Shankar Kailas – 3036861618

CS 265

Group: Shankar Kailas

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

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

Summary of Accomplishments:

In this task, I accomplished the following tasks:

1. SSA
2. Loop Normalization
3. Loop Invariant Code Motion

Algorithms and Design Decisions:

1. SSA

In this project, I was able to implement SSA. The SSA used was based on the algorithms taught in lecture. It began with ensuring entry and terminators as with the previous task and creating the basic CFG. From here, one modification I made was immediately removing dead code that followed a terminator (these would appear in blocks that had no predecessors and where not the entry block. From here, I iteratively remove blocks whos' only predecessors were those mentioned above, and I continued this process until I couldn't). After recomputing the new CFG, I computed dominance relations as described in class, the dominator tree, and the dominance frontier (definitions of strict dominance threw me off for a while but eventually I got it working). From here, I added the phi instructions and did renaming/phi-filling as taught in lecture. Unique names were given to variables based on the number of times that variable name had been seen before. To find what blocks defined what variables, I ended up using a pre-built function to save time. Overall, the SSA was successful.

2. Loop Normalization

To save time and split the loop normalization into another file, I simply reran all of the dominance calculation from the SSA step. From here, found all of the back-edges using the dominance relations as taught in lecture. The first normalization I did was the creation of latch blocks. I decided to do this because I thought it would reduce the number and types of loops I would have to worry about, and I thought it would help create a 1-1 relationship between headers and loops. Essentially, I went through all of the back-edges and if there were multiple that returned to the same header, I would change the CFG to make them all point to a newly created latch that jumped to the header. I only created latches if there were multiple back-edges to a header block (if there was only one then I made note that it was a latch). Then, I created the preheaders. This was easier since there was a 1-1 relation between headers and loops now, so all I did was find all of the predecessors of each header that were not latches and had them go to a newly created pre-header, and have the pre-header jump to header. Both of these operations were fine, but I overlooked how the phi instructions would change in response to these normalizations. For each (newly created) latch block, I needed to add phis in the latch so that predecessors of the latch who had previously placed their values directly in the header would put their values in the latchs' phis. Then I had to correct the phis in the headers so that they pulled their values from the latch. I also fixed the phis in the pre-header in a similar way, making all of the previous non-loop related predecessors of each header place values in the preheader phis and modify the header phis to look for the value in the preheader. After these transformations, the code was ready for LICM.

3. Loop Invariant Code Motion

Since my loop normalization had potentially invalidated my CFG and dominance calculations, I redid them again. However, I had created a 1-1-1-1 relation between loops, preheaders, headers, and latches, which made LICM very easy. I also found all of the blocks associated with every loop for the LICM. I sorted loops in reverse order based on the number of blocks in each loop. This was so that nested loops would be processed first, and hoisted code could be removed from the inner before the outer was considered. I also relaxed the assumption that loops may not execute for an easier time hoisting. I hoisted all code that was either a const operation or was a constant operation (i.e. "add", "sub", "mul", "div", "eq", "le", "lt", "gt", "ge", "and", "or", "not", "id") whose operands were loop invariant. This also implied that this process was run to a fixed-point on a per loop basis.

4. Attempts at returning out of SSA

I attempted to write code that could convert a program with phi instructions into one that had no phi instructions. I tried to follow the trivial method explained in lecture notes but quickly found that it was trickier than that. I did not include this in my analysis.

Results and Analysis:

