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:  <op\_name>\_<transformation>\_<loopinfo>\_v<varID>.c   |  |  |  | | --- | --- | --- | | **Part** | **Example** | **Notes** | | <op\_name> | conv4x3, blur2x2 | From `.operation` file | | <transformation> | ssa, base, unroll, split, interchange, etc. | Describes what’s applied | | <loopinfo> | Ijqr, jiqr, split\_i0\_unroll, etc. | [Optional] For loop order or split detail | | v<varID> | v00, v01, etc. | Matches `/var00/`, `/var01/`, etc. |   i.e., |
|  |  |
|  |  |

## Helpful Notes

### Schedules and Operations: Compatibility at a Glance

A screenshot of a computer

AI-generated content may be incorrect.

# 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.

## Tentative Delegation of Tasks

|  |  |
| --- | --- |
| Coleman Bixler | T03 |
| Garrett Snow | T01 |
| An Vu | T04 |
| Nathan Stephani | T02 |
| Luis Arcos | T00 |

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

### 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):
   1. Each computation must be assigned to a unique variable.
   2. No variable reassignment is allowed!
2. Apply loop unrolling:  
   Unrolling reduces loop overhead by manually duplicating loop iterations.
   1. Select any (5) operations [from /lab\_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
   2. Manually unroll the inner loop:
      1. Identify which loop needs to be unrolled (either j0, q0, or r0 for example from slides)
      2. Example given in the slides for this lab, specifically slide #6.  
         **A screenshot of a computer program

         AI-generated content may be incorrect.**
3. Implement the SSA-compliant unrolled C code:
   * 1. Store all unrolled implementations in their own folder [i.e., /task01/unrolled-versions/].
     2. Ensure consistent naming across versions [i.e., `unroll-var00`].
4. Verify correctness:
   1. Run `run-all-verify.sh`.
   2. Ensure unrolling does not introduce any unexpected behavior.
5. Benchmark performance:
   1. Run `run-all-measure.sh`.
      1. Unrolled SSA vs. non-unrolled SSAx
      2. Unrolled SSA vs. Baseline (non-SSA)
6. Compare results:
   1. Compare the execution time, FLOPs, and memory usage for:
      1. Unrolled SSA vs. non-unrolled SSA
      2. Unrolled SSA vs. Baseline (non-SSA)
      3. Which operations benefit the most from loop unrolling?
      4. Are there any cases in which unrolling hurts performance?
7. Clean up generated files:
   1. Run `run-clean-all.sh` to clean up all files generated by a given area.

## 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):
   1. Each computation must be assigned to a unique variable.
   2. 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.
   1. Select any (5) operations [from /lab\_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
   2. Manually split the loops:
      1. Identify which loops need to be split (either j0, q0, or r0 for example from slides)
      2. Example given in the slides for this lab, specifically slide #7.  
         **A screenshot of a computer code

         AI-generated content may be incorrect.**
3. Implement the SSA-compliant split C code:
   * 1. Store all split implementations in their own folder [i.e., /task02/split-versions/].
     2. Ensure consistent naming across versions [i.e., `split-var00`].
4. Verify correctness:
   1. Run `run-all-verify.sh`. (WIP)
   2. Ensure loop splitting does not introduce any unexpected behavior.
5. Benchmark performance:
   1. Run `run-all-measure.sh`. (WIP)
      1. Split SSA vs. non-split SSA
      2. Split SSA vs. Baseline (non-SSA)
6. Compare results:
   1. Compare the execution time, FLOPs, and memory usage for:
      1. Split SSA vs. non-split SSA
      2. Split SSA vs. Baseline (non-SSA)
      3. Which operations benefit the most from loop splitting?
      4. Are there any cases in which splitting hurts performance perhaps by introducing overhead?
7. Clean up generated files:
   1. Run `run-clean-all.sh` to clean up all files generated by a given area.

