Shankar Kailas – 3036861618

CS 265

Group: Shankar Kailas

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

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

Summary of Accomplishments:

In this task, I accomplished the following tasks:

1. Global points-to aliasing analysis
2. Global address liveness analysis
3. Global dead store elimination

Algorithms and Design Decisions:

1. Global points-to aliasing analysis

For this analysis, the first step was to create the blocks and the CFG for each function, since this was an intraprocedural analysis. For these steps, I largely used the provided code snippets with some minor edits. Functions here included ensuring a function entry block, creating a map between labels (artificial if needed) and blocks, adding terminators, and assembling the CFG using the control flow statements. From here, I removed dead code that followed a terminator (these would appear in blocks that had no predecessors and were 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). As taught in class, this was a forward, "may" analysis. To determine the optimized order, I opted for a reverse post order. I found the entry and 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 pointer variable names and values as the potential static address locations that the variable could point to. The addresses were generated from the line numbers of the bril program. If the address could be any (unknown) address, then a value of "__all__" was used to signify this (e.g. function parameter pointers were given this value since we do not know what). As taught, the meet over the predecessors was done as a union to indicate potential aliasing. The transfer function for this optimization involved handling "alloc", "id", "ptradd", "call", or "load". If it was an "alloc" then the destination pointer was given a set with the line number. If it was an "id" between pointers, then all locations from the argument pointer were copied to the destination pointer's set. This was also done for "ptradd" since the offset could be 0. If the pointer was a destination to a "call" or a "load", then it was given the value of "__all__", since we don't know what the argument pointer could point to. 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.

2. Global address liveness analysis

To conduct global address 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 were static addresses as computed before that indicated which were live at the beginning and end of the block respectively were represented as python sets. 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. If an instruction used a pointer in a load, as a parameter for a function, or as a return variable, then I added all addresses associated with that variable (from previous analysis) to the 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 an over-approximation of live static addresses at the beginning and end of each block.

3. Global dead store 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 address liveness analysis, updating a live address set line-by-line to have the most up-to-date liveness. However, this time, if an instruction was a "store" and none of the static addresses (found from the points-to out map of the block) associated with the pointer were considered live, then the instruction was removed. An important note is that pointers with the "__all__" value in their sets were not removed to be extra cautious.

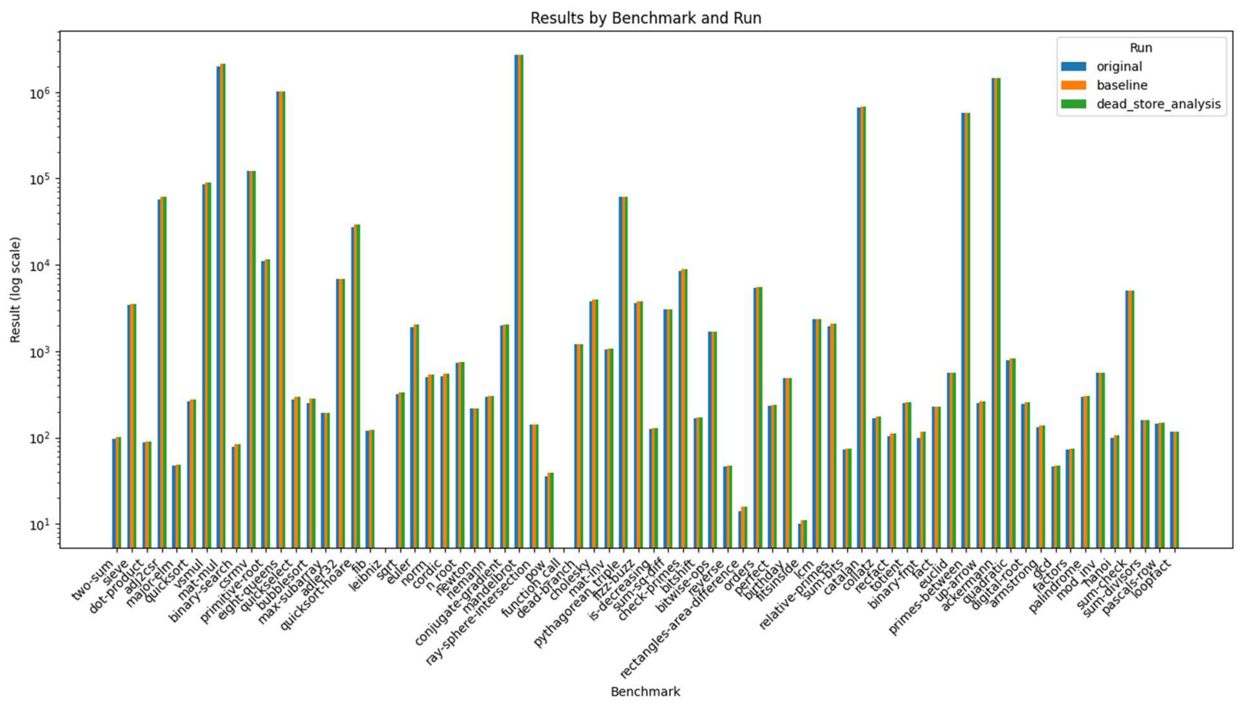
Results and Analysis:

