CS 243 Group Project:

Experiments with g++ Compiler Optimization Flags

Jacob Heller
Tyler Howe
Bailey Kong

Intro and Motivation

The goal of our project was to investigate the impact of various g++ compiler optimization flags by measuring the performance of C and C++ code compiled with different flags. As the vast majority of C and C++ coders just select the O1, O2 or O3 flags when compiling, we decided to explore the performance impact of specific compiler flags in different situations. In addition to creating more informed software developers, we hoped that this learning experience would help us make better decisions when deciding how to write code and when selecting options during compilation.

We accomplished this by compiling a number of different benchmark algorithms using each individual flag from -O1 optimization. We wrote some of these benchmarks, while we selected others from code online [1]. After doing this, we created a benchmark suite to run our code with different optimization flags and measure the execution time of these compiled programs. Our work showed which optimization flags were effective and ineffective at improving code execution.

Benchmarks and Flags

We used a variety of different algorithms and simple code routines as benchmarks. These were compiled using different optimization flags, and each benchmark's execution time was recorded. We chose to use the complete list of optimizations as turned on by the -O1 flag for our experiments. The "control" result was for -O0 (no optimization), so we could exactly determine the effect that each flag had on the output. Since our goal was to determine the *individual* effects of each flag, we compiled each benchmark with exactly one optimization.

After compiling with each of these flags (106 of them), we noticed that some results were not unique. In fact, the md5 checksums of the executables for some of the flags were identical to the -O0 executable. This meant that some flags resulted in no change to the underlying assembly code. Therefore, we filtered out the optimizations that fell into this category and report only the ones that produced different assembly code. This list of optimizations varies a bit between benchmarks. We also discovered that in the GCC documentation [2] where the suppose list of flags are used for O1 is far from complete. This page says that O1 uses 31 optimizations, but in reality the number is closer to 100. We discovered this by compiling with `gcc -O1 <src_code.c> -S -fverbose-asm.` We created a list of roughly 100 flags that we determined were the typical ones used by O1 and used that as our list.

Listed below are the benchmarks used, their operation, and the flags that impact the assembly code. Not included in the list of flags is -O0, -O1, -O2, and -O3, which were run for every benchmark function.

Flag Descriptions:

- **-mieee-fp:** Controls whether or not to use IEEE floating-point comparisons.
- **-fcombine-stack-adjustments:** Controls whether to trace stack adjustments (pushes and pops) and stack memory references to try to find ways of combining them.
- **-fipa-pure-const:** Controls whether to discover which functions are pure or constant.
- **-fmerge-constants:** Controls whether to try to merge identical constants (string constants and floating-point constants) across compilation units.
- **-fomit-frame-pointer:** Removes the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an

extra register available in many functions.

- **-ftoplevel-reorder:** Controls whether to reorder top-level functions, variables, and ASM statements.
- **-ftree-parallelize-loops=n:** Parallelize loops by splitting their iteration space to run in n threads. This is only possible for loops whose iterations are independent and can be arbitrarily reordered.
- **-ftree-ter:** Performs temporary expression replacement during the static single assignment (SSA)->normal phase. Single use/single def temporaries are replaced at their use location with their defining expression.
- **-pseudo O1:** The list of flags found using `gcc -O1 <src_code.c> -S -fverbose-asm`.

Ackermann's Function (ack.cpp)
Computes Ackermann's function A(x, y) for x=3 and y=10. This function is a simple example of a total function that is computable but not primitive recursive. This function grows much faster than polynomials or exponentials and stresses the compiler's ability to do deep recursion efficiently.
Flags tested: `-fomit-frame-pointer, -mieee-fp`

Linear Array Access (arrayAccessLinear.c)
Sets values in an array by accessing elements linearly. Tests the compiler's ability to optimize array access when elements are referenced sequentially from within a loop.
Flags tested: `-fomit-frame-pointer, -ftree-ter, -mieee-fp, -ftree-parallelize-loops=1`

