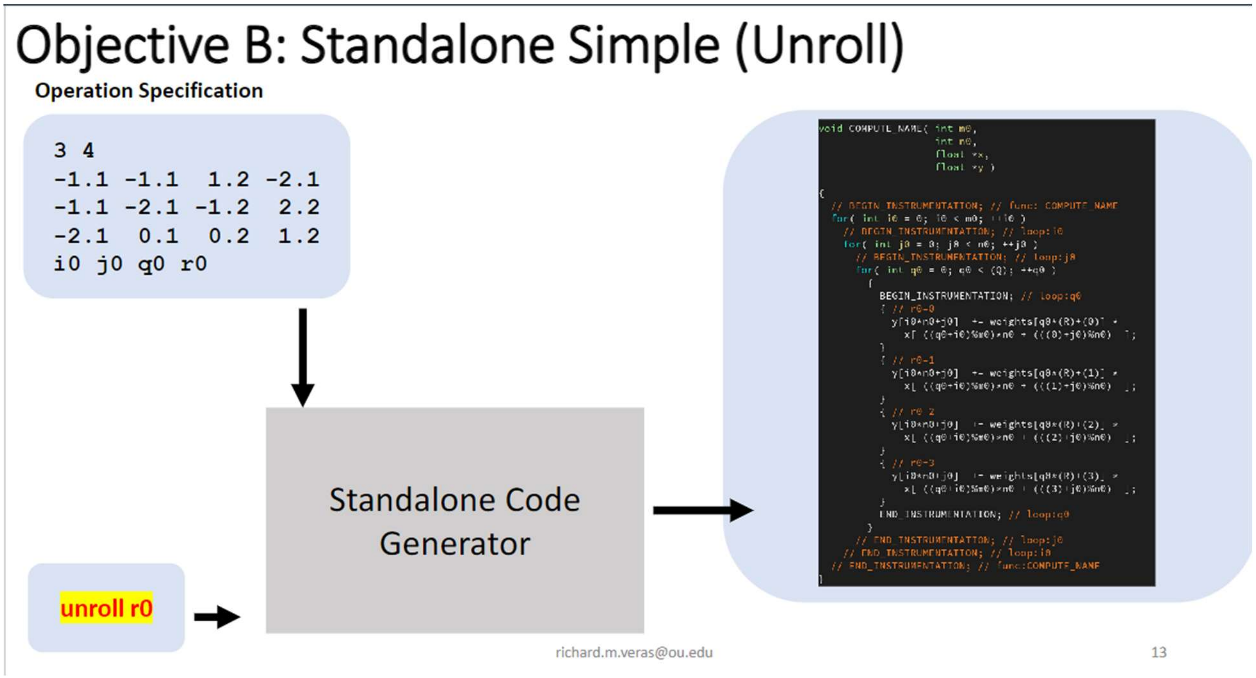
Implementation

Steps:

1. Instead of manually writing SSA-compliant code, we will instead:
    a. Write a generator that takes in an operation description and a simple schedule, and
    b. Automatically generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
    a. Read in an operation file.
    b. Read an optional scheduling file.
    c. Generate an SSA-compliant C code implementation.
    d. Ensure SSA compliance.
    e. Handles basic scheduling, if included.
    f. Preserves the original operation logic.
3. Verify correctness:
    a. Run `run-all-verify.sh`. (WIP)
4. Benchmark performance:
    a. Run `run-all-measure.sh`. (WIP)
        i. Generated SSA vs. Handwritten SSA
        ii. Generated SSA vs. baseline (non-SSA)
5. Compare results:
    a. Generated SSA vs. Handwritten SSA
    b. Generated SSA vs. baseline (non-SSA)
    c. Are there any discrepancies between manually written and generated SSA versions?
    d. Does SSA that is generated produce unnecessary overhead?
6. Clean up generated files:
    a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

T01: Simple Standalone + Unroll

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications for All relevant schedule specifications in SSA form.



Implementation

Modify the code generator to support unrolling.

