

Problem 4

I chose an algorithm to find the rank of a matrix ([https://en.wikipedia.org/wiki/Rank_\(linear_algebra\)](https://en.wikipedia.org/wiki/Rank_(linear_algebra))). The algorithm is a relatively basic one based on the commonplace Gauss-Jordan elimination method. More about it can be read here: https://cp-algorithms.com/linear_algebra/rank-matrix.html (this link also contains a C++ implementation, and I effectively rewrote their implementation in C).

Part 1:

Data for the optimization levels + input matrix size vs. execution time was taken, and results can be seen in the chart below. Unsurprisingly, the program runtime scaled with the matrix size, and it wasn't until the matrix hit 5000 x 5000 that time differences became significant. The program was compiled and executed on an M1 Pro Max Macbook Pro with 32 GB of RAM.

It's worth noting that even though there was a large difference between the unoptimized program and the -O1 optimized program, there wasn't a large difference for subsequent optimizations. The most notable difference would be how the -Oz was noticeably slower than the O1, O2, Os, O3, and Ofast optimizations. This makes me think that the vast majority of the major optimizations that could be made were included in -O1 optimization.

In terms of program size, there does not seem to be a big difference regardless of the optimization. The largest program was actually the -Ofast one, and the smallest one was tied between the O1, O2, and Os programs. One thing that I find interesting is that the -Oz compiled program was actually larger than the Os program. This is surprising because, from my understanding, the -Oz optimization attempts to optimize size more aggressively than the -Os optimization. I think this might be because the program is so small in relation to a real program that would actually benefit from size optimization.

Time	No optimizations (O0)	O1	O2	Os	Oz	O3	Ofast
10 x 10	0.387 sec	0.089 sec	0.238 sec	0.100 sec	0.450 sec	0.347 sec	0.269 sec
50 x 50	0.130 sec	0.103 sec	0.231 sec	0.107 sec	0.247 sec	0.095 sec	0.240 sec
500 x 500	0.386 sec	0.266 sec	0.129 sec	0.229 sec	0.142 sec	0.244 sec	0.130 sec
5000 x 5000	2 min 17.245 sec	46.277 sec	46.037 sec	46.265 sec	52.741 sec	46.411 sec	46.036 sec
10000 x 10000	17 min 39.286 sec	5 min 32.531 sec	5 min 31.481 sec	5 min 32.281 sec	6 min 35.099 sec	5 min 33.932 sec	5 min 33.565 sec
Optimization	Program size						
No optimizations (O0)	33604 bytes						
O1	33569 bytes						
O2	33569 bytes						
Os	33569 bytes						
Oz	33601 bytes						
O3	33633 bytes						
Ofast	33636 bytes						

Part 2:

1. `--type-promotion + --break-crit-edges` made an effect. Examining the difference between the unoptimized and optimized code with `vimdiff`, it seems like the majority of the differences seem to be these `"_crit_edgeX"` labels, as seen below in the screenshot. Interestingly, these `crit_edge` labels don't appear with just the `--break-crit-edges` optimization flag used. Both the `--type-promotion` and `--break-crit-edge` flags need to be set.

```

41 br label %23
42
43 23:
44 %24 = load i32, ptr %10, align 4
45 %25 = load i32, ptr %4, align 4
46 %26 = icmp slt i32 %24, %25
47 br i1 %26, label %27, label %28
48
49 27:
50 %28 = load ptr, ptr %8, align 8
51 %29 = load i32, ptr %10, align 4
52 %30 = sext i32 %29 to i64
53 %31 = getelementptr inbounds i32, ptr %28, i64 %30
54 %32 = load i32, ptr %31, align 4
55 %33 = icmp ne i32 %32, 0
56 br i1 %33, label %34, label %35
57
58 34:
59 %35 = load ptr, ptr %6, align 8
60 %36 = load i32, ptr %10, align 4
61 %37 = sext i32 %36 to i64
62 %38 = getelementptr inbounds ptr, ptr %35, i64 %37
63 %39 = load ptr, ptr %38, align 8
64 %40 = load i32, ptr %9, align 4
65 %41 = sext i32 %40 to i64
66 %42 = getelementptr inbounds i64, ptr %39, i64 %41
67 %43 = load i64, ptr %42, align 8
68 %44 = call @llabs(i64 noundef %43) #5
69 %45 = sitofp i64 %44 to double
70 %46 = fcmp ogt double %45, 1.000000e-09
71 br i1 %46, label %47, label %48
72
73 47:
74 br label %52
75
76 48:
77 br label %49
78
79 49:
80 %50 = load i32, ptr %10, align 4
81 %51 = add nsw i32 %50, 1
82 store i32 %51, ptr %10, align 4
83 br label %23, !llvm.loop !5
84
85 52:
86 %53 = load i32, ptr %10, align 4
87 %54 = load i32, ptr %4, align 4
88 %55 = icmp ne i32 %53, %54
89 br i1 %55, label %56, label %57
90
91 56:
92 br label %23, !llvm.loop !5
93
94 57:
95 %53 = load i32, ptr %10, align 4
96 %54 = load i32, ptr %4, align 4
97 %55 = icmp ne i32 %53, %54
98 br i1 %55, label %56, label %57
99
100 _crit_edge3:
101 br label %160
102

