# Optimization Approaches for Ray Tracing Software

Jeffrey Tamashiro
*ICS 432 •Casanova •Spring 2016*

***Abstract*—This project contains 10 versions of a ray tracing software with version 0 as the base code. Versions 1-7 make incremental changes to the original in order to observe changes in performance. Versions 8 & 9 are attempts to make the fastest possible run time based on observed knowledge from the previous versions.**

## I. Introduction

This project is meant to serve as a test for different approaches to speeding up code as learned in ICS 432.

### A. Testing Method

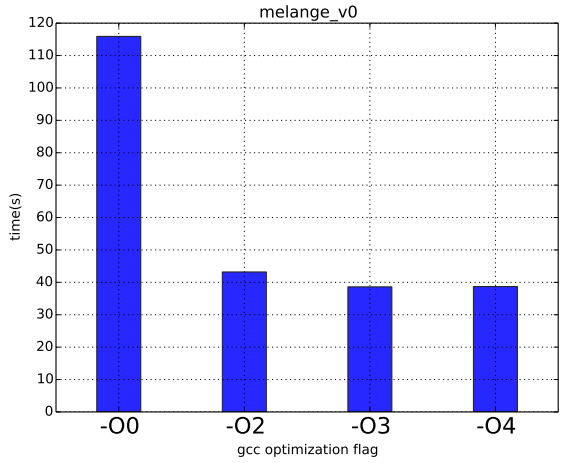
Time tests were conducted for each of the 10 versions, using four optimization compile time directives (O0, O2, O3, O4). Times were based on the mean value over 5 times running the code.
Comparing the different tiers of optimization was crucial for understanding how the compiler worked and what optimization methods may already be happening with a smarter compiler.

### B. Profiling

Profiling Tools gprof and valgrind were used mainly to examine call graphs and see which functions were being called more frequently and computationally dominant. This provided clues for how to proceed in speeding up the code.

## II. Version 0

### A. Results




This is the base time for comparing against other versions. There is a major difference between no optimization and some, but not much among the different levels.

## III. Version 1

### A. Utilizing Concurrent Threads

The first most obvious way to performance gain seemed to be taking the sequential execution and simply dividing the work among CPU cores. Based on the idea that a ray tracer works on each pixel, it was just a matter of finding the point in the code where those pixels were being looped through and putting an OpenMP pragma around it.

### B. Results

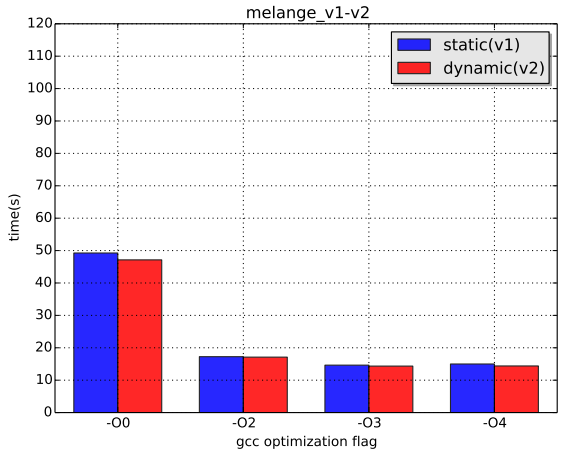
With very little effort there is a significant performance gain. It was tempting not to call the project finished here.

## IV. Version 2

### A. Dynamic Scheduling

The ray tracer should be tracing light paths which uniquely bounce of objects and reflect so each pixel's computation may vary in computational complexity. For this reason it seemed prudent to try a version with OpenMP dynamic scheduling instead of static.

### B. Results




The dynamic has just a slight edge over the static.

## V. Version 3

### A. Faster Power Function

Based on the profiler call graphs, the library function `pow` was called frequently and seemed to be computationally heavy. So, instead of writing my own version, I looked online to see that much smarter people had already done it.

### B. Results

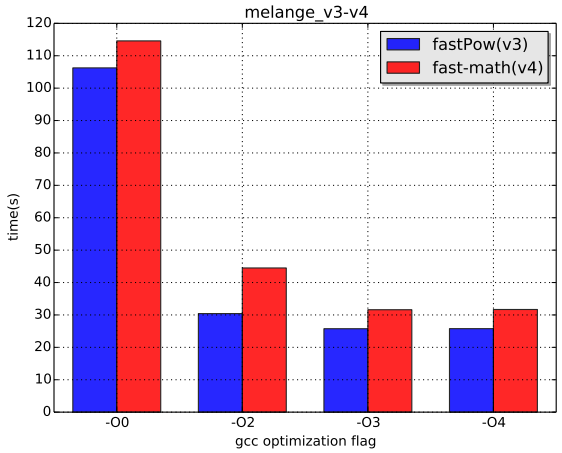
This had exceptional results with a potential flaw. This did not produce the exact same image as the base program. It was just a little different, most likely indistinguishable by a human eye. So, the idea of correctness is raised and depending on the program's purpose, could be enforced or dismissed.

## VI. Version 4

### A. Fast Math Optimization

Looking at the code through profiling tools, it became clear that most of the time was spent doing floating point operations. I found a compiler flag `-ffast-math` that I had not seen before.

### B. Results




The fast math inclusion gave a decent improvement indicating there may be special math based compiler optimizations not included in -O4.

