Shankar Kailas – 3036861618

CS 265

Group: Shankar Kailas

GitHub Link: https://github.com/shkailas/bril

(All optimization scripts can be found in bril/examples/local_optimizations)

Summary of Accomplishments:

In this task, I accomplished the following tasks:

1. Global (trivial) Dead Code Elimination
2. Local Dead Code Elimination
3. Local Value Numbering
    a. Common Subexpression Elimination
    b. Copy Propagation
    c. Constant Propagation
    d. Constant Folding

Algorithms and Design Decisions:

1. Global Dead Code Elimination

    Global DCE was implemented almost identically to how it was demonstrated in class. There was very little variation or room for design decisions. To ensure layers of global dead code were removed, the global DCE algorithm was run until no new lines of code were removed. In terms of runtime optimality, a flag variable was used to mark whether there had been any changes in the program to determine if convergence was reached. Due to the use of python list comprehension, it was difficult to determine if instructions were removed (i.e. convergence was reached) without checking if the program before an iteration of global DCE was different than the program after. In hindsight, perhaps checking just the number of instructions instead of a list comparison may have been more efficient, but correctness remains the same.

    Code Snippet:

```
# enter each (main) function
for fn in prog["functions"]:

    # record prior to an iteration to see if there has been a change
    flag = True
    while flag:
        old = fn["instrs"]
        # create a set of variables who's use we will monitor
        used = set()
        #iterate through the instructions backwards
        #find all the variables needed for the print and remove ops that are not relevant to them
        for instr in fn["instrs"]:
            if "op" in instr and "args" in instr:
                for arg in instr["args"]:
                    used.add(arg)
        for fn in prog["functions"]:
            fn["instrs"] = [instr for instr in fn["instrs"] if not ("dest" in instr and instr["dest"] not in used)]
```

```
        if fn["instrs"] == old:
            flag = False
```

2. Local Dead Code Elimination

Similar to global DCE, local DCE also very strictly adhered to the classroom implementation. The first step in this process was to create the maximal basic blocks. This was borrowed from the pre-built form_blocks function provided. As taught, local DCE was run on each block. Arguments in an instruction were handled before destinations so that unused instructions where a variable appeared in both the arguments and destination were handled appropriately. To ensure layers of local dead code were removed from each block, the local DCE algorithm was run until no new lines of code were removed per block. Some notable design decisions included indexing the instructions when iterating through a block. This made it easier to make sure that the correct instruction was deleted (and not another instruction that may have been identical but in a different location). Furthermore, to ensure this, instruction indices were stored until the end of a pass through a block, then sorted and removed in descending order so as to preserve the previous instruction indices. Finally, since the order of the blocks after local DCE was maintained, converting the modified blocks back to instruction format was as simple as combining all of the instructions in the order they appeared in the block, in block order.

Code Snippet:

```
# enter each (main) function

for fn in prog["functions"]:
#for fn in range (len(prog["functions"])):
    blocks = form_blocks(fn['instrs'])


    new_blocks = []

    for block in blocks:
        # record prior to an iteration to see if there has been a change
        flag = True
        while flag:
            flag = False
            old = block
            unused = {}

            # instructions that need to be removed
            inst_to_be_removed = []

            # If the variable is used,
            for i in range(0, len(block)): # for instruction block[i]

                # if its used, its not unused
                if "args" in block[i]:
                    for argument in block[i]["args"]:

                        unused.pop(argument, None)

                # if the instruction defines something,
                # we can kill the unused definitions
                if "dest" in block[i]:
                    if block[i]["dest"] in unused:
```

```
                        inst_to_be_removed.append(unused[block[i]["dest"]])

                # mark the new def as unused for now
                    unused[block[i]["dest"]] = i

            # now we remove all of the clobbered unused instructions
            # sorting to make deletions not affect each other
            inst_to_be_removed = sorted(inst_to_be_removed, reverse = True)

            for index in inst_to_be_removed:
                del block[index]
                flag = True

        new_blocks.append(block)
    fn["instrs"] = [item for sublist in new_blocks for item in sublist]
```