To measure the value of these optimizations and their ability to work together, I used the provided benchmark suite:

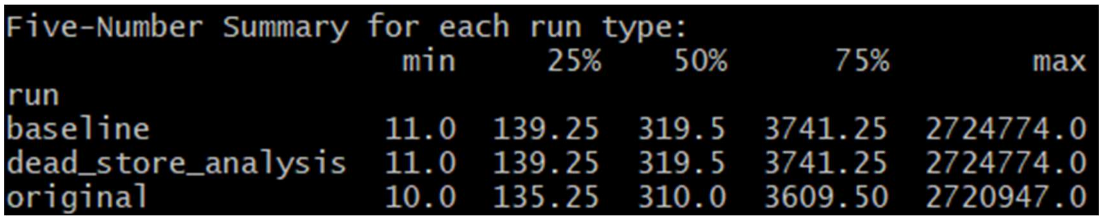
Core benchmarks

a. No Changes (original)
b. Simple Block Formatting (baseline)
c. Simple Block Formatting + Dead Store Elimination (dead_store_analysis)

Benchmarks Analysis:



It quickly became clear that my optimization was not optimizing anything. This was true for even the programs that had memory instructions (found in the mem, mixed, and float subfolders). On a manual inspection, I did not find any dead stores in the code that my optimization would have been able to capture, so this was an expected result. Both with and without my optimization gave the same statistics for instructions. However, it was nice to see that my optimization did not break anything either, since all of the programs ran correctly. Further analysis on this benchmark was not conducted since there were no programs that had scenarios that I optimized for. Instead, I opted to observe some of my own test cases.


```
Five-Number Summary for each run type:
                      min     25%    50%      75%        max
run
baseline             11.0  139.25  319.5  3741.25  2724774.0
dead_store_analysis  11.0  139.25  319.5  3741.25  2724774.0
original             10.0  135.25  310.0  3609.50  2720947.0
```

Optimized Programs:

This program highlights some of the benefits that my optimization can provide. This program has a dead store to mem2 which gets eliminated. It also has one live store and one dead store to mem1, and only the dead store is eliminated. Furthermore, the optimization does not touch the second store to mem2 since it is loaded elsewhere in the program in a different block that may run. Finally, this program also does not touch the store to ptr2. This is good because ptr2 is an alias to ptr1, and despite there not being a reason for liveness in this function call, it may be loaded from later in the program in another function:

```
@main(ptr1: ptr<int>) {
    size: int = const 1;
    mem1: ptr<int> = alloc size;
    mem2: ptr<int> = alloc size;
    store mem1 size;
    test: int = load mem1;
    rand: int = const 2;
    store mem2 rand;
    store mem1 size;
    cond: bool = const true;
    store mem2 size;
    br cond .A .B;
.A:
    test2: int = load mem2;
    jmp .end;
.B:
    ptr2: ptr<int> = id ptr1;
    jmp .end;

.end:
    free mem1;
    free mem2;
    store ptr2 rand;
    ret;
}
```

Optimized version:

```
@main(ptr1: ptr<int>) {
.b1:
  size: int = const 1;
  mem1: ptr<int> = alloc size;
  mem2: ptr<int> = alloc size;
  store mem1 size;
  test: int = load mem1;
  rand: int = const 2;
  cond: bool = const true;
  store mem2 size;
  br cond .A .B;
.A:
  test2: int = load mem2;
  jmp .end;
.B:
  ptr2: ptr<int> = id ptr1;
  jmp .end;
.end:
  free mem1;
  free mem2;
  store ptr2 rand;
  ret;
}
```

Here is a script that is not optimized well. The optimization suffers due to lack of handling interprocedural optimizations. It is clear that the stores to ptr1 and ptr2 here are dead, since they are never loaded from, but because the stores happen in a different function, my analysis assumes that they could be accessed since it assumes that ptr1 and ptr2 could point anywhere:

```
@process_memory(ptr1: ptr<float>, ptr2: ptr<float>) {
    a: float = const 10.0;
    b: float = const 20.0;
    c: float = const 30.0;
    store ptr1 a;
    store ptr2 b;
    ret;
}

@main() {
    size: int = const 1;
    mem1: ptr<float> = alloc size;
    mem2: ptr<float> = alloc size;
    call @process_memory mem1 mem2;
    result: float = load mem1;
    print result;
    free mem1;
    free mem2;
    ret;
}
```

Optimized version (unable to optimize):

```
@process_memory(ptr1: ptr<float>, ptr2: ptr<float>) {
    a: float = const 10.0;
    b: float = const 20.0;
    c: float = const 30.0;
    store ptr1 a;
    store ptr2 b;
    ret;
}
```

```
@main() {
    size: int = const 1;
    mem1: ptr<float> = alloc size;
    mem2: ptr<float> = alloc size;
    call @process_memory mem1 mem2;
    result: float = load mem1;
    print result;
    free mem1;
    free mem2;
    ret;
}
```