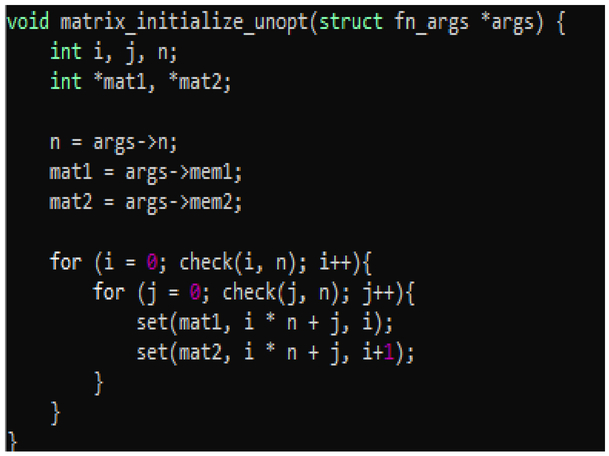
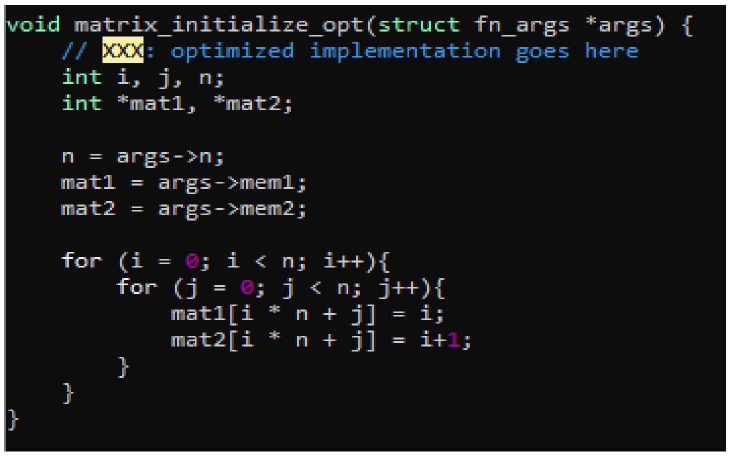
Sarah Huang
COSC 461: Intro to Compilers
December 9, 2022

Copt Project Report

1. **Descriptions of the optimizations you implemented for each operation, as well as any difficulties or challenges you had in implementing your approach.**



```
void matrix_initialize_unopt(struct fn_args *args) {
    int i, j, n;
    int *mat1, *mat2;

    n = args->n;
    mat1 = args->mem1;
    mat2 = args->mem2;

    for (i = 0; check(i, n); i++){
        for (j = 0; check(j, n); j++){
            set(mat1, i * n + j, i);
            set(mat2, i * n + j, i+1);
        }
    }
}
```