## VII. Version 5

### A. Manually Inlining

One function that was highlighted by the profiler as weighty was a simple vector dot product function called `vecdot`. So at that point in the code, I tried manually expanding it out in place of the function call.

### B. Results

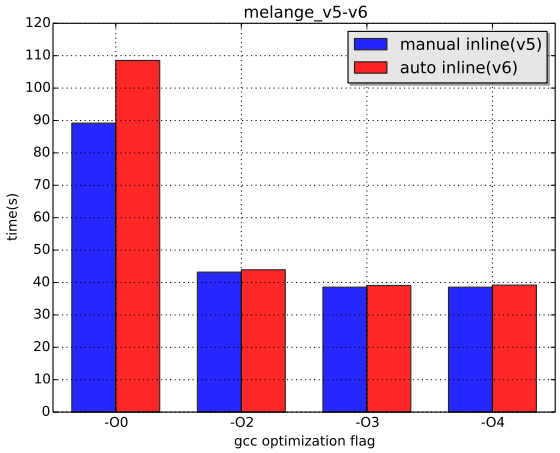
The manual did well for the -O0 flag which means it's likely the -O2 and above optimizations are doing things like inlining already.

## VIII. Version 6

### A. Auto Inlining

I also found that C provides a keyword `inline` that compels the compiler to do this automatically.

### B. Results




Both manual and automatic inlining did not offer much improvement over what the compiler was already doing with -O2 and above.

## IX. Version 7

### A. Branch Prediction

The profiler also showed another weighty part of the code which had several branching if-else statements. The most obvious fix was moving an if statement containing a return clause up to the top of the branch so the function could end more quickly in some cases.

### B. Results

Basically, the results were identical to the default version even for 0 optimization. So, this didn't really do anything in this case.

## X. Version 8

### A. Final Version Precise

This version used OpenMP with dynamic, fast-math flag, auto inlining, but not the faster power function. So this version should be the fastest that gives completely correct renderings.

### B. Results

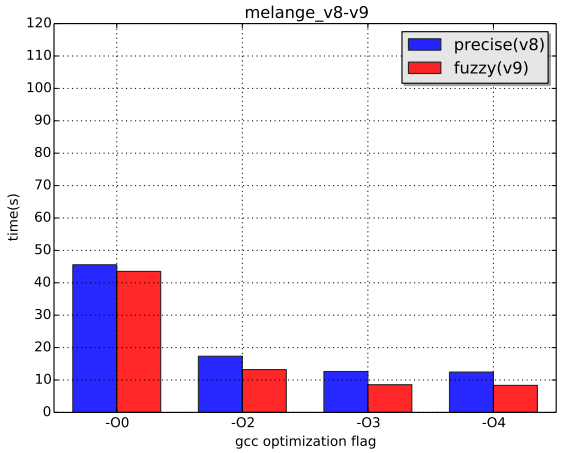
This gave a decent overall increase in speed, it was a little over 3 times faster than version 0.

## XI. Version 9

### A. Final Version Fuzzy

For this final version, I accepted the margin of error and tried to the fastest runtime possible.

### B. Results



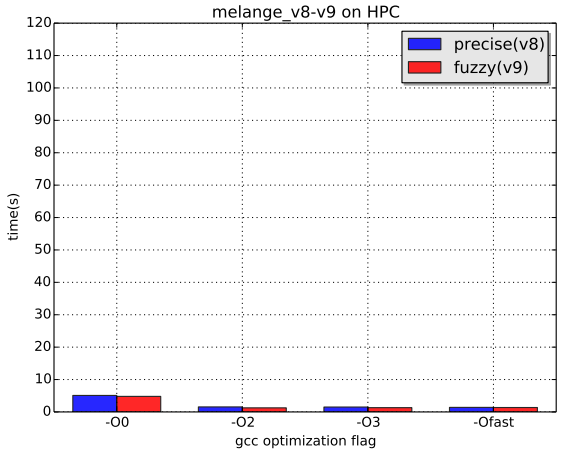

At the expense of a perfectly correct ray tracing image, the fast power function really did give a significant speed boost in the end.

## XII. Version X

### A. Running on an HPC

Just for fun, I ran versions 8 and 9 on the Cray computer on 20 cores with 20 OpenMP threads.

### B. Results




Although this doesn't take into account the time spent waiting in the queue.

## XIII. Conclusion

### A. Outcome based Optimization

I found myself wondering about if it was too specific to focus on the application running for a single input file. I really scrutinized over the profiling information and I can't help wonder if my code was generally faster or simply good at rendering these exact six colored balls. I probably could have used different setting files and fed them to the profilers, but a larger issue may be the difference between code that can handle any possible task a step faster, or do the most commonly requested tasks much faster.

### B. Optimization as a Job

One of the major themes I found relevant for this assignment was the somewhat disjointed relationship between understanding the code and trying to best optimize it.
It was amazing how easy it was to get an initial large speedup knowing almost nothing and then digging in and agonizing over every tenth of a second thereafter. Like I mentioned earlier, it would have been satisfying to slap an omp pragma on it and call it a day.
On the other hand, I do feel that it would be worthwhile to write code from the ground up to be more conducive to being optimized and parallelized. It would be especially interesting to work with a ray tracing expert to build something like that.