3. Local Value Numbering

a. Common Subexpression Elimination

The first step in this process was to create the maximal basic blocks. This was again borrowed from the pre-built form_blocks function provided. The common subexpression elimination code was written to adhere to the class pseudocode. The first major design decision was to write a line that essentially eliminated control flow statements and print statements from consideration. This was done under the assumption that instructions of this type typically do not involve non-singular expressions (where this algorithm would not really matter).

```
        for i in range(0, len(block) ): # for instruction block[i]
            if "op" in block[i] and (block[i]["op"] in TERMINATORS  or block[i]["op"] == "print"):
                continue
```

As expected, four maps were used to represent the LVN table, and a uniqueId variable was used whenever new entries were made involving numbering or the creation of temp variables to ensure uniqueness in the keys. Canonicalization was done relatively as expected, with values having the tuple form (<op>, <argument_1_num>, ...), and the creation of “symbolic” entries (for instructions where the arguments were variables that weren’t defined before in the same block) was also handled with the value “symbolic<uniqueId>”.

```
            if "op" in block[i]:
                # if the instr has arguemnts (sum, id, mult, div)
                if "args" in block[i]:
                    #args = [var2num[arg] for arg in block[i]["args"]]
                    args = []
                    for arg in block[i]["args"]:
                        if not (arg in var2num or arg in temp_references): # symbolic case
                            num2val[uniqueId] = "symbolic" + str(uniqueId)
                            val2num["symbolic" + str(uniqueId)] = uniqueId
                            var2num[arg] = uniqueId
                            num2var[uniqueId] = arg

                            uniqueId += 1

                        if arg in temp_references:
                            args.append( var2num[temp_references[arg]] )
                        else:
                            args.append( var2num[arg] )

                    value = tuple([block[i]["op"]] + args)

                elif block[i]["op"] == "const":
                    value = tuple(["const"]+[block[i]["value"]])
```

Then the typical common subexpression elimination was conducted, using the val2num map to see if the subexpression was already computed. Note that only direct matches were optimized for (i.e. no commutativity, partial matches, etc.). If the subexpression was already calculated previously and the variable that it was stored in still had that value, then an id instruction replaced the subexpression instruction, reducing computation.

```
                if value not in val2num:
                    num = uniqueId
                    val2num[value] = num
                    num2val[num] = value
                    uniqueId += 1

                else:
                    num = val2num[value]
                    if num2var[num] in temp_references: # its clobbered
                        pass
                    else: # eliminate-able common subexpression
                        block[i]["op"] = "id"
                        block[i]["args"] = [num2var[num]]
```

To handle clobbered variables, an aggressive approach was taken to introduce temporary variables. This was done partly because I wanted to try and implement it, but mostly to save precomputed entries in the value-number tables so that the computation could be reused (instead of being deleted when the clobbered variable was deleted). Temp variables were created using the naming format "<original_name>_temp<uniqueId>" so that their names would be unique. These temp variable names were also linked to their original names using a temp_references dictionary so that the most recent temp variable could always be utilized when converting from var names to num (see earlier code snippet).

```
                if block[i]["dest"] in var2num:
                    temp_name = block[i]["dest"]+"_temp"+str(uniqueId)
                    uniqueId +=1


                    temp_references[block[i]["dest"]] = temp_name
                    block[i]["dest"] = temp_name

                    var2num[temp_name] = num
                    num2var[num] = temp_name

                else:
                    var2num[ block[i]["dest"] ] = num
                    num2var[num] = block[i]["dest"]
```

Lastly, all of the blocks' instructions were merged back together in a similar manner as was done with local DCE.

b. Copy Propagation

This copy propagation also began with creating maximal blocks. This implementation of copy propagation was written to replace variables who share the same value (at that instruction in the same block) with the variable that appears earliest in the code. This was done in a separate pass from the LVN pass so that its results could be more usefully utilized by the LVN pass. Equivalence between variables was only determined when an "id" instruction was done.

```
for block in blocks:
            equivalences = {}
            for i in range(0, len(block)): # for instruction block[i]
                if "op" in block[i]  and  block[i]["op"] == "id":

                    if block[i]["args"][0] in equivalences:
```

```
                        equiv = equivalences[block[i]["args"][0]]
                        block[i]["args"] = [ equiv ]
                        equivalences[block[i]["dest"]] = equiv
                    else:
                         equivalences[block[i]["dest"]] = block[i]["args"][0]
```