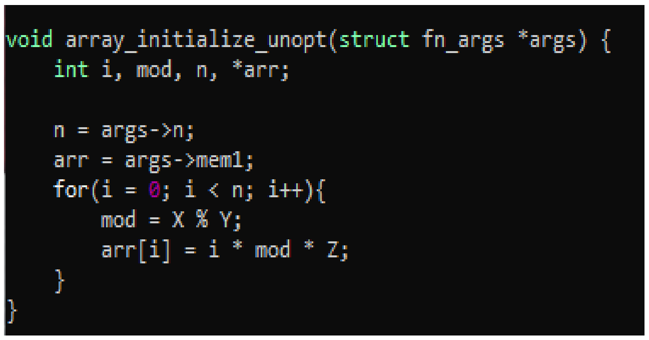


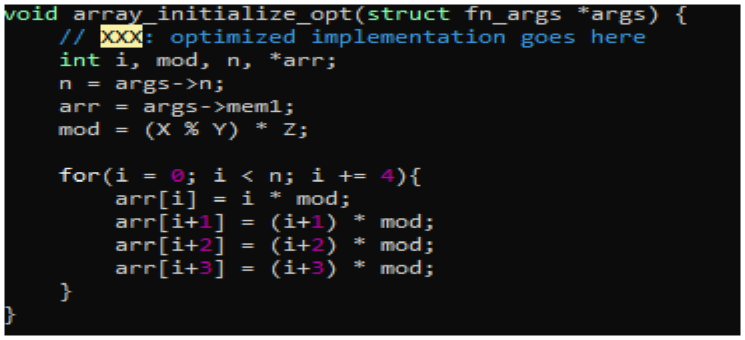
```
void matrix_initialize_opt(struct fn_args *args) {
    // XXX: optimized implementation goes here
    int i, j, n;
    int *mat1, *mat2;

    n = args->n;
    mat1 = args->mem1;
    mat2 = args->mem2;

    for (i = 0; i < n; i++){
        for (j = 0; j < n; j++){
            mat1[i * n + j] = i;
            mat2[i * n + j] = i+1;
        }
    }
}
```

Matrix Initialize

I only used the procedure integration or inlining optimization. Instead of using the functions `check()` and `set()`, I replaced the function calls with the bodies of the procedures.

There was no difficulty with implementing this approach. This operation was the easiest to identity an opportunity for optimization.


```
void array_initialize_unopt(struct fn_args *args) {
    int i, mod, n, *arr;

    n = args->n;
    arr = args->mem1;
    for(i = 0; i < n; i++){
        mod = X % Y;
        arr[i] = i * mod * Z;
    }
}
```

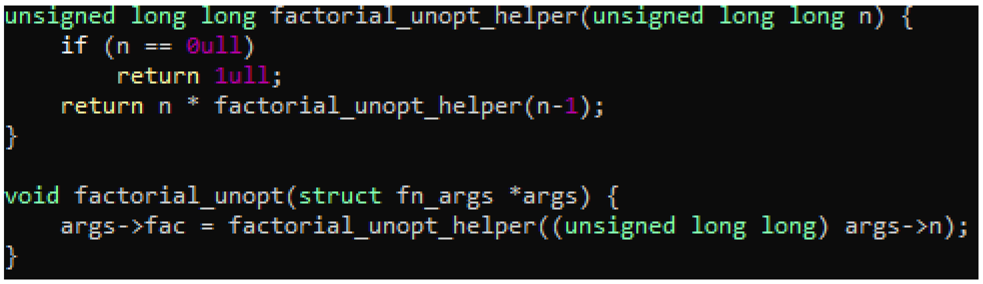

```
void array_initialize_opt(struct fn_args *args) {
    // XXX: optimized implementation goes here
    int i, mod, n, *arr;
    n = args->n;
    arr = args->mem1;
    mod = (X % Y) * Z;

    for(i = 0; i < n; i += 4){
        arr[i] = i * mod;
        arr[i+1] = (i+1) * mod;
        arr[i+2] = (i+2) * mod;
        arr[i+3] = (i+3) * mod;
    }
}
```

Array Initialize

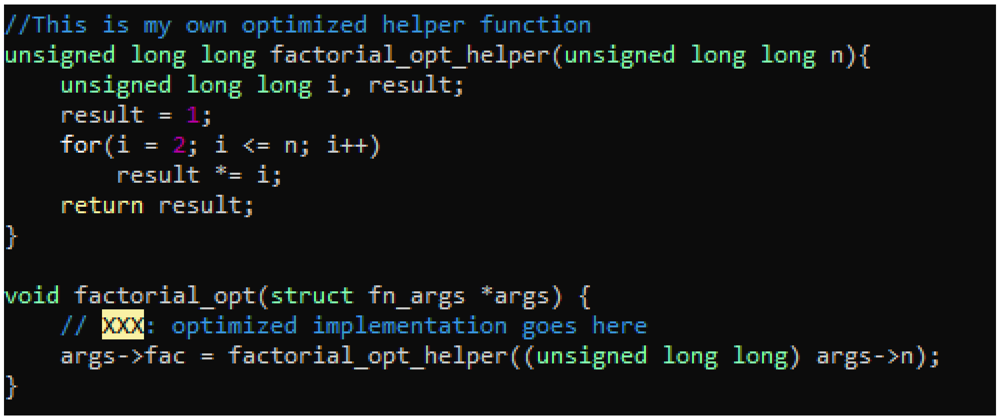
I implemented loop unrolling using an unroll factor of 4. I made copies of the loop body so the for loop doesn't require more loop branch testing. I also used loop invariant code motion because the array elements are being multiplied by a constant that does not need to be calculated repeatedly in the loop body. The loop invariant code motion is shown by the `mod = (X % Y) * Z`.