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

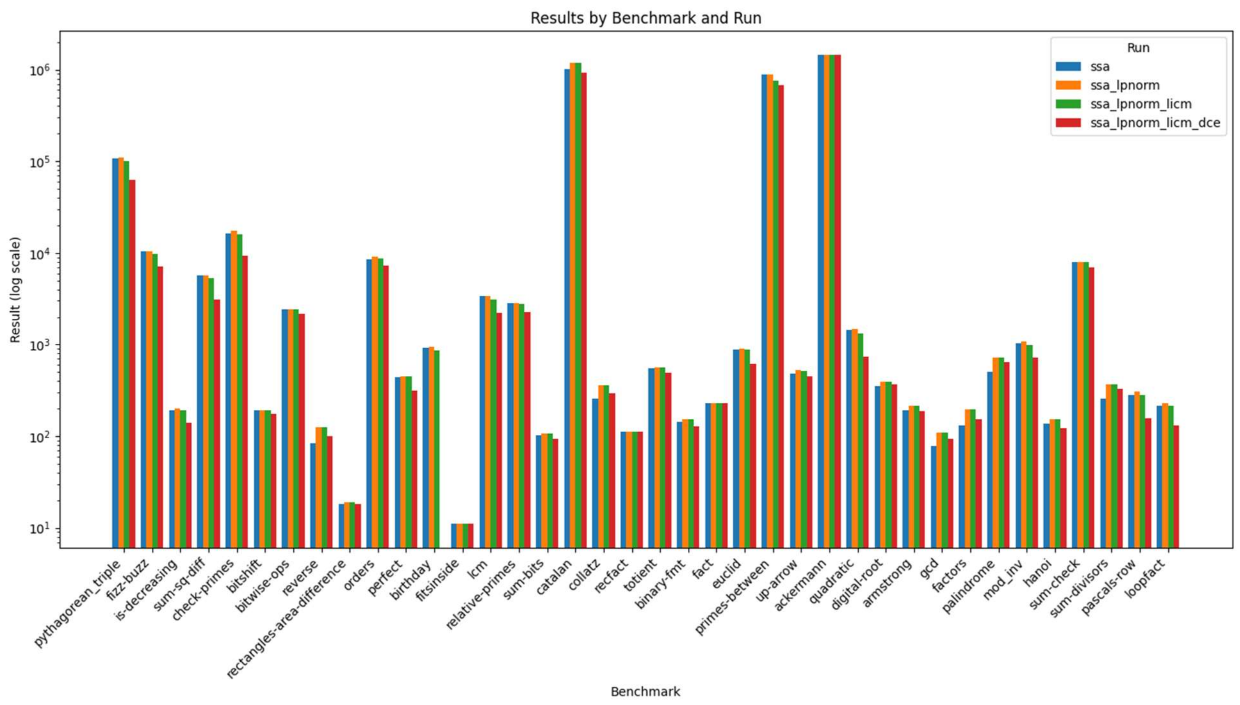
Benchmark Runs:

a. SSA (ssa)
b. SSA + Loop Normalization (ssa_lpnorm)
c. SSA + Loop Normalization + LICM (ssa_lpnorm_licm)
d. SSA + Loop Normalization + LICM + Global Constant Propagation/Folding + Global Liveness + Global Dead Code Elimination (ssa_lpnorm_licm_dce)

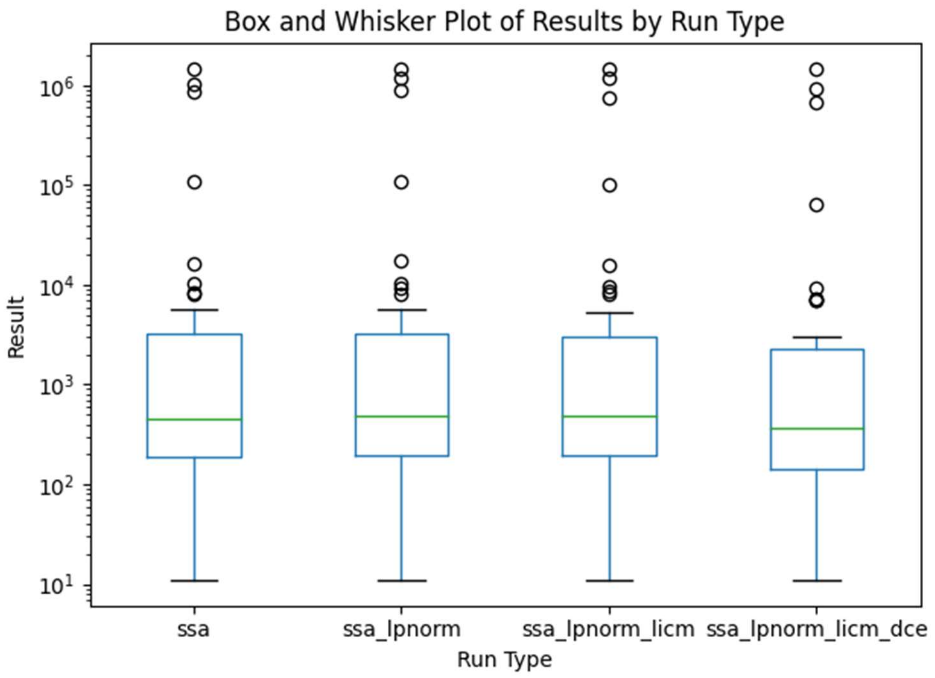
I added the last benchmark mostly for fun to see how much better it had gotten with both optimizations.

