Shankar Kailas – 3036861618

CS 265

Group: Shankar Kailas

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

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

Summary of Accomplishments:

In this task, I accomplished the following tasks:

1. Global constant propagation/folding analysis
2. Global liveness analysis
3. Global dead code elimination

Algorithms and Design Decisions:

1. Global constant propagation/folding analysis

For this analysis, the first step was to create the blocks and the CFG. For these steps, I largely used the provided code snippets with some minor edits. Functions here included adding an entry block for each label (not starting with a loop), creating a map between labels (artificial if needed) and blocks of code, adding terminators like jump if necessary to make control flow more obvious, and calculating the edges in the CFG. While these operations were not done by me, one design choice I had here was to do these operations in this particular order – if they weren't (like adding the terminators before the entry node), then floating blocks would appear, and they wouldn't properly be connected in the CFG. To determine the optimized order, I opted for a reverse post order. I found the block with no predecessors, computed the post order (approximated for cyclic graphs) in a depth first search manner, and then reversed the list. This was the order that blocks were initially added to the worklist. The worklist was implemented as a queue. The in and out of each block was represented as a dictionary with the keys as variable names and values as the calculated value of the corresponding variable. As taught, the meet over the predecessors was done as an intersection. The transfer function for this optimization replaced instructions (e.g. id, eq, add, etc.) for which all arguments' values were known with a const instruction to do the constant folding. The math was done by a self-built evaluation function. After each instruction, if the value of the destination variable was known, it was added to the corresponding block's out map. If after iterating though a block I found that the block's out map had changed (using a bool flag to reduce comparisons), I would add all of this block's successors to the worklist as taught. I decided to add it to the end of the worklist, but this was done arbitrarily. To reduce worklist additions, I also kept track of the contents in the worklist as a set so that I see if blocks were already in the worklist faster. Overall, no lines of code were removed, and in fact lines often increased after this. However, this analysis could do the same constant propagation and folding that the LVN pass could do and more, since it was able to pass guaranteed constant information from predecessors to successors.

2. Global liveness analysis

To conduct global liveness analysis, I did the same block and CFG processing as in global constant propagation/folding. The first major difference was that instead of reverse post order, I simply did the post order since I read that was generally best for reverse passes. As with earlier, a set (for faster lookups) and queue were used to track the worklist. The in and out sets of variables that indicated if variables were live at the beginning and end of the block respectively were represented as python sets. This was an optimistic optimization, so it was critical to make sure that everything worked as intended. As this was a reverse pass, meets were done over successors' in sets to determine a blocks out set. To implement the transfer function, I iterated over instructions of a block in the reverse order, first removing destination variables from the liveness set and then adding argument variables to it. This was done to keep variables in self-referential assignments in the liveness set. Again, a flag was used to track if changes were made to the in set, and if so, predecessors were added to the worklist. Predecessors were added to the front of the list this time. This was done with the mentality that blocks further from the initial block will be processed first. Through this process, I was able to successfully calculate all of the live variables at the beginning and end of each block. Again, no lines of code were optimized here, and no lines of code were really modified.

3. Global dead code elimination

I implemented global dead code elimination using the out sets for each block from the liveness analysis. Essentially, what I did was run the transfer function again as described in global liveness analysis, updating it line-by-line to have the most up-to-date liveness. However, this time, if an instruction had a destination variable that was not liveness set, then it was removed. This global DCE took advantage of both the constant folding/propagation analysis and the liveness analysis, as many times constants that were propagated or folded rendered other constants no longer live. Then by this global DCE that removed non-live assignments, many more instructions could be removed – much more than what LVN could do.

4. Note on SSA

One aspect that has been made painfully apparent to me is that I should have implemented SSA in this project. I did not think it would make things easier since SSA does not have any optimizations, but handling variable reassignments would have been much easier. At first it was a struggle until I properly modified my transfer function in liveness to conduct the global dead code analysis.