Random Array Access (arrayAccessRandom.c)
Sets values in an array by accessing elements randomly. Tests the compiler's ability to optimize array access when elements are referenced randomly from within a loop.
Flags tested: `-fomit-frame-pointer, -ftree-ter, -mieee-fp, -ftree-parallelize-loops=1`

Deep Recursion (deepRecursion.c)
Computes the length of a long string recursively. This function is a simple example of a large stack of recursive calls, which tests the compiler's ability to optimize a simple recursive function.
Flags tested: `-fomit-frame-pointer, -fmerge-constants, -mieee-fp, -ftree-parallelize-loops=1`

Float Division (floatDivision.c)
Continually computes the same value by dividing two floats repeatedly. This tests the compiler's ability to optimize floating point math as well as identify duplicate operations.
Flags tested: `-fomit-frame-pointer, -ftree-ter, -fmerge-constants, -mieee-fp, -ftree-parallelize-loops=1,`

List Operations (lists.cpp)
Performs various operations on a list. The operations are: Creating a list (L1) of 10k elements. Copying L1 to a new list (L2). Removing each item from the left side of L2 and appending it to the right side of L3

(preserving the original order). Removing each item from the right side of L3 and appending it to the right side of L2 (reversing the original order). Reversing L1 in place. Checking the size of a list, and comparing L1 and L2 for equality.
Flags tested: `-fomit-frame-pointer, -fipa-pure-const, -mieee-fp, -ftree-ter, -fcombine-stack-adjustments, -ftoplevel-reorder, -ftree-parallelize-loops=1`

Matrix Computations (matrix.cpp)
Performs a series of matrix computations on a 300x300 matrices (A and B) and 300x1 vector (x). The computations are: Initializing elements of matrices and vectors. Doing 5-pointed update (adjacent points around a matrix element), and 9-pointed update (adjacent + diagonal points around a matrix element). Also, doing multiplication of two matrices and multiplication of a matrix and a vector. This benchmark tests the compiler's loop optimizations.
Flags tested: `-fomit-frame-pointer, -ftree-ter, -mieee-fp, -ftree-parallelize-loops=1`

Nested Loops (nestloop.cpp)
Executes 7 nested loops. Each loop runs for 20 iterations, where the body of the innermost loop is an increment operations. This tests the efficiency of the code the compiler generates for loops.
Flags tested: `-fomit-frame-pointer, -mieee-fp, -ftree-ter, -ftoplevel-reorder, -ftree-parallelize-loops=1`

Quick Sort (quickSort.c)
Sorts an array of 16k integer numbers using the quick sort algorithm. This tests the compiler's ability to optimize a complex algorithm.
Flags tested: `-fomit-frame-pointer, -fipa-pure-const, -mieee-fp, -ftree-ter, -ftoplevel-reorder, -ftree-parallelize-loops=1`

Benchmark Suite

In order to run our benchmarks, we needed a way to accurately and reliably obtain performance metrics. To do this, we created a "benchmark suite" program, which we used to read the clock cycle before and after running a benchmark program. The benchmark suite was written in C to ensure that the clock cycle count was accurate and that the suite itself contributed minimum overhead to the result.

Fetching the clock cycle count was done by using an assembly call to read the *rdtsc* register, which contains a 64-bit unsigned integer representing the current clock cycle. Then, the suite used fork() to spawn a child process which executed the benchmark. Each benchmark file was compiled as a separate standalone executable for each optimization flag. To remove variability from the results, the suite repeated this process 20 times and produced a final result. For readability, the results were measured in thousands of clock cycles. The pseudocode for this execution is below:

```
runtime = 0
for 1:20,
        clk_start = get_rdtsc();
        // run benchmark
        …
```

```
        clk_end = get_rdtsc();
        runtime += (clk_end - clk_start) / 1000; // reduce # of digits
final_runtime = (runtime / 20);
```

Results

Below are the results and analysis for each benchmark. Speedup is calculated as (-O0 Result) / Result.

