Quantifying Compiler Effects on Code Performance and Reproducibility using FLiT

Poster Summary

Michael Bentley
University of Utah
mbentley@cs.utah.edu

Ganesh Gopalakrishnan (Advisor)
University of Utah
ganesh@cs.utah.edu

Dong H. Ahn (Advisor)
Lawrence Livermore National Laboratory
ahn1@llnl.gov

ACM Reference format:
Michael Bentley, Ganesh Gopalakrishnan (Advisor), and Dong H. Ahn (Advisor). 2017. Quantifying Compiler Effects on Code Performance and Reproducibility using FLiT. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis , Denver, Colorado USA , November 2017 (SC'17),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Experimental reproducibility is part of the foundation of the scientific method. However, compilers sometimes optimize floating-point code so aggressively that it can produce *different answers* depending on how it is compiled.

For any type of code performing floating-point computations, the amount of difference incurred by the changes from the compiler may be small enough that it simply does not matter. But without knowing how the compiler is changing the results of the computations, how can one know if the difference is negligible? Are these aggressive optimizations even necessary to get the code to run fast enough?

FLiT is a reproducibility testing framework that compiles user code under multiple compilers, optimization levels, and flags [1]. It analyzes the performance and difference between each compilation and a baseline compilation. The baseline compilation, used in comparisons, is chosen by the user to be a compilation that can be trusted, and is usually set to the unoptimized version.

There are efforts by the community to address concerns of floating-point reproducibility [2, 3]. Instead of looking at particular examples or optimizations directly, FLiT focuses on the reproducibility of user code under compiler optimizations.

FLiT has been used to analyze litmus tests that demonstrate differences with small and simple floating-point computations. The purpose of this work is to determine the usefulness of the FLiT tool for real-world libraries and applications. In this work, I show that (1) existing tests from a library can be repurposed as FLiT reproducibility tests, (2) performance can be achieved under high reproducibility constraints, and (3) FLiT can identify floating-point algorithms that show high sensativity to compiler-induced variability.

SC'17, November 2017 , Denver, Colorado USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 CASE STUDY: MFEM

MFEM was chosen because it is an influential library for a variety of large-scale simulations at the Lawrence Livermore National Lab (LLNL). MFEM is a finite element library that is typically used to solve discretized versions of partial differential equations.

The MFEM code comes with 17 usage examples. I migrated 12 of the 17 MFEM examples into FLiT tests and performed analysis over three compilers, clang, gcc, and icc (Intel's c++ compiler). The baseline was set to the unoptimized compilation from gcc (`gcc -O0`). The unoptimized compilation was chosen because it should follow the code as it is written.

There were five main findings from this research:

(1) A compiler bug in Clang was discovered in this effort involving linking against a shared library compiled with gcc 5.4.0. This is a failure to reference a static variable at link time. This bug has been reported to the Clang developers and a workaround in FLiT has been implemented.
(2) For four tests, icc got a different answer than the baseline on all of the compilations, even the unoptimized. It was found not to be the Intel compiler, but rather the Intel linker that causes floating-point variability. Exactly how the linker caused variability is still under investigation.
(3) 8 of 12 tests had better performance with safe compilations. Of the other four, only one of them had a performance boost higher than 7% over the fastest safe compilation (integration test 9 had a 28% boost with an unsafe compilation). Here *safe* means it agrees with the baseline result.
This was a surprising result to show that even with the strict requirement of bitwise reproducibility, the fastest executables can still be achieved (up to a 10x speedup from the unoptimized).
(4) For these 12 MFEM integration tests, the fastest compilation for each compiler is identified by fastest average speedup. This can be seen in the Table 1. The compilations for gcc and clang in that table were safe for all tests. The compilation for icc in that table was *unsafe* for all tests.
(5) MFEM integration test 8 had the largest difference of all tests. Its difference is an ℓ_2 norm over all values over the mesh, the largest value of which is of the order of 0.3. The difference is of the order of 10^{-6} despite being an iterative algorithm that stops at 10^{-12}. This was primarily traced to unsafe optimizations on a single function: vectorization of the dot product.
It is surprising that an iterative algorithm like this would be sensitive to compiler-induced variability. For a small example

Table 1: Compilation per compiler with the highest average speedup

Compilation	Avg. Speedup
`clang -O2 -fexcess-precision=standard`	6.29
`gcc -O2 -mfpmath=sse -mtune=native`	6.27
`icc -O2 -fp-model fast=2`	6.08

that runs in less than a second on a single core, one can imagine the potential accumulated error for a large long-running example.

3 FUTURE WORK

Looking at integration test 8, there may be code alterations that can prevent unsafe optimizations or reduce sensativity to these optimizations. This will be explored further.

Also of interest is to automate pinpointing the exact location where variability is caused by the compiler. By linking object files compiled by the baseline and problematic compilation and executing the test again, we can determine exactly which function causes variability. This approach will be automated and improved. Further, there may be machine learning approaches that can narrow down the search even further.

I would also like to explore searching over a larger space of multiple compiler flags, perhaps in a greedy way to avoid the exponential search space growth.

ACKNOWLEDGEMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-735968).

REFERENCES

[1] FLiT: Floating-point Litmus Tests 2017. FLiT: Floating-point Litmus Tests. http://www.cs.utah.edu/fv/FLiT. (2017). GitHub URL: https://pruners.github.io/flit/.

[2] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science)*, Xavier Rival (Ed.), Vol. 9837. Springer, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

[3] Andres Nötzli and Fraser Brown. 2016. LifeJacket: verifying precise floating-point optimizations in LLVM. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*, Charles Zhang and Xavier Rival (Eds.). ACM, 24–29. https://doi.org/10.1145/2931021.2931024