Code was also written to 1. propagate these equivalences through argument variables and 2. to remove equivalence relationships after a variable has been modified.

```
                elif "op" in block[i]  and  block[i]["op"] in operations_that_modify_dest:
                    if "args" in block[i]:
                        block[i]["args"] = [equivalences[arg] if (arg in equivalences) else arg for arg in
block[i]["args"]]
                    equivalences.pop(block[i]["dest"], None)# clobbered
```

Lastly, all of the blocks' instructions were merged back together in a similar manner as was done with local DCE. This implementation of copy propagation was done so that more local dead code would be revealed, since not all the variable assignments are really necessary in the program where previously defined variables would have sufficed.

c. Constant Propagation & Folding

Constant Propagation and Folding were done together and added to the LVN pass. This was done under the belief that constant-related optimizations would not help with subexpression optimizations as much and vice versa, since constant propagation would essentially remove the need for a subexpression anyways. The process boiled down to identifying instructions that could potentially benefit from constant propagation or folding. I specifically avoided division by 0 examples since this I did not want to implement what the optimizer should do if this case was reached. If it was determined that every argument in the instruction could be traced to a constant value, then the instruction was rewritten to be a "const" instruction with the appropriate calculated value using a self-written evaluate_operation function (not shown here).

```
                    const_ops = set([
                        "add", "mul", "div", "eq", "le", "lt", "gt", "ge", "and", "or", "not"
                        ])
                    if block[i]["op"] in const_ops and all( num2val[n][0] == "const" for n in value[1:]) and not
(block[i]["op"] == "div"  and num2val[value[2]][1] == 0):
                        inputs = [num2val[n] for n in value[1:]]
                        folding_result = evaluate_operation(block[i]["op"],  inputs)
                        # change the instruction
                        block[i] = {"dest": block[i]["dest"], "op": "const", "type": "bool" if
isinstance(folding_result, bool) else "int", "value": folding_result}
```

After the instruction was rewritten, the four state tables needed to be modified to reflect these changes and impacts of clobbering (as was done with common subexpression elimination)

```
                        # update the state tables
                        num = uniqueId
                        value = tuple([block[i]["op"]] + [folding_result])
                        val2num[value] = num
                        num2val[num] = value
                        if block[i]["dest"] in var2num:
                            temp_name = block[i]["dest"]+"_temp"+str(uniqueId)
                            temp_references[block[i]["dest"]] = temp_name
                            block[i]["dest"] = temp_name
                            var2num[temp_name] = num
                            num2var[num] = temp_name
                        else:
                            var2num[ block[i]["dest"] ] = num
                            num2var[num] = block[i]["dest"]
```

```
                    uniqueId += 1
                    #continue to next instruction
                    continue
```

Lastly, all of the blocks' instructions were merged back together in a similar manner as was done with local DCE.

Results and Analysis:

To measure the value of these optimizations and their ability to work together, I used the TCDE benchmark suite and the LVN benchmark suite. Below are the different pipelines of optimizations used on each benchmark:

1. TCDE benchmarks
a. Baseline
b. Global Dead Code Elimination (global_dce)
c. Local Dead Code Elimination (global_local_dce)

Test Case	Optimization Level	Result
reassign	baseline	3
reassign	global_dce	3
reassign	global_local_dce	2
combo	baseline	6
combo	global_dce	5
combo	global_local_dce	4
skipped	baseline	4
skipped	global_dce	4
skipped	global_local_dce	4
double	baseline	6
double	global_dce	4
double	global_local_dce	4
simple	baseline	5
simple	global_dce	4
simple	global_local_dce	4
double-pass	baseline	6
double-pass	global_dce	4
double-pass	global_local_dce	4
reassign-dkp	baseline	3
reassign-dkp	global_dce	3
reassign-dkp	global_local_dce	2
diamond	baseline	6
diamond	global_dce	6
diamond	global_local_dce	6

