Automatic Generation of Mixed-Precision Programs

Extended Abstract

Logan Moody
Lawrence Livermore National Laboratory
moodylg@dukes.jmu.edu

Nathan Pinnow
Lawrence Livermore National Laboratory
pinnown@wwu.edu

Michael O. Lam
James Madison University
lam2mo@jmu.edu

Harshitha Menon
Lawrence Livermore National Laboratory
gopalakrishn1@llnl.gov

Markus Schordan
Lawrence Livermore National Laboratory
schordan1@llnl.gov

G. Scott Lloyd
Lawrence Livermore National Laboratory
lloyd23@llnl.gov

Tanzima Islam
Western Washington University
tanzima.islam@wwu.edu

ABSTRACT

Floating-point arithmetic is foundational to scientific computing in HPC, and choices about floating-point precision can have a significant effect on the accuracy and speed of HPC codes. Unfortunately, current precision optimization tools require significant user interaction and few work on the scale of HPC codes due to significant analysis overhead. We propose an automatic search and replacement system that finds the maximum speedup using mixed precision given a required level of accuracy. To achieve this, we integrated three existing analysis tools into a system that requires minimal input from the user. If a speedup is found, our system can provide a ready-to-compile mixed-precision version of the original program.

CCS CONCEPTS

• **Software and its engineering** → *Dynamic analysis*; *Software notations and tools*; *Software verification and validation*; *Source code generation*; • **General and reference** → *Performance*;

KEYWORDS

scalable tools, instrumentation, mixed-precision, performance, source code generation

ACM Reference Format:
Logan Moody, Nathan Pinnow, Michael O. Lam, Harshitha Menon, Markus Schordan, G. Scott Lloyd, and Tanzima Islam. 2018. Automatic Generation of Mixed-Precision Programs: Extended Abstract. In *Proceedings of Super Computing (SC'18).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

IEEE floating-point arithmetic is ubiquitous in scientific computing, providing different levels of precision (e.g., single or double precision). Lower precisions can improve speed, reduce memory bandwidth, and lower energy consumption. However, it is often difficult to determine when single precision floating point can be used over double precision, so a common practice is to use double precision exclusively. Although the difference in runtime is marginal on everyday programs, HPC codes can experience a significant speedup.

Unfortunately, current floating-point precision analysis and optimization tools require significant user interaction and most do not scale to HPC workloads. Determining which variables can be converted to single precision while maintaining the required accuracy is difficult or impractical on HPC codes when using a search-based approach because of the enormous search space (the power set of all variables). An additional challenge is determining if a particular mixed-precision configuration actually realizes a speedup because of the added overhead of inserting cast operations.

We propose an automated pipeline that given source code will produce optimized mixed-precision source that maintains a provided level of accuracy. In order to achieve this we use the Rose compiler framework [4] tool TypeForge for source-to-source translation, ADAPT [3] to narrow the search space of configurations, and CRAFT [1, 2] to perform the search and verify the result.

2 CONTRIBUTIONS

The posters pipeline diagram shows an overview of our system. The user provides as input the original source code, a verification routine, and an error threshold. After analysis, our system will produce a mixed-precision version of the source. TypeForge extracts source code information that the other tools need, ADAPT calculates adjoints for each variable and suggests a mixed-precision configuration, and CRAFT uses the output of TypeForge and ADAPT to search for an optimal configuration. CRAFT further uses TypeForge to generate mixed-precision configurations for compilation and testing.

2.1 TypeForge

TypeForge serves as a source code manipulation and search tool for other stages of the pipeline.

In the initial configuration stage, TypeForge generates a list of possible variable replacement candidates for the CRAFT tool while also generating unique handles for every variable to facilitate communication between all stages of the pipeline.

To enable the ADAPT analysis stage, TypeForge converts floating-point variables to an `AD_real` type and inserts calls to the ADAPT library. ADAPT uses operator overloading to perform algorithmic differentiation whenever an `AD_real` variable is assigned to. Previously, ADAPT instrumentation and type replacement had to be added manually by a developer.

During the search stage, CRAFT searches for the optimum configuration based on ADAPT results. CRAFT uses TypeForge to generate and compile new source code with the converted types.

2.2 ADAPT

ADAPT [3] is a tool that analyzes a program to identify critical variables using a technique called automatic differentiation. Automatic differentiation is used to calculate the numerical derivative of a computer program. ADAPT conducts a reverse pass of the AD tree to calculate the sensitivity of the output with respect to all inputs and intermediate variables. With this information ADAPT is able to determine which variables in a program are more likely to need higher precision. CRAFT uses this list of variables along with the calculated error information to reduce search time.

2.3 CRAFT

CRAFT [2] searches a given set of variables to find a mixed-precision configuration that minimizes runtime while passing a user-provided verification routine. Originally, CRAFT worked at the binary machine code instruction level and could not guarantee a speedup because it lacked the ability to incorporate optimizations such as vectorization. For our system, we enhanced CRAFT to conduct a search on source-code variables instead of machine code instructions. CRAFT is now able to find a mixed-precision speedup (if one exists) and provide compilable source code for the final configuration. CRAFT uses TypeForge to perform source code transformations.

We have also added new search strategies to CRAFT in an attempt to speed up configuration searching. The original binary hierarchical search strategy is no longer applicable to source-level searching, and the original *combinational* strategy was an exhaustive search, testing all possible configurations $2^n - 1$ for n variables and taking far too much time for any non-trivial code.

A new *compositional* search replaces every variable individually and then attempts to build better configurations using compositions of already-passing configurations. The compositional search avoids large areas of the search space dominated by variables that cannot be replaced, providing results that are globally optimal using less analysis time.

Finally, a *delta-debugging* search was implemented based on the algorithm used by Precimonious [5]. This strategy uses a binary search approach to search an asymptotically smaller space than either of the other approaches, however it is not guaranteed to find the global optimum.

2.4 JSON Interchange Format

To facilitate communication between the three tools, we designed a JSON-based format for describing tool actions and integrated it into all three tools. For the tools written in C++, we wrote a wrapper class to make reading and writing from the files seamless. This made communicating between the three tolls much simpler and will allow for easy expansion of the pipeline in the future.

3 CONCLUSIONS

The pipeline we have created is capable of generating source code that will improve performance. Results on the GSL benchmark FFT show that our pipeline will generate a 1.2x speedup. Currently this pipeline is set up to run on CPUs and we believe that running mixed precision on GPUs will lead to even larger speedups. The pipeline could be further improved by exploring alternative search strategies for CRAFT in an attempt to reduce analysis time. We also conjecture that the integration of a performance analysis model would allow CRAFT to avoid executing program variants, removing much of the analysis overhead.

ACKNOWLEDGMENTS

This work was performed under auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Release LLNL-POST-756004.

REFERENCES

[1] Michael O. Lam and J. K. Hollingsworth. 2016. Fine-Grained Floating-Point Precision Analysis. *International Journal of High Performance Computing Applications* (jun 2016).

[2] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013. Automatically Adapting Programs for Mixed-Precision Floating-Point Computation. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)*. ACM Press, New York, New York, USA, 369.

[3] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeff Hittinger. (in press) 2018. ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning. In The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18).

[4] Dan Quinlan. 2000. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters 10* (2000), 215–226.

[5] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on (SC'13)*. ACM Press, New York, New York, USA, 1–12.