Benchmark Name	Compiler Flag	Result	-O0 Result	Speedup
ack	O3	84869.855	909134.619	10.712103
ack	O2	148684.851	909134.619	6.1145074
ack	O1	664146.003	909134.619	1.3688776
ack	pseudo O1	774164.963	909134.619	1.1743422
ack	fomit-frame-pointer	791935.596	909134.619	1.1479906
ack	mieee-fp	1038064.622	909134.619	0.8757977

Analysis

Ackermann's function creates a large recursion stack performing integer operations. Since operations were entirely integer related the inclusion of the -mieee-fp flag was not expected to impact performance, however, we actually a noticeable decrease in performance with it enabled. Enabling the -fomit-frame-pointer flag resulted in a speedup of 1.15, which occurred because an extra register was available. This speedup is comparable to the 1.17 speedup in pseudo O1, which had all of the flags enabled. This suggests that -fomit-frame-pointer provides most of the speedup possible at the -O1 optimization level. -O2 and -O3 provided further speedup by enabling more expensive optimizations, like global common subexpression elimination and predictive commoning optimization (i.e., reusing computations).

Benchmark Name	Compiler Flag	Result	-O0 Result	Speedup
arrayAccessLinear	O3	1758.087	2073469.724	1179.3897
arrayAccessLinear	O2	1790.08	2073469.724	1158.3112
arrayAccessLinear	O1	187674.432	2073469.724	11.048227
arrayAccessLinear	ftree-ter	2020346.233	2073469.724	1.0262943
arrayAccessLinear	pseudo O1	2022771.078	2073469.724	1.0250640
arrayAccessLinear	ftree-parallelize-loops=1	2065901.679	2073469.724	1.0036633
arrayAccessLinear	fomit-frame-pointer	2073532.379	2073469.724	0.9999698

arrayAccessLinear	mieee-fp	2079556.228	2073469.724	0.9970732

Analysis

As this benchmark looped through an array and assigned values sequentially, we once again, expect -mieee-fp to have negligible effect on the runtime, which is confirmed by our results. At the same time, none of the other flags had any effect on the runtime either. This is because the benchmark is so simple that there is very little to optimize. Setting the -ftree-parallelize-loops flag to 1 did not provide any speedup because the compiler could not determine if there was a dependency between array elements for different loop iterations. The -O2 and -O3 benchmarks provided speedup of over 1000 because the array assignments were unused and were completely removed from the final assembly instructions.

Benchmark Name	Compiler Flag	Result	-O0 Result	Speedup
arrayAccessRandom	O1	11099374.75	12210246.42	1.1000842
arrayAccessRandom	O2	11170726.49	12210246.42	1.0930575
arrayAccessRandom	ftree-ter	11260971.35	12210246.42	1.0842978
arrayAccessRandom	pseudo O1	11520031.942	12210246.42	1.0599143
arrayAccessRandom	O3	11527173.73	12210246.42	1.0592576
arrayAccessRandom	mieee-fp	11752603.14	12210246.42	1.0389397
arrayAccessRandom	ftree-parallelize-loops=1	11829442.64	12210246.42	1.0321912
arrayAccessRandom	fomit-frame-pointer	11877465.11	12210246.42	1.0280179

Analysis

Although the code of this benchmark is quite similar to that of the last one, the results were not. O2 and O3 are no longer 1000 times faster (meaning the array assignments were not completely removed from the generated assembly code). And for similar reasons, O1 is no longer 10 times faster. We believe the compiler's ability to analyze code is being thrown off by the use of rand. Random number generators are by nature unpredictable it is likely that the compile ceases analysis of code blocks that make use of them. As a result, the best improvement we got was 1.1 using O1 and O2.