Steps:

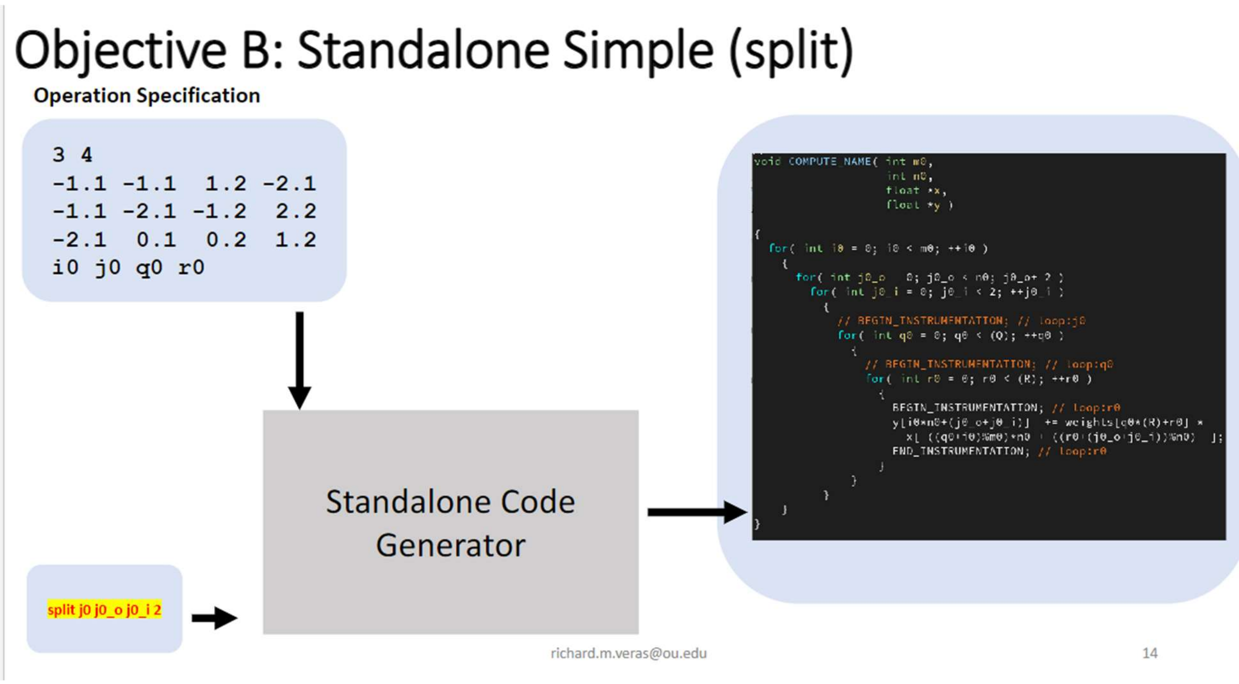
1. Instead of manually writing SSA-compliant code, we will instead:
    a. Write a generator that takes in an operation description and a simple unrolling schedule, and
    b. Automatically apply unrolling to loops then generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
    a. Read in schedule file.
    b. Generate an SSA-compliant C code implementation with modified loop structure according to schedule.
    c. Ensure SSA compliance.
    d. Preserves the original operation logic.
    e. Store all unrolled SSA implementations in their own folder [i.e., /unrolled-versions/].
    f. Ensure consistent naming across versions [i.e., ` ssa_unroll_blur3x3.c`].
3. Verify correctness:
    a. Run `run-all-verify.sh`. (WIP)
4. Benchmark performance:
    a. Run `run-all-measure.sh`. (WIP)

i. Unrolled SSA vs. non-unrolled SSA
ii. Unrolled SSA vs. baseline (non-SSA)

5. **Compare results:**
    a. Unrolled SSA vs. non-unrolled SSA
    b. Unrolled SSA vs. baseline (non-SSA)
6. **Clean up generated files:**
    a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

T02: Simple Standalone + Split

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications for All relevant schedule specifications in SSA form.