## 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):
   1. Each computation must be assigned to a unique variable.
   2. No variable reassignment is allowed!
2. Apply loop interchange in (6) different ways:  
   Loop interchange affects both cache locality and execution efficiency by changing the order of nested loops.
   1. Select any (1) operation [from /lab\_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
   2. Manually interchange the loops:
      1. Identify which loops need to be interchanged.
      2. Example given in the slides for this lab, specifically slide #8.  
         **A screenshot of a computer program

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

## 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

**Basic Steps:**

1. **Setup:** Pick *any* (3) `.operation` files from `/lab\_initial/operations/` which are compatible with the schedule you have chosen to implement (from `/lab\_initial/schedules/`). See [here](#_Schedules_and_Operations:) for rough compatibility.  
   **For example:** you might choose `complex\_0.schedule` which should be compatible with `blur\_3x3\_ijqr`, `jacobi2d\_5pt\_ijqr`, and `Toeplitz\_4x4\_ijqr`, and so on…
2. **SSA Transformation:** Convert each operation into a manually written `.c` file, which should be in SSA form.   
   For each operation:
   1. Inspect the operation kernel weights.
   2. Write a `COMPUTE\_NAME()` C function that:
      1. Iterates through the loop order from the chosen `.operation` file.
      2. *Has no variable reassignments*; every temporary has a unique name.
   3. **Use `ssa\_var000.c` as a reference.**
3. **Apply the Schedule by Hand:** Use the (3) scheduling commands from within the chosen schedule and transform the loop nest accordingly.  
   **For example:**   
   **A screenshot of a computer

   AI-generated content may be incorrect.  
   \*See slides 9 and 10 below for further clarification and details.**
4. **Save and Name the Outputs:** Place the generated `.c` file(s) in: ` lab/objectives/objective-a/task04/scheduled-versions/`and use clear naming conventions such as ` ssa-scheduled-blur3x3.c`.
5. **Verify:** Run the shell script `run-all-verify.sh` once it is fully implemented
6. **Benchmark:** Run the shell script `run-all-measure.sh` once it is fully implemented
7. **Compare and Analyze:**Collect performance differences between:
   1. Scheduled SSA vs. non-scheduled SSA
   2. Scheduled SSA vs. baseline (T00)
8. **Clean Things Up:** Run the shell script

**Detailed Steps:**

1. Before applying complex scheduling, ensure the (3) desired operations are in SSA form (see objective-a T00 for more details):
   1. Each computation must be assigned to a unique variable.
   2. No variable reassignment is allowed!
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:
   1. Apply loop splitting: split j0 into j0\_outer and j0\_inner
   2. Apply loop interchange: swap q0 and j0\_outer
   3. Apply loop unrolling: unroll j0\_inner by a factor of 2
3. Apply the complex schedule to (3) operations:
   1. Select any (3) operations [from /lab\_initial/operations/]. (For example: blur, conv, jacobi2d, etc.)
   2. Apply the same transformation sequence across all (3) operations.
   3. Example given in the slides for this lab, specifically slide #9 and slide #10.  
      **A screenshot of a computer program

      AI-generated content may be incorrect.**  
      **A diagram of a computer code

      AI-generated content may be incorrect.**
4. Implement SSA-compliant scheduled code:
   1. Store all scheduled implementations in their own folder [i.e., /task04/scheduled-versions/].
   2. Ensure consistent naming across versions [i.e., `ssa-schedule-blur3x3.c`].
5. Verify correctness:
   1. Run `run-all-verify.sh`.
   2. Ensure complex schedule does not introduce any unexpected behavior.
6. Benchmark performance:
   1. Run `run-all-measure.sh`.
      1. Scheduled SSA vs. non-scheduled SSA
      2. Scheduled SSA vs. baseline (non-SSA)
7. Compare results:
   1. Compare the execution time, FLOPs, and memory usage for:
      1. Scheduled SSA vs. non-scheduled SSA
      2. Scheduled SSA vs. baseline (non-SSA)
      3. Which combinations of transformations performs the best?
      4. Are there any cases in which a specific operation benefits more from the schedule?
8. Clean up generated files:
   1. Run `run-clean-all.sh` to clean up all files generated by a given area.

## 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.

For this objective we will only focus on simple schedules still and additionally only those with one line, for example:   
```  
unroll j0 2

split j0 j0\_outer j0\_inner 4

interchange j0 q0  
```

For this objective we do not yet want to mess with handling complex schedules.

## Tentative Delegation of Tasks

|  |  |
| --- | --- |
| Coleman Bixler | T03 |
| Garrett Snow | T01 |
| An Vu | T00 |
| Nathan Stephani | T02 |

## T00: Simple Standalone Baseline

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

A diagram of a computer code generator

AI-generated content may be incorrect.