Benchmark Name	Compiler Flag	Result	-O0 Result	Speedup
deepRecursion	O2	1934.366	10012.52	5.1761249
deepRecursion	O3	2048.622	10012.52	4.8874414
deepRecursion	O1	3184.915	10012.52	3.1437323
deepRecursion	fomit-frame-pointer	4954.642	10012.52	2.0208362

deepRecursion	pseudo O1	5027.932	10012.52	1.9913794
deepRecursion	fmerge-constants	5283.471	10012.52	1.8950648
deepRecursion	mieee-fp	5311.908	10012.52	1.8849197
deepRecursion	ftree-parallelize-loops=1	5659.529	10012.52	1.7691437

Analysis

DeepRecursion is a simple program that counts the length of a string using recursion. ftree-parallelize-loops=1 resulted in a 77% speedup, which is a bit confusing because this function has no loops. To the contrary, recursion has taken the place of where a loop would traditionally be. Perhaps, due to the simplicity of the function, the compiler realized that it could be converted to a loop and that's where the optimization happened. Once again, the fomit-frame-pointer resulted in a significant speedup since the program could make use of another register.

Benchmark Name	**Compiler Flag**	**Result**	**-O0 Result**	**Speedup**
floatDivision	O3	1640.992	5790187.184	3528.4676
floatDivision	O2	3255.003	5790187.184	1778.8577
floatDivision	O1	928959.56	5790187.184	6.2329809
floatDivision	ftree-ter	5624709.731	5790187.184	1.0294197
floatDivision	pseudo O1	5682262.668	5790187.184	1.0189932
floatDivision	ftree-parallelize-loops=1	5786626.522	5790187.184	1.0006153
floatDivision	fmerge-constants	5788955.037	5790187.184	1.0002128
floatDivision	fomit-frame-pointer	5815153.612	5790187.184	0.9957067
floatDivision	mieee-fp	5822831.666	5790187.184	0.9943937

Analysis

The float division benchmark repeatedly performs the same floating point division operation. It is therefore surprising that none of the single flags were able to provide any speedup. In O1, the compiler was able to identify that the same operation was being performed repeatedly, and it optimized this into one expression, resulting in a speedup of 6.23. It is unclear why pseudo-O1 was not able to perform this same optimization. Once again, O2 and O3 removed unused values and optimized the floating point operation out of the benchmark entirely.

Benchmark Name	**Compiler Flag**	**Result**	**-O0 Result**	**Speedup**
lists	O1	6877.268	16498.716	2.3990218

lists	O2	6912.603	16498.716	2.3867588
lists	O3	8253.873	16498.716	1.9989059
lists	pseudo O1	12868.962	16498.716	1.2820549
lists	fomit-frame-pointer	13517.282	16498.716	1.2205646
lists	fipa-pure-const	14050.002	16498.716	1.1742857
lists	mieee-fp	15998.36	16498.716	1.0312755
lists	ftree-ter	15998.718	16498.716	1.0312524
lists	fcombine-stack-adjustments	16222.699	16498.716	1.0170142
lists	ftoplevel-reorder	16448.798	16498.716	1.0030348
lists	ftree-parallelize-loops=1	16672.262	16498.716	0.9895907

Analysis

The lists benchmark performs many list operations using the built-in C++ list class. The -fipa-pure-const flag was able to provide a speedup of 1.17 because many list operations are “pure” functions, meaning the function only depends on its parameters. This leads to speedup because functions that are called multiple times can be optimized to one function call if the parameters are the same. For example, a loop that terminates based on the size of the list could be sped up by assigning a variable to list.size() and saving that value, rather than calling list.size() each time through the loop. It is interesting in this case that -O1 actually resulted in similar speedup to -O2 and -O3. This suggests that most of the optimizations that can be performed on loops only occur at the -O1 level.