2. LVN Benchmarks (of those that would run)
a. Global/Local Dead Code Elimination (dce)

b. Global/Local Dead Code Elimination + Common Subexpression Elimination (dce_cse)
c. Global/Local Dead Code Elimination + Common Subexpression Elimination + Global/Local Dead Code Elimination (dce_cse_dce)
d. Global/Local Dead Code Elimination + Common Subexpression Elimination + Global/Local Dead Code Elimination (dce_cse_dce)
e. Global/Local Dead Code Elimination + Copy Propagation + Common Subexpression Elimination + Global/Local Dead Code Elimination (dce_copy_cse_dce)
f. Global/Local Dead Code Elimination + Copy Propagation +Constant Propagation + Constant Folding + Common Subexpression Elimination + Global/Local Dead Code Elimination (dce_copy_const_cse_dce)

Note: logical-operators, divide-by-zero, fold-comparisons tests would not run on the brench tool and were run by hand.

logical-operators	baseline	missing
logical-operators	dce	0
logical-operators	dce_cse	0
logical-operators	dce_cse_dce	0
logical-operators	dce_copy_cse_dce	0
logical-operators	dce_copy_const_cse_dce	0
reassign	baseline	3
reassign	dce	2
reassign	dce_cse	2
reassign	dce_cse_dce	2
reassign	dce_copy_cse_dce	2
reassign	dce_copy_const_cse_dce	2
idchain	baseline	5
idchain	dce	5
idchain	dce_cse	5
idchain	dce_cse_dce	5
idchain	dce_copy_cse_dce	5
idchain	dce_copy_const_cse_dce	2
redundant-dce	baseline	6
redundant-dce	dce	6
redundant-dce	dce_cse	6
redundant-dce	dce_cse_dce	6
redundant-dce	dce_copy_cse_dce	6
redundant-dce	dce_copy_const_cse_dce	2
nonlocal-clobber	baseline	4
nonlocal-clobber	dce	4
nonlocal-clobber	dce_cse	4
nonlocal-clobber	dce_cse_dce	3
nonlocal-clobber	dce_copy_cse_dce	3
nonlocal-clobber	dce_copy_const_cse_dce	3
divide-by-zero	baseline	5
divide-by-zero	dce	5

divide-by-zero	dce_cse	5
divide-by-zero	dce_cse_dce	5
divide-by-zero	dce_copy_cse_dce	5
divide-by-zero	dce_copy_const_cse_dce	5
clobber-arg	baseline	3
clobber-arg	dce	3
clobber-arg	dce_cse	3
clobber-arg	dce_cse_dce	0
clobber-arg	dce_copy_cse_dce	0
clobber-arg	dce_copy_const_cse_dce	0
idchain-nonlocal	baseline	6
idchain-nonlocal	dce	6
idchain-nonlocal	dce_cse	6
idchain-nonlocal	dce_cse_dce	6
idchain-nonlocal	dce_copy_cse_dce	6
idchain-nonlocal	dce_copy_const_cse_dce	6
nonlocal	baseline	7
nonlocal	dce	7
nonlocal	dce_cse	7
nonlocal	dce_cse_dce	7
nonlocal	dce_copy_cse_dce	7
nonlocal	dce_copy_const_cse_dce	5
commute	baseline	6
commute	dce	6
commute	dce_cse	6
commute	dce_cse_dce	6
commute	dce_copy_cse_dce	6
commute	dce_copy_const_cse_dce	2
rename-fold	baseline	7
rename-fold	dce	7
rename-fold	dce_cse	7
rename-fold	dce_cse_dce	6
rename-fold	dce_copy_cse_dce	6
rename-fold	dce_copy_const_cse_dce	4
redundant	baseline	6
redundant	dce	6
redundant	dce_cse	6
redundant	dce_cse_dce	6
redundant	dce_copy_cse_dce	6
redundant	dce_copy_const_cse_dce	2
clobber	baseline	10
clobber	dce	5
clobber	dce_cse	5
clobber	dce_cse_dce	5
clobber	dce_copy_cse_dce	5
clobber	dce_copy_const_cse_dce	2

idchain-prop	baseline	5
idchain-prop	dce	5
idchain-prop	dce_cse	5
idchain-prop	dce_cse_dce	5
idchain-prop	dce_copy_cse_dce	5
idchain-prop	dce_copy_const_cse_dce	5
clobber-fold	baseline	10
clobber-fold	dce	5
clobber-fold	dce_cse	5
clobber-fold	dce_cse_dce	5
clobber-fold	dce_copy_cse_dce	5
clobber-fold	dce_copy_const_cse_dce	2
fold-comparisons	baseline	15
fold-comparisons	dce	0
fold-comparisons	dce_cse	0
fold-comparisons	dce_cse_dce	0
fold-comparisons	dce_copy_cse_dce	0
fold-comparisons	dce_copy_const_cse_dce	0