### Implementation

**Steps:**

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

## T01: Simple Standalone + Unroll

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

A diagram of a computer code

AI-generated content may be incorrect.

### Implementation

Modify the code generator to support unrolling.

**Steps:**

1. Instead of manually writing SSA-compliant code, we will instead:
   1. Write a generator that takes in an operation description and a simple unrolling schedule, and
   2. Automatically apply unrolling to loops then generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
   1. Read in schedule file.
   2. Generate an SSA-compliant C code implementation with modified loop structure according to schedule.
   3. Ensure SSA compliance.
   4. Preserves the original operation logic.
   5. Store all unrolled SSA implementations in their own folder [i.e., /unrolled-versions/].
   6. Ensure consistent naming across versions [i.e., ` ssa\_unroll\_blur3x3.c`].
3. Verify correctness:
   1. Run `run-all-verify.sh`.
4. Benchmark performance:
   1. Run `run-all-measure.sh`.
      1. Unrolled SSA vs. non-unrolled SSA
      2. Unrolled SSA vs. baseline (non-SSA)
5. Compare results:
   1. Unrolled SSA vs. non-unrolled SSA
   2. Unrolled SSA vs. baseline (non-SSA)
6. Clean up generated files:
   1. Run `run-clean-all.sh` to clean up all files generated by a given area.

## T02: Simple Standalone + Split

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

A diagram of a computer code

AI-generated content may be incorrect.

### Implementation

Modify the code generator to support splitting.

**Steps:**

1. Instead of manually writing SSA-compliant code, we will instead:
   1. Write a generator that takes in an operation description and a simple splitting schedule, and
   2. Automatically apply splitting to loops then generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
   1. Read in schedule file.
   2. Generate an SSA-compliant C code implementation with modified loop structure according to schedule.
   3. Ensure SSA compliance.
   4. Preserves the original operation logic.
   5. Store all split SSA implementations in their own folder [i.e., /split-versions/].
   6. Ensure consistent naming across versions [i.e., ` ssa\_split\_blur3x3.c`].
3. Verify correctness:
   1. Run `run-all-verify.sh`.
4. Benchmark performance:
   1. Run `run-all-measure.sh`.
      1. Split SSA vs. non-split SSA
      2. Split SSA vs. baseline (non-SSA)
5. Compare results:
   1. Split SSA vs. non-split SSA
   2. Split SSA vs. baseline (non-SSA)
6. Clean up generated files:
   1. Run `run-clean-all.sh` to clean up all files generated by a given area.

## T03: Simple Standalone + Interchange

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

A diagram of a computer code

AI-generated content may be incorrect.

### Implementation

Modify the code generator to support loop interchanging.

**Steps:**

1. Instead of manually writing SSA-compliant code, we will instead:
   1. Write a generator that takes in an operation description and a simple interchanging schedule, and
   2. Automatically apply interchanging to loops then generate SSA-compliant C code as the output.
2. Implement the simple standalone code generator:
   1. Read in schedule file.
   2. Generate an SSA-compliant C code implementation with modified loop structure according to schedule.
   3. Ensure SSA compliance.
   4. Preserves the original operation logic.
   5. Store all interchanged SSA implementations in their own folder [i.e., /interchanged-versions/].
   6. Ensure consistent naming across versions [i.e., ` ssa\_interchange\_blur3x3.c`].
3. Verify correctness:
   1. Run `run-all-verify.sh`.
4. Benchmark performance:
   1. Run `run-all-measure.sh`.
      1. Interchanged SSA vs. non-interchanged SSA
      2. Interchanged SSA vs. baseline (non-SSA)
5. Compare results:
   1. Interchanged SSA vs. non-interchanged SSA
   2. Interchanged SSA vs. baseline (non-SSA)
6. Clean up generated files:
   1. Run `run-clean-all.sh` to clean up all files generated by a given area.

# Objective C: EoC Simple Schedule Code Generator

## Tentative Delegation of Tasks

|  |  |
| --- | --- |
| Coleman Bixler | T04,T08 |
| Garrett Snow | T02,T06 |
| An Vu | T01(?),T05(?) |
| Nathan Stephani | T03,T07 |

## 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

This objective extends the compiler infrastructure to support **complex schedules**, which are sequences of multiple transformation commands (e.g., split + interchange + unroll). The work here builds on prior AST-based infrastructure from Objective C and emphasizes correct sequencing and robustness.

## 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