Benchmark Name	**Compiler Flag**	**Result**	**-O0 Result**	**Speedup**
matrix	O2	89355.921	466722.566	5.2231857
matrix	O3	96617.965	466722.566	4.8305982
matrix	O1	160589.42	466722.566	2.9063096
matrix	ftree-ter	382666.024	466722.566	1.2196603
matrix	pseudo O1	389931.882	466722.566	1.1969336
matrix	mieee-fp	466328.995	466722.566	1.0008439
matrix	fomit-frame-pointer	466357.446	466722.566	1.0007829
matrix	ftree-parallelize-loops=1	468735.483	466722.566	0.9957056

Analysis

The matrix program creates three matrices and two vectors and does matrix-matrix and matrix-vector multiplication. It makes sense that mieee-fp does not result in a speedup this time because

all of the mathematical operations are done with integers. However, ftree-ter did result in 21% speedup. This flag removes the use of temporary values by replacing it with the original expression. In this case, it reduced the overhead of creating these variables.

Benchmark Name	Compiler Flag	Result	-O0 Result	Speedup
nestloop	O1	128415.275	361565.727	2.8155975
nestloop	O3	183796.839	361565.727	1.9672032
nestloop	O2	184600.62	361565.727	1.9586377
nestloop	ftree-ter	359798.954	361565.727	1.0049104
nestloop	pseudo O1	360927.507	361565.727	1.0017683
nestloop	ftree-parallelize-loops=1	362247.727	361565.727	0.9981173
nestloop	ftoplevel-reorder	362597.587	361565.727	0.9971543
nestloop	fomit-frame-pointer	364274.898	361565.727	0.9925628
nestloop	mieee-fp	371012.755	361565.727	0.9745372

Analysis

The nested loop benchmark performs an incremental operations inside seven nested loops and outputs the final result. While O1 provided 200% speed up over no optimization, O2 and O3 only provided a speedup of 100%. It seems that the additional optimizations in O2 and O3, actually generate slower running code. As we discussed in class, the compiler can actually generate worse code by trying to optimize, but this is no way to know for sure without actual input and running the code, emphasizing the importance of profile guided optimization.

Benchmark Name	Compiler Flag	Result	-O0 Result	Speedup
quickSort	O2	7327.441	13577.306	1.8529397
quickSort	O3	7701.272	13577.306	1.7629953
quickSort	O1	8140.16	13577.306	1.6679409
quickSort	pseudo O1	12287.162	13577.306	1.1049993
quickSort	fomit-frame-pointer	12848.233	13577.306	1.0567450
quickSort	ftree-ter	12980.654	13577.306	1.0459647
quickSort	mieee-fp	13338.197	13577.306	1.0179266
quickSort	fmerge-constants	13633.882	13577.306	0.9958503

quickSort	ftoplevel-reorder	13673.269	13577.306	0.9929817
quickSort	ftree-parallelize-loops=1	13895.849	13577.306	0.9770764

Analysis

Quicksort is a recursive algorithm, with hardly any looping, so ftree-parallelize-loops did not do anything beneficial in this case. It appears that fomit-frame-pointer has freed up a register which the algorithm has taken advantage of. fmerge-constants did not help because there were no duplicate constant values in this algorithm which allowed optimization. There were also no float numbers in this method, so mieee-fp did not help.

Future Work

Our research opened up several questions that we were unable to answer. One of the most perplexing questions is why we couldn't imitate the effects of -O1 using "pseudo O1". By using gcc <src_code.c> -O1 -S -fverbose-asm, we could determine exactly which compilation flags O1 uses. But when compiling with `gcc <src_code.c> -flag1 -flag2 -...` where we listed all the flags manually, the assembly code was wildly different despite the fact that the flags were identical. It appears that O1 performs some additional undocumented optimizations besides the listed flags. It is not clear whether additional unlisted flags are being used or if there is some other optimization going on. Furthermore, one flag (-fleading-underscore), which is listed as a flag in -O1 produced linking errors when we tried to use it ourselves. It is unlikely that this flag is the cause of the discrepancy between -O1 and pseudo-O1.