Results and Analysis:

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

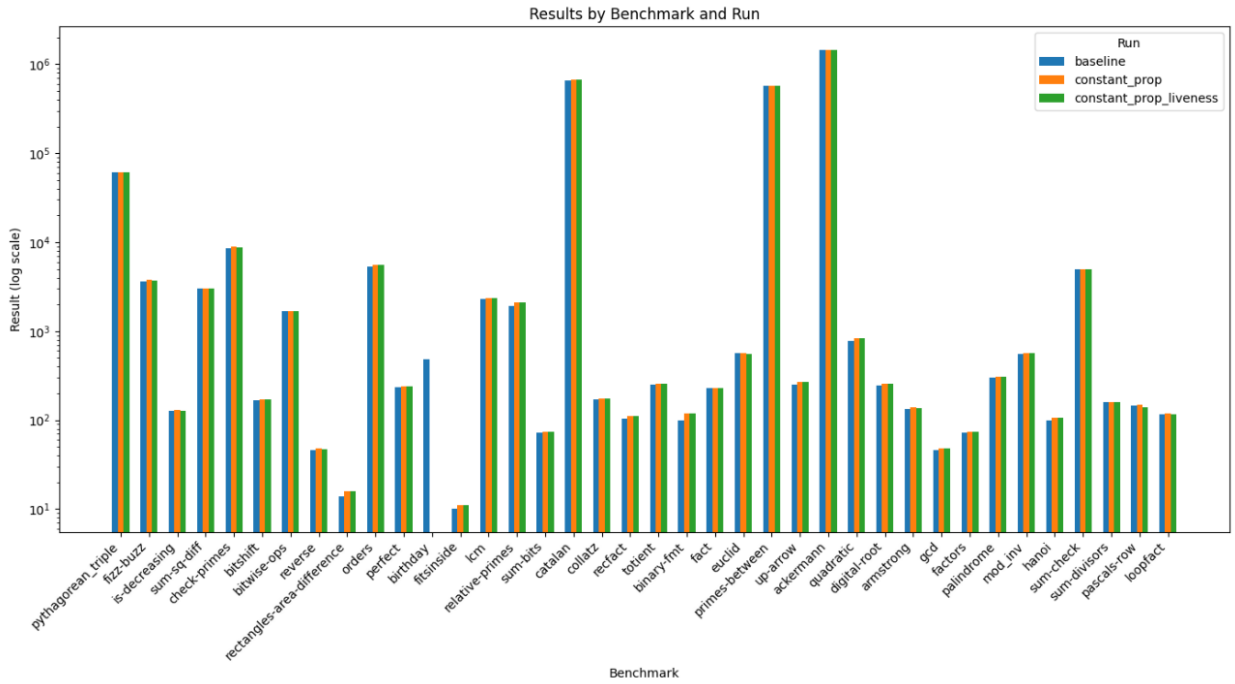
Core benchmarks

a. Baseline
b. Global Constant Propagation/folding (constant_prop)

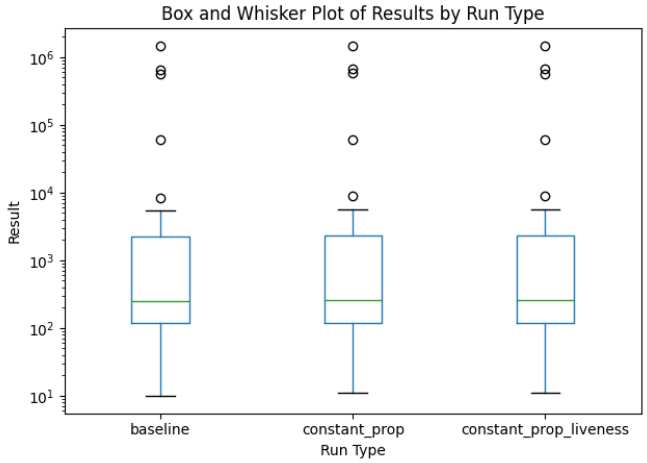
c. Global Constant Propagation/folding + Global Liveness + Global Dead Code Elimination (constant_prop_liveness)