Benchmarks Analysis:

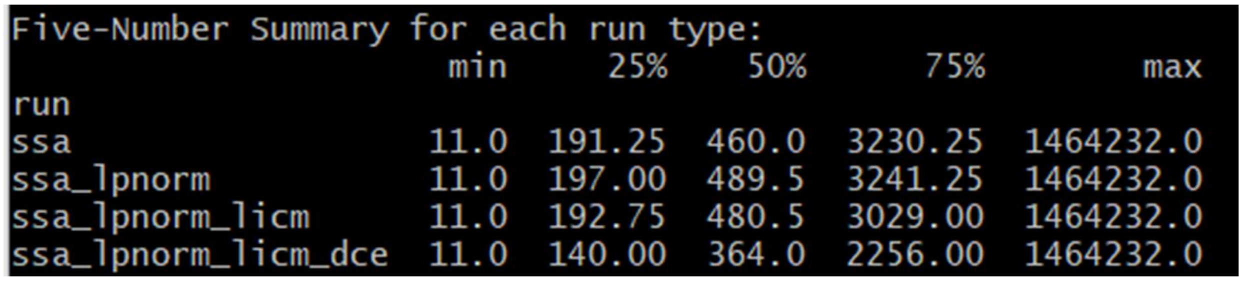
Below is a plot of the various benchmarks:



One note is that ‘birthday’ benchmark did not pass my last run because I did not handle floating point instructions in constant propagation. Like last time, I noticed a general pattern, that instruction count increased from the baseline SSA to SSA + Loop Normalization, then decreased when SSA + Loop Normalization + LICM was added. This is very expected since the loop normalization only strictly added blocks and instructions, while the LICM strictly decreased the number of instructions run. I was surprised when observing the average instructions across all the optimization passes: the baseline SSA had 93226.789474, SSA+Loop Normalization had 97673.184211, and SSA+Loop Normalization+LICM had 94084.631579. I expected the LICM to reduce more of the test cases to beneath the SSA benchmark, and it did in some cases. But for the majority of the cases this was not true. Outside of the Task 3 requirements, I also ran global constant propagation and liveness, which brought instructions down to 86075.162162 on average, which was nice to see. The large reductions are also in part due to previous optimizations not handling phis that are essentially useless. Below is the box-whisker plot showing the same benchmark runs:



As well as 5-number summary:



```
Five-Number Summary for each run type:
                      min     25%     50%      75%        max
run
ssa                  11.0  191.25   460.0  3230.25  1464232.0
ssa_lpnorm           11.0  197.00   489.5  3241.25  1464232.0
ssa_lpnorm_licm      11.0  192.75   480.5  3029.00  1464232.0
ssa_lpnorm_licm_dce  11.0  140.00   364.0  2256.00  1464232.0
```

From this data, we can see that for the Q1, median, and Q3 of the dataset, the LICM did in fact decrease the number of instructions from SSA, indicating that there was likely a skewness in the average and that LICM is meaningfully reducing the number of instructions run. The dead code reductions also were very nice to see.

Optimized Programs:

Here is a program that my optimizations do well on (and also do poorly on).:

```
@main {
.F:
  condit: bool = const true;
  loop_var: int = const 0;
  br condit .D .E;
.D:
  d_test: int = const 9;
  jmp .A;
.E:
  e_test: int = const 8;
  jmp .A;
.A:
  one: int = const 1;
  loop_var: int = add loop_var one;
  loop_var_copy: id = loop_var;
  print loop_var_copy;
  print one;
  cond1: bool = const true;
  br cond1 .B .C;

.B:
  two: int = const 2;
  print two;
  cond2: bool = const false;
  br cond2 .A .exit;

.C:
  three: int = const 3;
  print three;
#  jmp .A;
  br cond1 .A .inner;
.inner:
  hoist: int = const 17;
  jmp .C;

.exit:
  ret;
}
```

