6.035 Team **Antidaephobiax**: Optimizations Writeup

Abel Tadesse, Devin Morgan, Madhav Datt

March 17, 2017

1 Overview

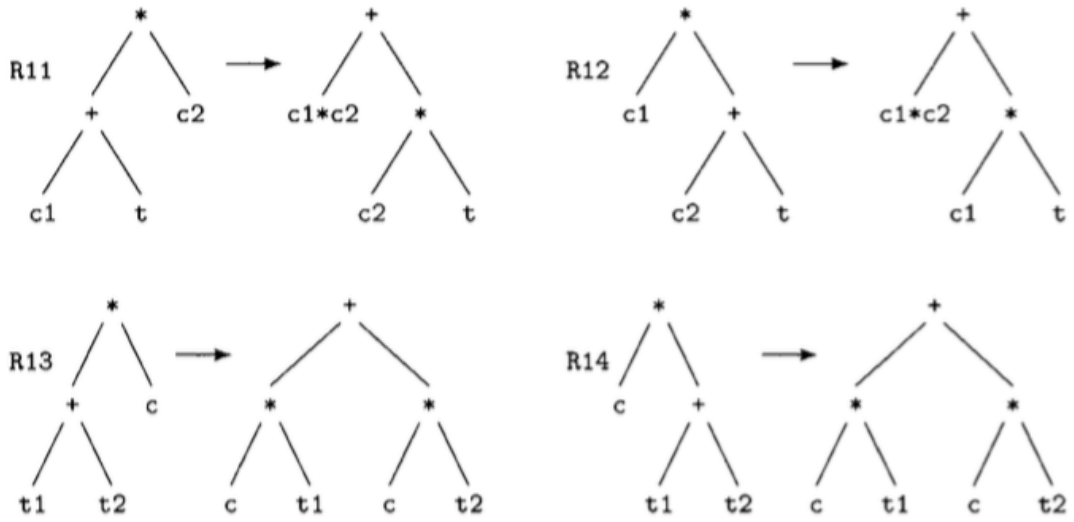
In this written documentation we discuss in detail, each optimization that we had considered while building our compiler and in the optimization phase of the project. We go through the rationale and decision making process for each optimization that we finally implemented. Supplemented by examples from our generated code, in the form of High Level Intermediate Representations, Low Level Intermediate Representations, and generated assembly programs, we further talk about how each optimization has helped in making our compiler more efficient, both in general, and specifically with respect to the image processing benchmarks. We also briefly discuss the correctness of each optimization, through examples and sample programs, in the process. Towards the end of this document, we discuss our full-optimizations command line option and present a detailed analysis of how and why we went our specific order of optimizations, followed by which optimizations were applied multiple times at different

---

2 Optimizations Implemented

We use various small Decaf programs as examples to explain the various optimizations implemented. The low level IR of such programs is represented by a mapping between a unique line label as key and the actual statement as value. Variables of the form `#_tX` are temporary variables generated for use in the program by the compiler.

2.1 Expression Canonicalization



Expressions in a computer program are often written so that they can be easily read and understood by a human or programmer. Examples of this may include the use of parenthetical expressions to indicate the demensions of a multi-dimensional array or multiplying a mixture of variables and constants. Moreover, these expressions can be arbitrarily complex. However, Expression Canonicalization is a technique that attempts to standardize the format of all expressions by treating expressions as tress and recursively applying algebriac properties. Expression Canonicalization often intrudes additional common subexpressions that were inacessible as well opportunities to perform compile-time computations of constants.

2.2 Parallelization

We implement an extremely restrictive version of loop parallelization for our compiler, that parallelizes very specific functions. We perform this optimization in the high level intermediate representation (IR), and the compiler detects loops that can be safely and efficiently executed in parallel and generates multithreaded code. We specifically target void return-type functions with no parameters and perfectly nested loops, and use the Analyze class of the provided library to compute their distance vectors. The following function is an example of code that can be automatically parallelized by our compiler.

```
1 int A[1000000];
2 void f() {
3   int i;
4   for (i = 0; i < 1000000; i += 1)
5     A[i] = A[i] * 2;
6 }
```