At first, it was a little challenging figuring out how to make it faster. Originally, I experimented with using a while loop and only decremented one by one. I also tried implementing a for loop like shown above beforehand, but I incremented the variable `i` in the loop body.

```
for(i = 0; i < n; i++){
    arr[i] = i * mod; i++;   //This is how I tried doing it originally
    arr[i] = i * mod; i++;
    ...
}
```



```
unsigned long long factorial_unopt_helper(unsigned long long n) {
    if (n == 0ull)
        return 1ull;
    return n * factorial_unopt_helper(n-1);
}

void factorial_unopt(struct fn_args *args) {
    args->fac = factorial_unopt_helper((unsigned long long) args->n);
}
```

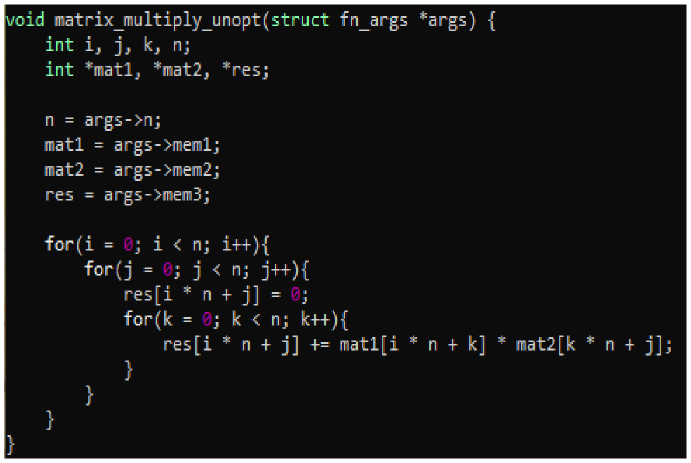


```
//This is my own optimized helper function
unsigned long long factorial_opt_helper(unsigned long long n){
    unsigned long long i, result;
    result = 1;
    for(i = 2; i <= n; i++)
        result *= i;
    return result;
}

void factorial_opt(struct fn_args *args) {
    // XXX: optimized implementation goes here
    args->fac = factorial_opt_helper((unsigned long long) args->n);
}
```

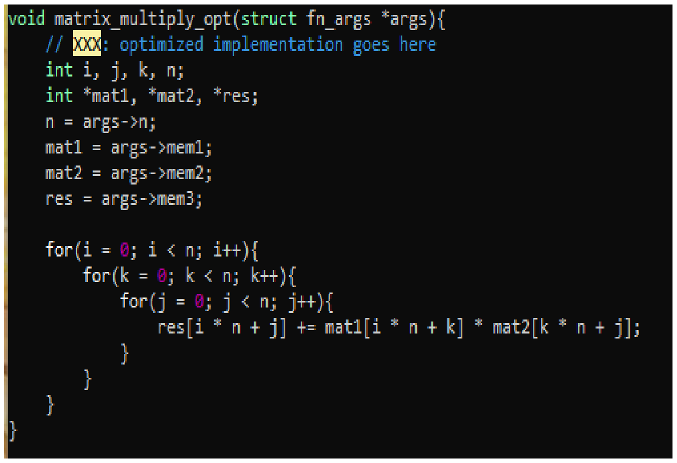
Factorial

I implemented tail recursion elimination by getting rid of the recursive call all together. Instead of calculating the factorial recursively, I did it iteratively with a for loop.