```

2. `--global-opt` also made an effect. This time it seems like it affected function definitions. From the `vimdiff` output, it seems like the tag `"local_unnamed_addr"` has been added

before the "#X" portion.

```

1 #include <optimizations.h>
2 source_filename = "optimizations.c"
3 target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
4 target triple = "armv8-apple-macosx13.0.0"
5
6 @str = private unnamed_addr constant [3 x i8] @"/00", align 1
7
8 ; Function Attrs: noinline nounwind optnone ssp uwtable
9 @llvm.local_unnamed_addr@ = local_unnamed_addr @#
10
11 %4 = alloca i32, align 4
12 %5 = alloca i32, align 4
13 %6 = alloca ptr, align 8
14 %7 = alloca i32, align 4
15 %8 = alloca ptr, align 8
16 %9 = alloca i32, align 4
17
18 ; 229 lines: %10 = alloca i32, align 4 ; preds = %10
19
20 %105 = load i32, ptr %7, align 4
21 ret i32 %105
22
23 ; Function Attrs: allocsize(0)
24 @llvm.ptr_malloc164.nonleaf@ =
25 ; Function Attrs: nounwind readnone willreturn
26 @llvm.ptr_malloc164.nonleaf@ =
27 ; Function Attrs: noinline nounwind optnone ssp uwtable
28
29 %1 = alloca i32, align 4
30 %2 = alloca i32, align 4
31 %3 = alloca i32, align 4
32 %4 = alloca ptr, align 8
33 %5 = alloca i32, align 4
34 %6 = alloca i32, align 4
35
36 ; 87 lines: store i32 %3, ptr %1, align 4
37
38 %58 = call i32 @rank1(i32 noundef %55, i32 noundef %56, ptr noundef %57)
39 %59 = call i32 (ptr, ...) @print1(ptr noundef @str, i32 noundef %58)
40 %60 = load i32, ptr %1, align 4
41 ret i32 %60
42
43 ; Function Attrs: noinline nounwind optnone ssp uwtable
44 @llvm.local4.random@ =
45 ; Function Attrs: noinline nounwind optnone ssp uwtable
46 @llvm.local4.random@ =
47
48 ; 19 lines: attributes #5 = { nounwind readnone willreturn }
49
50 attributes #0 = { noinline nounwind optnone ssp uwtable "frame-pointer"="non-leaf" "min-legal-vector-width"="0" "no-
51 attributes #1 = { allocsize(0) "frame-pointer"="non-leaf" "no-trapping-math"="true" "stack-protector-buffer-size"="8
52 attributes #2 = { nounwind readnone willreturn "frame-pointer"="non-leaf" "no-trapping-math"="true" "stack-protector
53 attributes #3 = { "frame-pointer"="non-leaf" "no-trapping-math"="true" "stack-protector-buffer-size"="8" target-cpu
54 attributes #4 = { allocsize(0) }
55
56 ; 19 lines: attributes #5 = { nounwind readnone willreturn }
57

```

3. `--strip` had a minor effect. It seems like a lot of the `"@.str"`s were simply replaced with `"@0"`.

```

@EPS = constant double 1.000000e-09, align 8
@str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @rank(i32 noundef %0, i32 noundef %1, ptr no
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca ptr, align 8
  ;--333 lines: %7 = alloca i32, align 4
  54:                                     ; preds = %28
  %55 = load i32, ptr %2, align 4
  %56 = load i32, ptr %3, align 4
  %57 = load ptr, ptr %4, align 8
  %58 = call i32 @rank(i32 noundef %55, i32 noundef %56, ptr noundef %57)
  %59 = call i32 (ptr, ...) @printf(ptr noundef @str, i32 noundef %58)
  %60 = load i32, ptr %1, align 4
  ret i32 %60
}
declare i64 @random() #3
;-- 26 lines: declare i32 @printf(ptr noundef, ...) #3

```