Once the maximum number of threads available is set to 8, the following assembly code is a snippet of the assembly generated by our compiler for a parallelized version of the function $f()$. All calls to $f()$ in the *main* function of the program are replaced by calls to `create_and_run_threads(f)` from the 6035 library. Since, for the perfectly nested loops in the shown function, our distance vector is [0], we generate the following assembly code for the parallelized version of the function.

```
1  f_entry:
2        movq    $0, %r12
3        movq    $0, %r12
4  f_FOR_COND_L4:
5        movq    $1000000, %r13
6        movq    $1000000, %r10
7        movq    %r12, %r11
8        cmp     %r10, %r11
9        movq    $0, %r11
10       movq    $1, %r10
11       cmovl   %r10, %r11
12       movq    %r11, %r13
13       movq    %r13, %r10
14       movq    $0, %r11
15       cmp     %r10, %r11
16       jne     f_FOR_L9
17       jmp     f_END_FOR_L21
18 f_FOR_L9:
19       movq    -48(%rbp), %r10
20       movq    %r12, %r11
21       addq    %r10, %r11
22       movq    %r11, %r13
23       movq    $2, %r14
24       movq    %r13, %rax
25       movq    A(,%rax, 8), %rax
26       movq    %rax, -56(%rbp)
27       movq    $2, %r10
28       movq    -56(%rbp), %r11
29       imulq   %r10, %r11
30       movq    %r11, %r14
31       movq    %r13, %r15
32       movq    %r13, %r10
33       movq    %r14, %r11
34       movq    %r11, A(, %r10, 8)
35       movq    $1, %r15
36       movq    $8, %rbx
37       movq    $8, %r10
38       movq    $1, %r11
39       imulq   %r10, %r11
40       movq    %r11, %rbx
41       movq    %r12, %r11
42       addq    %rbx, %r11
43       movq    %r11, %r15
44       movq    %r15, %r12
45       jmp     f_FOR_COND_L4
46 f_END_FOR_L21:
```

```
47 f_exit:
```

Since, we are only parallelizing the innermost loop after making sure that it is a FOR EACH type of loop, each iteration, and consequently, the computations on each thread, is independent of all the others. Given the wide prevalence of such loops in general, and in the image processing benchmark programs, implementing this optimization allowed us to leverage multithreading to gain as much as a 4× speed-up for certain functions. One of the largest challenges for this design was to ensure that unsafe/non-parallelizable loops don't get parallelized, as this could lead to incorrect execution and results, or extremely slow timings, given the cost of the system calls that are made by the 6036.*c* library. We addressed this problem by developing an extremely restrictive criteria for functions and the loops they contain, to be parallelized. Though this caused a lot of completely parallelizable loops to get skipped over, we were able to avoid incorrect evaluations.

2.3 Common Subexpression Elimination

Here, we started by performing the available expressions data flow analysis, and determined for each point in the program the set of expressions that need not be re-computed. The following code will be our running example for common subexpression elimination, copy propagation, and dead code elimination.

```
1  void main ( ) {
2    int a, b, c, d;
3    a = get_int ( 2 );
4    b = get_int ( 3 );
5    c = 0;
6    d = 0;
7    c = ( a + b ) * ( a + b );
8    d = ( a + b ) / ( a + b );
9    printf ( "%d\n", c );
10   printf ( "%d\n", d );
11 }
```

For the above mentioned program, we get the following unoptimized low level IR, and corresponding set of basic blocks for its control flow graph.

```
1  main :  EMPTY_STATEMENT
2  L0 :  #_t0 = 2
3  L1 :  #_t1 = get_int(#_t0,)
4  L2 :  a = #_t1
5  L3 :  #_t2 = 3
6  L4 :  #_t3 = get_int(#_t2,)
7  L5 :  b = #_t3
8  L6 :  #_t4 = 0
9  L7 :  c = #_t4
10 L8 :  #_t5 = 0
11 L9 :  d = #_t5
```

```
L10 :  #_t6 = a + b
L11 :  #_t7 = a + b
L12 :  #_t8 = #_t6 * #_t7
L13 :  c = #_t8
L14 :  #_t9 = a + b
L15 :  #_t10 = a + b
L16 :  #_t11 = #_t9 / #_t10
L17 :  d = #_t11
L18 :  #_t13 = printf(#str_t12 ,c,)
L19 :  #_t15 = printf(#str_t14 ,d,)
```