This operation took me a while to figure out despite it's simple solution because I thought I was tied down to putting all of my optimized code in `factorial_opt()`. While it is probably possible to place all the code in one function, I figured it was alright if I made my own helper function, so I could reuse the code for `factorial_unopt()` and focus more on fixing hte helper function.



```c
void matrix_multiply_unopt(struct fn_args *args) {
    int i, j, k, n;
    int *mat1, *mat2, *res;

    n = args->n;
    mat1 = args->mem1;
    mat2 = args->mem2;
    res = args->mem3;

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            res[i * n + j] = 0;
            for(k = 0; k < n; k++){
                res[i * n + j] += mat1[i * n + k] * mat2[k * n + j];
            }
        }
    }
}
```



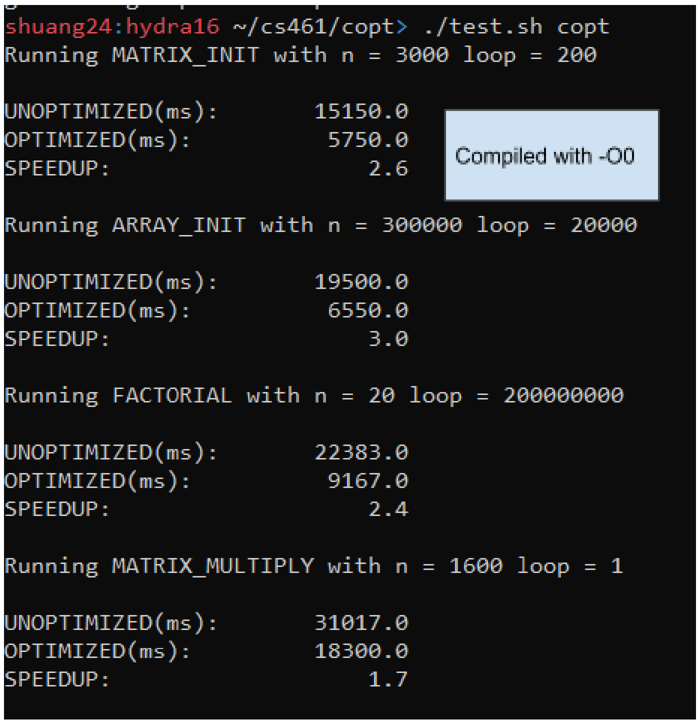
```c
void matrix_multiply_opt(struct fn_args *args){
    // XXX: optimized implementation goes here
    int i, j, k, n;
    int *mat1, *mat2, *res;
    n = args->n;
    mat1 = args->mem1;
    mat2 = args->mem2;
    res = args->mem3;

    for(i = 0; i < n; i++){
        for(k = 0; k < n; k++){
            for(j = 0; j < n; j++){
                res[i * n + j] += mat1[i * n + k] * mat2[k * n + j];
            }
        }
    }
}
```

Matrix Multiplication

I implemented loop interchange by swapping the j and k for loop arguments. I also did dead assignment elimination because `res[i * n + j] = 0;` was not needed since the array elements are intialized later on in the nested loops.