1. TCDE benchmarks Analysis

   The TCDE data reveals that our local and global analysis reduced the number of instructions in almost every benchmark. This was to be expected, since many of these programs were simple and had some obvious dead code in them. Furthermore, running both local and global dead code analyses on these benchmarks by hand revealed that using our classroom environments, these algorithms were working as expected (assuming I did it correctly by hand).

2. LVN benchmarks Analysis

   The LVN data also reveals the potency of compounding each optimization as it was developed. Some of the biggest jumps in instruction count across the board were from 1. no optimizations to local/global DCE 2. local/global DCE to local/global DCE+common subexpression elimination + local/global DCE, and 3. Global/Local Dead Code Elimination + Copy Propagation + Common Subexpression Elimination + Global/Local Dead Code Elimination to Global/Local Dead Code Elimination + Copy Propagation +Constant Propagation + Constant Folding + Common Subexpression Elimination + Global/Local Dead Code Elimination.

   The first was expected, considering the previous result. The second major jump was likely due to how the common subexpression elimination was converting many instructions to id instructions while also rendering those id instructions as dead code since the same result was stored in another variable (also the dataset happened to have lots of these examples).

The last major jump was likely from the constant propagation/folding addition, since this step removed many instructions and simplified them to const instructions. Taken together with the fact that there weren't many basic blocks in these benchmarks, there were likely lots of instructions that became dead code generated by these optimizations (since there were lots of instructions whose arguments were const vars in the same block) , which got cleaned up in the final sweeps of DCE. It was cool to see how the instructions drastically dropped; however, the style of benchmarks contributed to that heavily. That being said, there are definitely times when my copy propagation does not work as intended. Some of the

Beyond a theoretical understanding, these benchmarks also provide some evidence of the correctness of our algorithm in not touching the observable features of the code. In this case, observable refers to the printed output of the program, and every optimization pipeline had the same results as the baseline.

Optimized Programs:

Here is a program that my optimizations do well on. This example features elements of common subexpression elimination, copy propagation, constant propagation and folding, and cleanup by DCE sweeps. It also effectively handles instances of variable clobbering:

```
@main {
  a: int = const 4;
  b: int = const 2;
  c: int = const 2;
  sum1: int = add a b;
  sum2: int = add a b;
  prod1: int = mul sum1 sum2;
  copy1: int = id prod1;
  copy2: int = id copy1;
  copy3: int = id copy2;
  # Clobber both sums.
  sum1: int = const 0;
  sum2: int = const 0;
  sum3: int = add a b;
  prod2: int = mul sum3 sum3;
  print prod2;
.testLabel:
  f: int = const 100;
  f: int = const 42;
  print f;
  f: int = const 4;
  b: int = const 2;
  c: int = const 1;
```

```
  d: int = add f b;
  e: int = add c d;
  print d;
}
```

Optimized version:

```
@main {
  prod2: int = const 36;
  print prod2;
.testLabel:
  f: int = const 42;
  print f;
  d: int = const 6;
  print d;
}
```

Here is a script that is not optimized well. The optimization suffers due to lack of handling commutativity (add a b vs. add b a), lack of optimizations around dataflow (i.e. skipping useless mylabel, sharing state across blocks with only one entry point), and lack of obvious constant folding scenarios that can be hard coded (multiplying by 0). In fact, the “optimized” version does not make any changes except some constant folding:

```
@main {
  a: int = const 36;
  b: int = const 47;
  c: int = add a b;
  print c;
  print d;
  jmp termination;

.dumblabel:
  f: int = add a b;
  d: int = add b a;
  print f;
  g: int = const 0;
  h: int = mul f g;
.termination:
  print a;
}
```

Optimized version:

```
@main {
```

```
  a: int = const 36;
  b: int = const 47;
  c: int = const 83;
  print c;
  print d;
  jmp termination;
.dumblabel:
  f: int = add a b;
  d: int = add b a;
  print f;
.termination:
  print a;
}
```