After performing CSE on this algorithm, we had the following resulting low level IR. It is important to note that this has an increased size in terms of temporaries and subexpressions than the original program.

```
main :  EMPTY_STATEMENT
L0 :  #_t0 = 2
L1 :  #_t1 = get_int(#_t0 ,)
L2 :  a = #_t1
L3 :  #_t2 = 3
L4 :  #_t3 = get_int(#_t2 ,)
L5 :  b = #_t3
L6 :  #_t4 = 0
L7 :  c = #_t4
L8 :  #_t5 = 0
L9 :  d = #_t5
L10 :  #_t6 = a + b
L11 :  #_t7 = #_t6
L12 :  #_t8 = #_t6 * #_t7
L13 :  c = #_t8
L14 :  #_t9 = #_t6
L15 :  #_t10 = #_t6
L16 :  #_t11 = #_t9 / #_t10
L17 :  d = #_t11
L18 :  #_t13 = printf(#str_t12 ,c,)
L19 :  #_t15 = printf(#str_t14 ,d,)
```

While this process had an overall negative impact on our compiler performance against the benchmarks because of the increased size of code caused by insertion of lines with temporary variables getting allocated to available common expressions, we implemented this optimization none-the-less as it paved our way towards other optimizations, making it possible to perform copy-propagation on the program, which is discussed further in the document.

2.4 Copy Propagation

Next we perform a data flow analysis on the various variables to check for their liveness. Here, we also built Use-Definition Chains (UD Chain) for each use of each

variable. These were structured as hash tables where the symbol (or location variable) and their respective definitions were used as keys and array lists of all the corresponding uses was present as the value.

Based on the above program, we had the following result obtained from common subexpression elimination:

```
main : EMPTY_STATEMENT
L0 : #_t0 = 2
L1 : #_t1 = get_int(#_t0,)
L2 : a = #_t1
L3 : #_t2 = 3
L4 : #_t3 = get_int(#_t2,)
L5 : b = #_t3
L6 : #_t4 = 0
L7 : c = #_t4
L8 : #_t5 = 0
L9 : d = #_t5
L10 : #_t6 = a + b
L11 : #_t7 = #_t6
L12 : #_t8 = #_t6 * #_t7
L13 : c = #_t8
L14 : #_t9 = #_t6
L15 : #_t10 = #_t6
L16 : #_t11 = #_t9 / #_t10
L17 : d = #_t11
L18 : #_t13 = printf(#str_t12,c,)
L19 : #_t15 = printf(#str_t14,d,)
```

On running our copy propagation algorithm on this, we replace the occurrences of targets of direct assignments with their values. While this optimization did not directly reduce our code size, it lead to a small number of reduction in the number of the operations being performed, and acted as an enabling optimization for dead code elimination, which had very significant impact on performance of our compiler.

```
main : EMPTY_STATEMENT
L0 : #_t0 = 2
L1 : #_t1 = get_int(2,)
L2 : a = #_t1
L3 : #_t2 = 3
L4 : #_t3 = get_int(3,)
L5 : b = #_t3
L6 : #_t4 = 0
L7 : c = 0
L8 : #_t5 = 0
L9 : d = 0
L10 : #_t6 = #_t1 + #_t3
L11 : #_t7 = #_t6
L12 : #_t8 = #_t6 * #_t6
L13 : c = #_t8
L14 : #_t9 = #_t6
L15 : #_t10 = #_t6
L16 : #_t11 = #_t6 / #_t6
```

```
19 L17 :  d = #_t11
20 L18 :  #_t13 = printf(#str_t12,#_t8,)
21 L19 :  #_t15 = printf(#str_t14,#_t11,)
```

2.5 Dead Code Elimination

Next we performed dead code elimination, where we removed local variables that are assigned a value but are not read by any subsequent instruction. This optimization was a lead up from common subexpression elimination and copy propagation, and had a very significant impact on the overall performance, and led to a decrease by about 15-20% in the total execution time of the benchmark programs. In the process of translating the given program from high level to low level intermediate representation, we added multiple lines repeated lines of code. These extra lines of dead code were also cleaned up by this process - that contributed to the total perceived speedup in execution time.