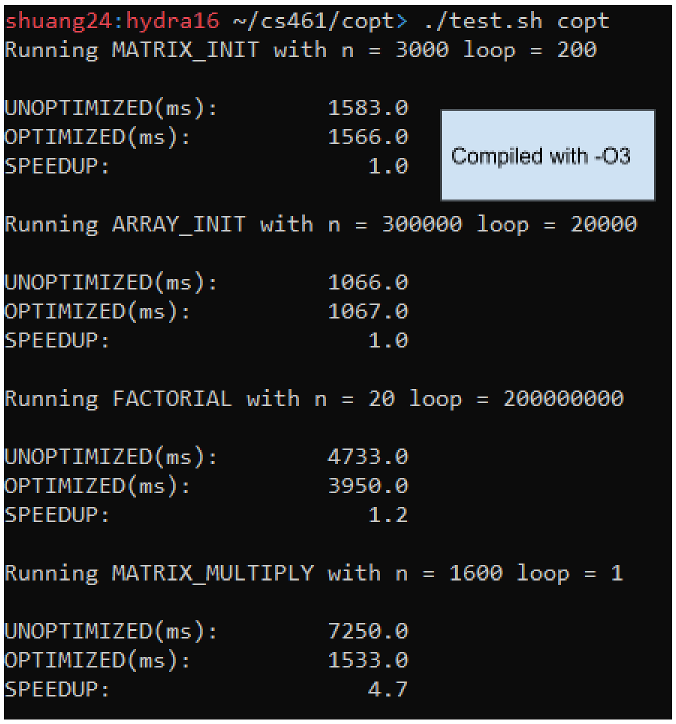
This operation was easy to optimize because the main change was swapping two for loops. It's also really surprising how one little change cuts the time significantly.

2. Comparisons and analysis of the results with your copt program and the provided reference executables.
Overall, the speed performance is roughly the same with my copt program and the reference executables.



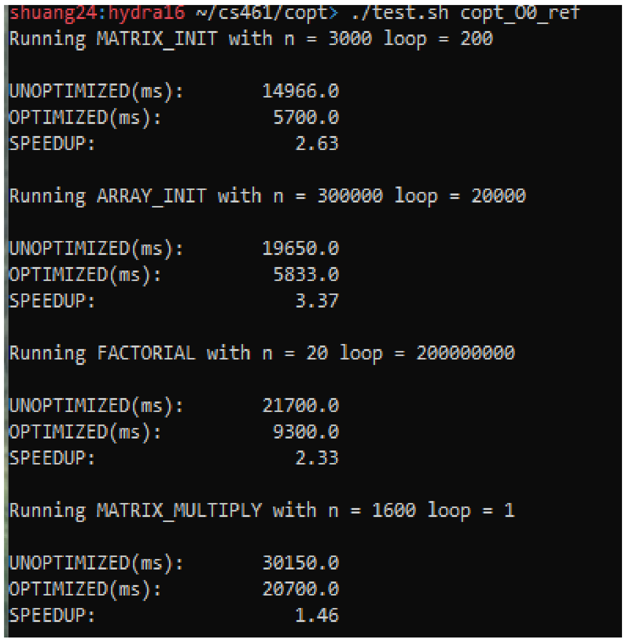
```
shuang24:hydra16 ~/cs461/copt> ./test.sh copt
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):       15150.0
OPTIMIZED(ms):          5750.0
SPEEDUP:                   2.6

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):       19500.0
OPTIMIZED(ms):          6550.0
SPEEDUP:                   3.0

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):       22383.0
OPTIMIZED(ms):          9167.0
SPEEDUP:                   2.4

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):       31017.0
OPTIMIZED(ms):         18300.0
SPEEDUP:                   1.7
```