Implementation

Modify the code generator to support splitting.

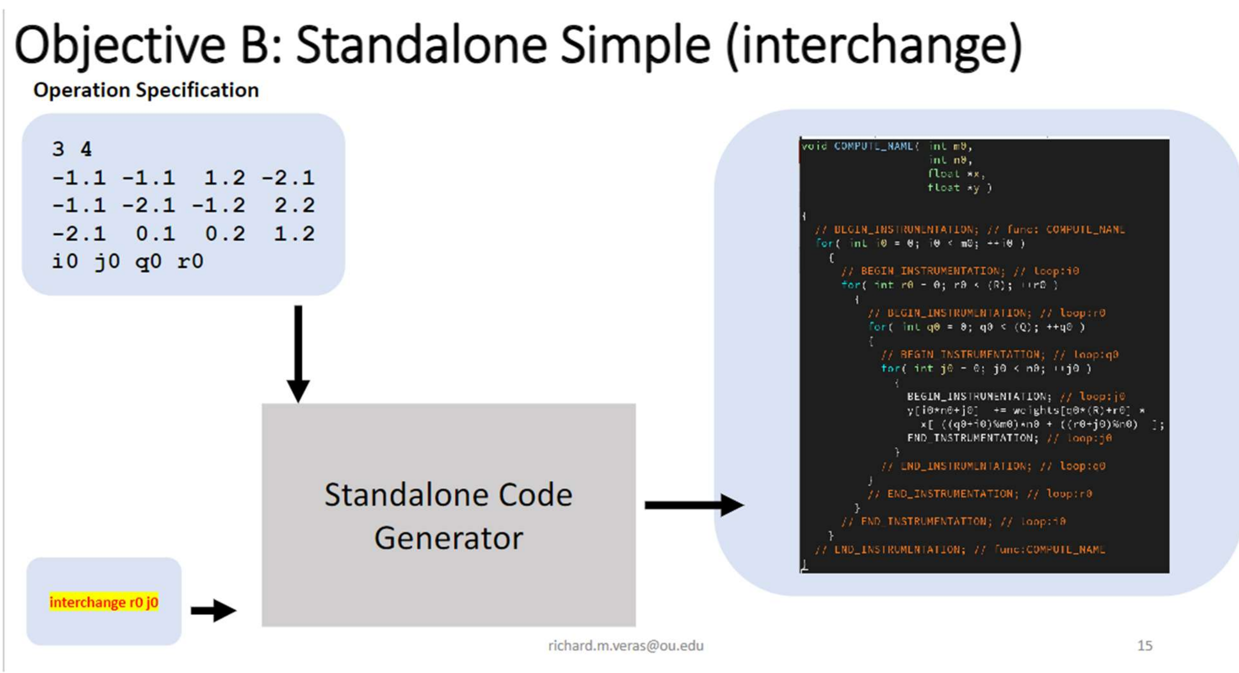
Steps:

1. Instead of manually writing SSA-compliant code, we will instead:
    a. Write a generator that takes in an operation description and a simple splitting schedule, and
    b. Automatically apply splitting to loops then generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
    a. Read in schedule file.
    b. Generate an SSA-compliant C code implementation with modified loop structure according to schedule.
    c. Ensure SSA compliance.

   d. Preserves the original operation logic.
   e. Store all split SSA implementations in their own folder [i.e., /split-versions/].
   f. Ensure consistent naming across versions [i.e., ` ssa_split_blur3x3.c`].
3. Verify correctness:
   a. Run `run-all-verify.sh`. (WIP)
4. Benchmark performance:
   a. Run `run-all-measure.sh`. (WIP)
      i. Split SSA vs. non-split SSA
      ii. Split SSA vs. baseline (non-SSA)
5. Compare results:
   a. Split SSA vs. non-split SSA
   b. Split SSA vs. baseline (non-SSA)
6. Clean up generated files:
   a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

T03: Simple Standalone + Interchange

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications for All relevant schedule specifications in SSA form.