From the result of the copy propagation carried out previously, we had,

```
1  main :  EMPTY_STATEMENT
2  L0 :  #_t0 = 2
3  L1 :  #_t1 = get_int(2,)
4  L2 :  a = #_t1
5  L3 :  #_t2 = 3
6  L4 :  #_t3 = get_int(3,)
7  L5 :  b = #_t3
8  L6 :  #_t4 = 0
9  L7 :  c = 0
10 L8 :  #_t5 = 0
11 L9 :  d = 0
12 L10 :  #_t6 = #_t1 + #_t3
13 L11 :  #_t7 = #_t6
14 L12 :  #_t8 = #_t6 * #_t6
15 L13 :  c = #_t8
16 L14 :  #_t9 = #_t6
17 L15 :  #_t10 = #_t6
18 L16 :  #_t11 = #_t6 / #_t6
19 L17 :  d = #_t11
20 L18 :  #_t13 = printf(#str_t12,#_t8,)
21 L19 :  #_t15 = printf(#str_t14,#_t11,)
```

After eliminating all the dead stores, we can see that the code size, especially for this particular example from the test suite is considerably reduced, and is as follows:

```
1 main :  EMPTY_STATEMENT
2 L1 :  #_t1 = get_int(2,)
3 L4 :  #_t3 = get_int(3,)
4 L10 :  #_t6 = #_t1 + #_t3
5 L12 :  #_t8 = #_t6 * #_t6
6 L16 :  #_t11 = #_t6 / #_t6
```

```
L18 :  #_t13 = printf(#str_t12,#_t8,)
L19 :  #_t15 = printf(#str_t14,#_t11,)
```

In this particular case, we observed a reduction in the size of the generated code by approximately a factor of 14th. This further confirms the utility of this optimization and the applicability across various different programs.

2.6 Constant Folding

We perform constant folding at the low level IR stage, where we evaluate and compute constant expressions at compile time rather than computing them at runtime. This is done to avoid repeated and multiple computations, especially inside for loops, such as are in the provided benchmarks and testing suites.

For example, in the following case, constant folding saves 3 multiplication evaluations for each iteration of the loops, leading to a total reduction in about 300000 instructions getting executed. Because of the nature of programs, and presence of repeated computations in loops, this led to a very significant improvement in compiler performance.

```
void convert2RGB () {
  int row, col;
  for (row = 0; row < rows; row += 1) {
    for (col = 0; col < cols; col += 1) {
            q = image[((row * 2193) + (col * 3)) + 2] *
                (1024 * 60 - image[((row * 2193) + (col * 3)) + 1] * f)
                   / (1024 * 60 * 4);
            t = image[((row * 2193) + (col * 3)) + 2] *
                (1024 * 60 - image[((row * 2193) + (col * 3)) + 1] * (60
                    - f)) / (1024 * 60 * 4);

  }
}
```

Upon application of constant folding, followed by code generation, this program was translated to the following assembly code, where we can observe that the total number of multiplication computations relating to constants have been considerably reduced and the multiplication product for $(1024 * 60 * 4)$, that is 245760 has been introduced in the generated code. Note that the result shown below is a small snippet of a much larger generated code for the benchmark program.

```
    movq   -784(%rbp), %r11
    imulq  %r10, %r11
    movq   %r11, %r10
    movq   %r10, -792(%rbp)
    movq   $245760, %r10
    movq   %r10, -800(%rbp)
    movq   $245760, %r10
```

```
8       movq    -792(%rbp), %r11
9       movq    %r11, %rax
10      cqo
11      idivq   %r10
12      movq    %rax, %r10
13      movq    %r10, -808(%rbp)
14      movq    -808(%rbp), %r10
15      movq    %r10, -176(%rbp)
16      movq    $2193, %r10
17      movq    %r10, -816(%rbp)
18      movq    $2193, %r10
19      movq    -16(%rbp), %r11
20      imulq   %r10, %r11
21      movq    %r11, %r10
22      movq    %r10, -824(%rbp)
23      movq    $3, %r10
24      movq    %r10, -832(%rbp)
25      movq    $3, %r10
26      movq    -40(%rbp), %r11
27      imulq   %r10, %r11
```

2.7 Algebraic Simplification

```
1 i+0 = 0+i = i-0 = i
2 0 - i = -i
3 i*1 = 1*i = i/1 = i
4 i*0 = 0*i = 0
5 -(-i) = i
6 i + (-j) = i - j
7 b || true = true || b = true
8 b || false = false || b = b
```

Algebraic Simplification is the process by which we invoke basic algebraic identifies for to simplifying expressions or to determine whether 2 expressions are equivalent. Boolean algebraic simplifications can be particularly as they can sometimes illuminate that certain conditional branch will always evaluate to the same condition. This would enable the compiler to be able to swap out the conditional branch with an unconditional branch saving several lines of code in the process.