Compiled with -O0

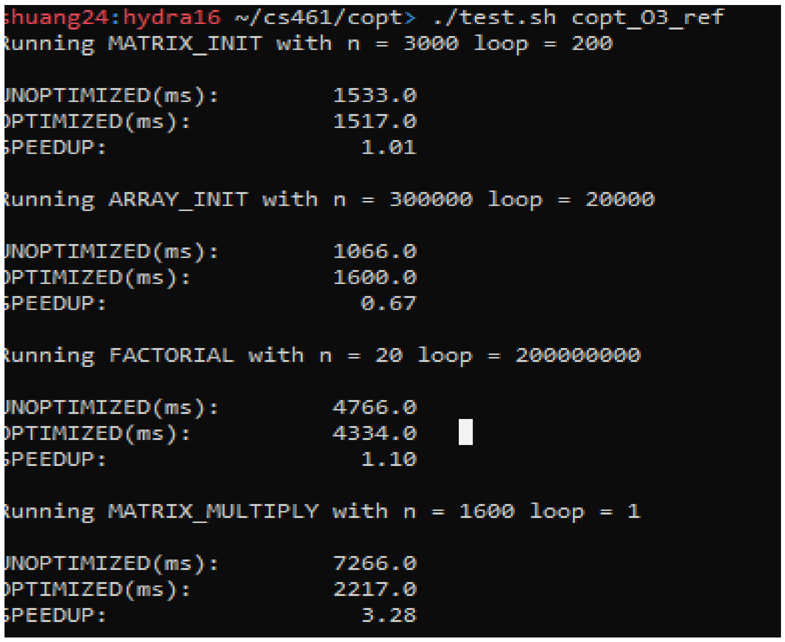


```
shuang24:hydra16 ~/cs461/copt> ./test.sh copt
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):        1583.0
OPTIMIZED(ms):          1566.0
SPEEDUP:                   1.0

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):        1066.0
OPTIMIZED(ms):          1067.0
SPEEDUP:                   1.0

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):        4733.0
OPTIMIZED(ms):          3950.0
SPEEDUP:                   1.2

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):        7250.0
OPTIMIZED(ms):          1533.0
SPEEDUP:                   4.7
```

Compiled with -O3



```
shuang24:hydra16 ~/cs461/copt> ./test.sh copt_O0_ref
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):       14966.0
OPTIMIZED(ms):          5700.0
SPEEDUP:                  2.63

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):       19650.0
OPTIMIZED(ms):          5833.0
SPEEDUP:                  3.37

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):       21700.0
OPTIMIZED(ms):          9300.0
SPEEDUP:                  2.33

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):       30150.0
OPTIMIZED(ms):         20700.0
SPEEDUP:                  1.46
```



```
huang24:hydra16 ~/cs461/copt> ./test.sh copt_O3_ref
unning MATRIX_INIT with n = 3000 loop = 200

NOPTIMIZED(ms):        1533.0
PTIMIZED(ms):          1517.0
PEEDUP:                  1.01

unning ARRAY_INIT with n = 300000 loop = 20000

NOPTIMIZED(ms):        1066.0
PTIMIZED(ms):          1600.0
PEEDUP:                  0.67

unning FACTORIAL with n = 20 loop = 200000000

NOPTIMIZED(ms):        4766.0
PTIMIZED(ms):          4334.0
PEEDUP:                  1.10

unning MATRIX_MULTIPLY with n = 1600 loop = 1

NOPTIMIZED(ms):        7266.0
PTIMIZED(ms):          2217.0
PEEDUP:                  3.28
```

3. In your comparisons and analysis, please answer the following questions:
(a) Which optimizations are most effective for each operation?
For matrix initialization, the most effective optimization for me was procedure integration/inlining.
For array initialization, the most effective optimization for me was loop unrolling.
For factorial, the most effective optimization for me was tail recursion elimination.
For matrix multiplication, the most effective optimization for me was loop interchange.

(b) Some optimizations are less effective when -O3 is enabled. Why do you think this is the case?
It could be because -O3 uses more flags that change the performance in ways we don't expect. When I was optimizing array initialization, I tried increasing the loop unroll factor to 10 to make it faster. Compared to an unroll factor of 4, the speeds were similar with the -O0 tag, but an unroll factor of 10 was slower than a factor of 4 with the -O3 tag. Some extreme optimization did not like that I put more in the loop body.

(c) Are any optimizations just as effective (or even more effective) when -O3 is enabled? If so, which optimization(s)? Why do you think these optimization(s) are still effective when -O3 is enabled?
The tail recursion elimination worked alright maybe because recursion is slow in general. Loop interchange was very effective for matrix multiplication even with the -O3 tag possibly because the -O3 tags do not help with spatial locality. The order of the for loops accessing memory is up to the developer.