Implementation

Modify the code generator to support loop interchanging.

Steps:

1. Instead of manually writing SSA-compliant code, we will instead:
   a. Write a generator that takes in an operation description and a simple interchanging schedule, and

   b. Automatically apply interchanging to loops then generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
   a. Read in schedule file.
   b. Generate an SSA-compliant C code implementation with modified loop structure according to schedule.
   c. Ensure SSA compliance.
   d. Preserves the original operation logic.
   e. Store all interchanged SSA implementations in their own folder [i.e., /interchanged-versions/].
   f. Ensure consistent naming across versions [i.e., ` ssa_interchange_blur3x3.c`].
3. Verify correctness:
   a. Run `run-all-verify.sh`. (WIP)
4. Benchmark performance:
   a. Run `run-all-measure.sh`. (WIP)
      i. Interchanged SSA vs. non-interchanged SSA
      ii. Interchanged SSA vs. baseline (non-SSA)
5. Compare results:
   a. Interchanged SSA vs. non-interchanged SSA
   b. Interchanged SSA vs. baseline (non-SSA)
6. Clean up generated files:
   a. Run `run-clean-all.sh` to clean up all files generated by a given area. (WIP)

Objective C: EoC Simple Schedule Code Generator

T00: Implement AST to C codegen

Implement an abstract syntax tree (AST) to C code generator, and then test its correctness.

Implementation

T01: Implement Operation Specification to AST

Implement and test.

T02: Implement the Compiler Pass (AST to AST) for Implementing Unroll

Implement and test.

T03: Implement the Compiler Pass (AST to AST) for Implementing Split

Implement and test.

T04: Implement the Compiler Pass (AST to AST) for Implementing Interchange

Implement and test.

T05: EoC Standalone Baseline

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications in SSA form.

T06: EoC Standalone + Unroll

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications for All relevant schedule specifications in SSA form.

T07: EoC Standalone + Split

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications for All relevant schedule specifications in SSA form.

T08: EoC Standalone + Interchange

Implement, Verify, Benchmark, and Compare for All of the Operation Specifications for All relevant schedule specifications in SSA form.

Objective D: Complex Schedule Code Generator

T00: Implement Generalized Schedule (AST to AST)

T01: Complex Schedule Generator + Unroll + Unroll

T02: Complex Schedule Generator + Split + Split + Split + Split

T03: Complex Schedule Generator + Interchange + Interchange + Interchange + Interchange

T04: Complex Schedule Generator + Mixed Schedule (length 3)

T05: Complex Schedule Generator + Mixed Schedule (length 4)

T06: Complex Schedule Generator + Mixed Schedule (length 5)

Objective E: Sensitivity Analysis

T00: Analysis of Complex Schedule (length 3)

T01: Compare the Same Schedule Across Different Operations

T02: Sensitivity Study (vary the factor of a loop split)

T03: Sensitivity Study (vary architecture)

T04: Analysis of Complex Schedule (length 4)

T05: Compare the Same Schedule Across Different Operations

T06: Sensitivity Study (vary the factor of a loop split)

T07: Sensitivity Study (vary architecture)

T08: Analysis of Complex Schedule (length >4)

T09: Compare the Same Schedule Across Different Operations

T10: Sensitivity Study (vary the factor of a loop split)

T11: Sensitivity Study (vary the factor of a loop split for two split factors)

T12: Sensitivity Study (vary architecture)

T13: Post your best performance plot and schedule on Canvas

Optional: Repeats for Additional Points

T08-Opt00: Analysis of Complex Schedule (length >4)

T09-Opt00: Compare the Same Schedule Across Different Operations

T10-Opt00: Sensitivity Study (vary the factor of a loop split)

T11-Opt00: Sensitivity Study (vary the factor of a loop split for two split factors)

T12-Opt00: Sensitivity Study (vary architecture)

T13-Opt00: Post your best performance plot and schedule on Canvas

Utilities & Testing