2.8 Unreachable Code Elimination

To minimize cache misses to whatever extent possible, we reduced the size of the code and implemented unreachable code elimination as an optimization to reduce the overall size of our code base and get rid of all the code blocks in the control flow graph (CFG) that can't ever be reached during the execution of the program, on starting

with the basic block containing the entry point for the function, and on following the directed edges from one basic block to another. This provided our compiler with a minor speedup. URE is performed at the low level IR stage via a breadth first search on the CGF, to check for connectivity.

```
f : EMPTY_STATEMENT
L0 : #_t0 = 0
L1 : i = #_t0

FOR_COND_L2 : EMPTY_STATEMENT
L4 : #_t1 = 1000000
L5 : #_t2 = i < #_t1
L6 : if #_t2 goto FOR_L3

L7 : goto END_FOR_L3

FOR_L3 : EMPTY_STATEMENT
L8 : #_t3 = 2
L9 : #_t4 = A[i] * #_t3
L10 : A[i] = #_t4
L11 : #_t5 = 1
L12 : #_t6 = i + #_t5
L13 : i = #_t6
L14 : goto FOR_COND_L2

L15 : EMPTY_STATEMENT
L16 : EMPTY_STATEMENT
L17 : EMPTY_STATEMENT
L18 : EMPTY_STATEMENT
L19 : EMPTY_STATEMENT

END_FOR_L3 : EMPTY_STATEMENT
```

In the low level IR, the set of empty (deleted/removed) statements from $l15$ and $L1$ is here the unreachable code basic block is removed, as they occur after a continue statement in the following example program and will be executed.

```
extern printf();
int A[1000000];
void f() {
  int i;
  for (i = 0; i < 1000000; i += 1) {
    A[i] = A[i] * 2;
    continue;
    printf("This code is unreachable");
  }
}
```

2.9 Register Allocation

We used a heuristic based greedy algorithm along with the use-def chains of all variable to evaluate the most used variables and locations, and allocate them to registers, thus attempting to keep the value of a variable in a register for as long as possible. Given that registers have significantly faster execution times for instructions, storing variables, large constants and values in registers also reduced the total number of reads and writes we had to make to the stack. This lead to extremely significant improvement in performance against benchmark programs and test suites, especially for programs with loops and multiple definitions and consequent uses of variables. Since we only allocated callee-saved registers for this purpose, calling conventions were not maintained and thus the optimization is extremely general and works independent of the program. For example, if we have the following unoptimized snippet of assembly code:

```
1     # c = a + b,
2     movq   -24(%rbp), %r10
3     movq   -16(%rbp), %r11
4     addq   %r10, %r11
5     movq   %r11, -8(%rbp)
```

We significantly optimize runtime by allocating registers to each of the three variables and eliminate all accesses to the stack, as shown below:

```
1     movq   %rbx, %r10
2     movq   %r13, %r11
3     addq   %r10, %r11
4     movq   %r11, %r12
```

As we realized that increasing the number of available register for allocation would have a significantly positive impact on performance, we made the $\%rbx$ register available as that wasn't being used for any other purposes.

2.10 Code Generation Optimizations

In trying to make our code run faster, we noticed that the size of the generated code, in terms of number of lines, has an impact on its performance, at least to some extent. We put in an effort to make the generated code as compact as possible. We did this by trying to take advantage of our register allocation to compress some operations that would use multiple lines had they been applied on stack locations. The following snippet of code, for example, shows how we can do a arithmetic operation in much fewer lines of code once we know what register the operands are mapped to after the register allocation.

```
1
2     # an assembly code for c = a + b,
3     # where c is stored in the stack location -8(%rbp)
```

```
    # a to -16(%rbp) and b to -24(%rbp)
    movq   -24(%rbp), %r10
    movq   -16(%rbp), %r11
    addq   %r10, %r11
    movq   %r11, -8(%rbp)
```

Applying register allocation in the above code will change the code to:

```

    # an assembly code for c = a + b,
    # where c is mapped to the register %r12
    # a to %r13 and b to %rbx
    movq   %rbx, %r10
    movq   %r13, %r11
    addq   %r10, %r11
    movq   %r11, %r12
```

Next we apply codegeneration optimization techniques to remove unncesscesary lines and make the code even shorter

```

    # an assembly code for c = a + b,
    # where c is mapped to the register %r12
    # a to %r13 and b to %rbx
    movq   %r13, %r11
    addq   %rbx, %r11
    movq   %r11, %r12
```

Over several thousands of lines of code, such small improvements add up to give us a much leaner code, that sometimes results in a slight improvement in performance, possibly due to better cache hit ratio.