The scope of the project could also be expanded to provide more complex assembly-level comparisons between different optimizations. Comparing the assembly instructions between files could provide further insight into the different sources of speedup in our results.

References

[1] "The Great Iota Compiler Shootout '01." The Great Iota Compiler Shootout '01. 2001. Cornell University -- Computer Science Department. 07 Dec. 2013 <http://www.cs.cornell.edu/courses/cs412/2001sp/hw/bench/>.

[2] "Optimize Options - Using the GNU Compiler Collection (GCC)." A GNU Manual. 11 Apr. 2013. Free Software Foundation, Inc. 07 Dec. 2013 <http://gcc.gnu.org/onlinedocs/gcc-4.7.3/gcc/Optimize-Options.html>.

This page contains additional work for earning the Master's Pass

What is Parallelism good for?

Parallelism was developed to take advantage of the multiple processing units (processors, cores, ALUs) that most modern computers have. Pretty much every type of device today - computers, tablets, phones, etc. - have multiple processing units. Multiple cores, specifically, became a big deal in the early 2000's when chip designers started bumping up against a physical limit on the clock speed of processors. Rather than making chips faster, they started putting two (and eventually more) chips on a single computer. Without parallelized code, our use of multiple cores would be extremely limited. It would still help with running two separate processes at once, but we would be unable to increase the processing speed of a single expensive process. Parallelism has been widely exploited in particular problems that show up frequently. Graphics rendering is a task that benefits enormously from parallelism. Since the value for a particular pixel can be computed entirely independently from the value of other pixels, multiple cores can seamlessly render an image together. There is also a class of methods that are called *decomposable*, these are methods that break data into smaller chunks. Each smaller check is operated on, and the result from each small chunk are combined for the answer. One example is finding the sum of all the elements in a list, where we could divide the list into two halves, compute the sum for each half and then add the results for each half together. Such problems lend themselves perfectly to parallelization.

But parallelization does not come for free and much research has gone into how to best achieve it. For many years it was believed that good optimization could be achieved at the machine level, which would involve the processor examining compiled code and intelligently dividing up instructions to different cores. This appeared to be a boon for developers since the parallelization would be decided at runtime and none of the burden would fall on the developers. Unfortunately, this turned out to be too optimistic as the low-level machine instructions proved to be too difficult to parallelize. Modern efforts focus on parallelizing at the compiler level. But this, too, has costs. Software that determines what can be parallelized is very computationally intensive, and programs which are compiled with these optimizations take much longer to compile. There is also the risk of errors. Parallelization opens a host of issues such as race conditions. A race condition happens when a single variable is accessed by two separate processes which access, modify and then store the variable, resulting in the variable being overwritten between the processes. To avoid race conditions we must use variable locking using mutual exclusion methods, but these are expensive. In the worst cases, the overhead involved in synchronizing between multiple threads can actually result in slower code than the non-parallelized version.

Vectorization - Benefits and drawbacks

Vectorization is an optimization in which multiple adjacent elements in an array are accessed and processed together. Vectorization is used, for example, when adding the values of two arrays of equal length together. Starting with vectors A and B, A typical for-loop would start by computing A[0]+B[0], then A[1]+B[1], and so on until the end. With Vectorization, the processor might compute A[0]+B[0],A[1]+B[1], A[2]+B[2],A[3]+B[3] all at once in a single operation. This would result in a 4X speedup from the original. Since iterating over arrays in this fashion is so common, vectorization is great in many situations.

The main drawback in vectorization is that there are many scenarios which are easy to create in

which it does it cannot be used. Vectorization breaks down when a result relies on the values of multiple array indices, such as computing the windowed average of values in an array. Accessing the array in a non-linear fashion, such as using an array for heapsort will also render vectorization useless. Fortunately, it is easy to assess when vectorization can and cannot be safely used, so a modern compiler will never erroneously use it and cause an error.