Core benchmarks Analysis

Below is a plot of the various benchmarks:



Despite being plotted on a log scale, it is still difficult to tell the difference between the different optimizations. One note is that 'birthday' benchmark did not pass, but this is because I did not handle floating point instructions. Upon close inspection, I noticed a general pattern, that instruction count increased from the baseline to just constant propagation\folding, then decreased when Global Liveness + Global Dead Code Elimination was added. This is very expected. Constant propagation does not remove any instructions, and the formation of more ordered and proper blocks for easier computation only increases the number of instructions and labels. When adding Global Liveness + Global Dead Code Elimination, instructions that both liveness and constant propagation/folding (which affected liveness) rendered not live were removed, resulting in a decrease in instructions. I was very surprised when observing the average instructions across all the optimization passes: the baseline had 73583.578947, just global constant propagation/folding had 76184.189189, and Global Constant Propagation/folding + Global Liveness + Global Dead Code Elimination had 76078.270270. I was surprised by how much more Global Constant Propagation/folding + Global Liveness + Global Dead Code Elimination had than the baseline, however, this may be due to the number of instructions added to make blocks proper and ordered. Furthermore, I figured that larger programs may be skewing the average, so I created a box and whisker plot:



As well as 5-number summary:

```
Five-Number Summary for each run type:
                          min     25%    50%      75%        max
run
baseline                 10.0  118.75  249.5  2225.25  1464231.0
constant_prop            11.0  118.00  258.0  2328.00  1464232.0
constant_prop_liveness   11.0  118.00  258.0  2328.00  1464232.0
```

From this data, we can see that the discrepancy in instruction count is much less for at least 75% of the data. However, almost each value indicates that my optimizations added more instructions than deleted them. That being said, many of the added instructions likely do not reduce the speed of the compiler that much, and more constants will be in the program, rather than unnecessary calculations, which may still make compile and runtime faster. This should be verified using wall clock time, however.

Optimized Programs:

Here is a program that my optimizations do well on. This script has clear opportunities for constant folding (x1 =x+x), rendering the first assignment to be global dead code. There is also dead code that is discovered in liveness analysis (z=y). Furthermore, local dead code analysis is still active (one=2):

```
@main {
.entry:
  x: int = const 0;
  z: int = id y;
  x1: int = add x x;
  jmp .header;
```

```
.header:
  c: bool = lt x1 a;
  br c .loop .exit;

.loop:
  one: int = const 2;
  one: int = const 1;
  x1: int = add x1 one;
  jmp .header;

.exit:
  print x1;
}
```

Optimized version:

```
@main {
.entry:
  x1: int = const 0;
  jmp .header;
.header:
  c: bool = lt x1 a;
  br c .loop .exit;
.loop:
  one: int = const 1;
  x1: int = add x1 one;
  y: int = add y one;
  jmp .header;
.exit:
  print x1;
  ret;
}
```

Here is a script that is not optimized well. The optimization suffers due to lack of handling common subexpressions, lack of constant propagation for branch instructions (and subsequent conversion from branch to jump). Furthermore, realizing that a block has no predecessors and is not the start block is another way to remove blocks entirely (like the .body here) that my optimizations do not handle.

```
@main {
  result: int = const 1;
  i: int = const 0;
  a: int = add b c;
  d: int = add b c;
```

```
  print a
  print d

.header:
  zero: int = const 0;
  cond: bool = eq i zero;
  br cond .end .body;

.body:
  result: int = mul result i;

  # i--
  one: int = const 1;
  i: int = sub i one;

  jmp .header;

.end:
  print result;
}
```

Optimized version:

```
@main {
.b1:
  result: int = const 1;
  i: int = const 0;
  a: int = add b c;
  d: int = add b c;
  print a
  print d
  jmp .header;
.header:
  zero: int = const 0;
  cond: bool = eq i zero;
  br cond .end .body;
.body:
  result: int = mul result i;
  one: int = const 1;
  i: int = sub i one;
  jmp .header;
.end:
  print result;
  ret;
}
```