---

3 Full Optimizations

On running our compiler with full optimizations enabled, we run our implemented optimizations in the following order (some optimizations are run multiple number of times as explained below):

1. Parallelization
2. Canonicalization of expressions
3. Algebraic Simplification
4. Common Subexpression Elimination
5. Copy Propagation

6. Constant Folding
7. Common Subexpression Elimination
8. Copy Propagation
9. Dead Code Elimination
10. Unreachable Code Elimination
11. Register Allocation

In determining the order to perform these optimizations, we first took into consideration which Intermediate Format they would need to be applied to. Both Parallelization and Canonicalization of Expressions (CE) need to be preformed on the High-Level IR because they require specific information about the program's implementation. Since much of this information would get lost after transitioning from the the High-Level IR into the Lower-Level IR, we knew that Parallization and Canonicalization of Expressions be preformed first. However, the decision of determining whether Parallelization or CE should come first ultimately to a matter of convinience. That is, our implementation of parallelism was particularly sensitive the format of algebraic expressions such that we decided that it should be preformed prior to CE.

Once the High-Level optimizations were preformed, we would convert the program to a Lower-Level IR. In the Lower-Level IR, we could now preform Algebraic Simplifications, CSE, Copy Propogation, and Constant Folding. To decide on the order for these optimizations, we considered both the benefit from running the optimization itself and the how the optimization would transform the program (i.e. did the transformed program actually make more opportunities for future optimizations available). We found that while Alegbraic Simplifications did not significantly reduce the size or the runtime of the program, it did increase the number of common subexpressions throughout the program. In addition, and not as much of a surprise, we found a similar synergestic affect in running Copy Propogation after CSE and Dead Code Elimination after that.

As can be observed from the listings of our optimizations above, we found that executing CSE and CP multiple times more impactful than just running either once. This was largely due to the fact that after CSE was preformed, it could take multiple iterations of CP to propogate all variable copies that now existed throughout the program. We decide to run CSE multiple times as well since, after CP, we sometimes found that new common subexpressions were created and we wanted to optimize those as well. After running the previous optimizations multiple times, multiple superfluous variables and assignments would exist. As a result, we decided to finished off our Lower-Level IR optimizations with Dead Code Elimination and Unreachable Code Elimination.

The last optimization that we performed was Register Allocation. We necessarily had to hold off this optimization until last because it gets performed on our absolute lowest Intermediate Representation. In addition to this, we decide that it simply

did not make sense to allocate register once, perform some optimization that would further transform the program, only to have perform register allocation again.

---

4 Challenging Problems Faced

One of the largest challenges that we faced while attempting to leverage parallel execution for methods with loops that have iterations independent of each other, for this design, was to ensure that unsafe/non-parallelizable loops don't get parallelized, as this could lead to incorrect execution and results, or extremely slow timings, given the cost of the system calls that are made by the 6036.*c* library. We addressed this problem by developing an extremely restrictive criteria for functions and the loops they contain, to be parallelized. Though this caused a lot of completely parallelizable loops to get skipped over, we were able to avoid incorrect evaluations.

---

5 Command Line Options

5.1 Turing Optimizations On

To turn on various optimizations, the following flags may be added, in any order, to turn on their corresponding optimizations.

- `--opt=cse` for common subexpression elimination
- `--opt=cp` for copy propagation
- `--opt=dce` for dead code elimination
- `--opt=ure` for unreachable code elimination
- `--opt=reg` for register allocation
- `--opt=parallel` for loop parallelization
- `--opt=as` for canonicalization and algebraic simplification
- `--opt=cf` for constant folding

To turn on all optimizations and run them in order, as described in the section on full optimizations [3], the `--opt=all` flag may be used, as follows:

```
$ ./run.sh --target=assembly --opt=all <filename> -o <outputname>
```

5.2 Running Generated Code

The $x86-64$ Assembly code generated by our compiler may be run using *gcc* to execute the program and get results, as follows:

```
$ gcc <filename> -o <outputname> -L /<ROOT_DIRECTORY_PATH>/tests/optimizer/lib/
       -l6035 -lc -lpthread
$ ./<outputname>
```