Optimized version:

```
@main {
.F:
  condit_0: bool = const true;
  loop_var_0: int = const 0;
  br condit_0 .D .E;
.D:
  d_test_0: int = const 9;
  jmp .A_preheader;
.E:
  e_test_0: int = const 8;
  jmp .A_preheader;
.A:
  hoist_0: int = phi hoist_0_L hoist_0_pre .A_latch .A_preheader;
  three_0: int = phi three_0_L three_0_pre .A_latch .A_preheader;
  cond2_0: bool = phi cond2_0_L cond2_0_pre .A_latch .A_preheader;
  two_0: int = phi two_0_L two_0_pre .A_latch .A_preheader;
  e_test_1: int = phi e_test_1_L e_test_1_pre .A_latch .A_preheader;
  d_test_1: int = phi d_test_1_L d_test_1_pre .A_latch .A_preheader;
  loop_var_1: int = phi loop_var_1_L loop_var_1_pre .A_latch .A_preheader;
  loop_var_2: int = add loop_var_1 one_1;
  loop_var_copy_1: id = loop_var_2;
  print loop_var_copy_1;
  print one_1;
  br cond1_1 .B .C_preheader;
.B:
  print two_1;
  br cond2_1 .A_latch .exit;
.C:
  hoist_1: int = phi hoist_2 hoist_1_pre .inner .C_preheader;
  print three_2;
  br cond1_1 .A_latch .inner;
.inner:
  jmp .C;
.exit:
  ret;
.A_latch:
  hoist_0_L: int = phi hoist_1 hoist_0 .C .B;
  three_0_L: int = phi three_2 three_0 .C .B;
  cond2_0_L: bool = phi cond2_0 cond2_1 .C .B;
  two_0_L: int = phi two_0 two_1 .C .B;
  cond1_0_L: bool = phi cond1_1 cond1_1 .C .B;
  loop_var_copy_0_L: id = phi loop_var_copy_1 loop_var_copy_1 .C .B;
  one_0_L: int = phi one_1 one_1 .C .B;
  e_test_1_L: int = phi e_test_1 e_test_1 .C .B;
  d_test_1_L: int = phi d_test_1 d_test_1 .C .B;
  loop_var_1_L: int = phi loop_var_2 loop_var_2 .C .B;
  jmp .A;
.A_preheader:
  hoist_0_pre: int = phi __undefined __undefined .E .D;
  three_0_pre: int = phi __undefined __undefined .E .D;
  cond2_0_pre: bool = phi __undefined __undefined .E .D;
  two_0_pre: int = phi __undefined __undefined .E .D;
  cond1_0_pre: bool = phi __undefined __undefined .E .D;
  loop_var_copy_0_pre: id = phi __undefined __undefined .E .D;
  one_0_pre: int = phi __undefined __undefined .E .D;
  e_test_1_pre: int = phi e_test_0 __undefined .E .D;
  d_test_1_pre: int = phi __undefined d_test_0 .E .D;
  loop_var_1_pre: int = phi loop_var_0 loop_var_0 .E .D;
  two_1: int = const 2;
  cond2_1: bool = const false;
  one_1: int = const 1;
  cond1_1: bool = const true;
  three_2: int = const 3;
  hoist_2: int = const 17;
  jmp .A;
.C_preheader:
  hoist_1_pre: int = phi hoist_0 .A;
  three_1_pre: int = phi three_0 .A;
  jmp .C;
}
```

In these programs, some of the things my program does well are handling the phi placements and making sure that when latches and preheaders are created, the phis are updated accordingly. Furthermore, my program is able to hoist code from the inner loop to the outer loop, and if still hoist-able like the variable “hoist”, it is taken outside of all the loops. Some easy instructions to remove are some of the phi instructions that only have “__undefined” values as arguments, since those corresponding destinations should never be used in that block (this happens when after SSA, I do patchup to fix adding phis in preheaders and latches). However, dead code elimination can be easily modified to capture that. Another aspect my code misses out on is induction variable analysis. For example, the variable “loop_var_copy” is clearly an induction variable, but no optimizations are made to eliminate the copy and only use a minimal set of induction variables.