Summer Report: Tool Integration for Variable-Precision Computing

Michael O. Lam

August 9, 2018
LLNL-TR-756348

Summary

This summer I worked in the tools group of the variable-precision computing effort at Lawrence Livermore National Laboratory (LLNL). I coordinated an integration project that combined three existing tools into a single pipeline for automated mixed-precision tuning at the source code level. This involved coordinating efforts between several different project groups and making improvements to all three tools. Two of the tools are already available publicly as open-source projects, and the third has been submitted for release. This document serves as an overview of the summer's efforts, focusing on our contributions as well as ideas for future work.

Contents

1 Contributions

1.1 Overview

My primary effort this summer was the integration of three projects into a single pipelined system for automated mixed-precision tuning. This effort was anticipated in the future work section of my Summer 2017 report [1], and has been expanded to incorporate three major pre-existing software analysis components:

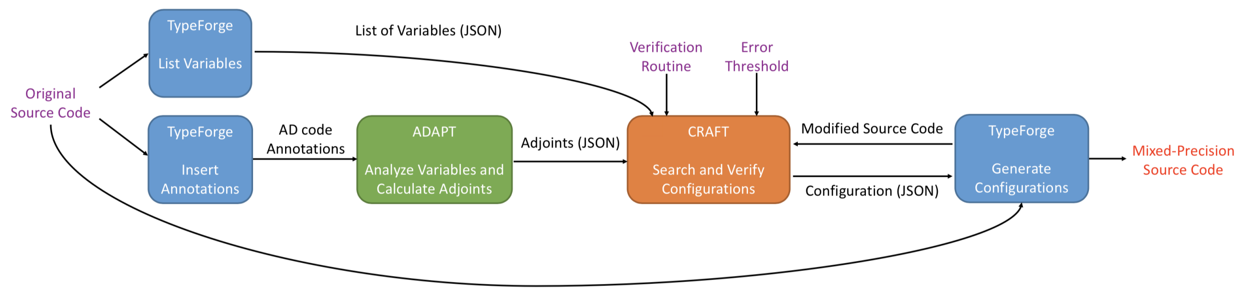

Figure 1: Integrated system pipeline overview

1. Configurable Runtime Analysis for Floating-point Tuning (CRAFT) [4, 3, 2] is a general framework for floating-point program analysis with support for a variety of different kinds of analysis. The part most useful for this summer's project is the automated mixed-precision search system. This system is implemented as a Ruby script and was originally designed to find double-precision machine code instructions that could be replaced by their single-precision equivalents. For this project, we leveraged the existing search framework to implement source-level tuning.
URL: `github.com/crafthpc/craft`

2. Algorithmic Differentiation Applied to Precision Tuning (ADAPT) [to appear, SC'18] is a wrapper for the CoDiPack library for algorithmic differentiation [7] that adds floating-point precision tuning analysis. For this project, we use ADAPT to narrow the CRAFT search space and focus on variables that can be replaced according to the algorithmic differentiation results.

3. TypeForge is a tool written using the Rose compiler framework [5] that performs type substitutions and other operations on source code. It can convert the types of specific variables or of all variables in a given scope. For this project, we used the type conversion routines to convert variables to lower precision, and we used the general instrumentation framework to insert ADAPT calls.
URL: `github.com/rose/rose-develop`

Figure 1 shows an overview of the integrated pipeline. We use TypeForge to extract a list of tunable variables and optionally to insert ADAPT instrumentation. If ADAPT is used, the user must also insert some pragma-based annotations to describe the program outputs and allowed error thresholds. The ADAPT-instrumented program runs and produces a mixed-precision recommendation that is guaranteed not to exceed the user-provided error threshold; however, it might not speed up the program.

Finally, CRAFT performs a variable configuration space search (optionally starting from the ADAPT results) and attempts to find a mixed-precision configuration that both passes a user-defined verification routine and achieves a speedup. If such a configuration is found, CRAFT reports the the fastest one and provides the modified source code. Each configuration is built using TypeForge for type conversion, and the search process can parallelize naturally if multiple cores are available.

1.2 Data Interchange Format

To facilitate data interchange between the three tools, we designed a new JSON-based format with support for all of the kinds of data required by the three tools. We then added support for this format to all three tools. Because CRAFT is written in Ruby (which has a built-in JSON module), minimal changes were necessary to support the format in CRAFT. For ADAPT and TypeForge, we used the "JSON for Modern C++" header library (`https://github.com/nlohmann/json`). Figure 2 shows the schema for this format.

1.3 CRAFT Improvements

The primary change required for CRAFT was to add the notion of a program variable as a first-class object for tuning. We implemented this by adding a new "variable mode" activated using a command-line switch

```
{
  "type": "object",
  "properties": {
    "source_files": {
      "type": "array",
      "items": {
        "type": "string",
        "pattern": ".*.cpp|.*.c|.*.C|.*.h|.*.hpp|.*.H"
      }
    },
    "tool_id": {
      "type": "string",
      "pattern": "TypeForge|CRAFT|ADAPT|Master|None"
    },
    "executable": { "type": "string" },
    "version": { "type": "string" },
    "actions": {
      "type": "array",
      "title": "actions",
      "items": {
        "type": "object",
        "title": "action",
        "properties": {
          "action": {
            "type": "string",
            "pattern": "(replace|change)_(type|vartype|varbasetype)|transform|
                         list_basereplacements|introduce_include|replace_pragma"
          },
          "name": { "type": "string" },
          "handle": { "type": "string" },
          "scope": { "type": "string" },
          "source_info": { "type": "string" },
          "from_type": { "type": "string" },
          "to_type": { "type": "string" },
          "error": { "type": "number" },
          "dynamic_assignments": { "type": "number" }
        },
        "additionalProperties": false,
        "required": [ "action" ]
      }
    }
  },
  "additionalProperties": false,
  "required": [
    "version",
    "tool_id"
  ]
}
```

Figure 2: JSON interchange format schema

("`-V`"). As a prototype, the mode initially used an internal representation where it treated variables as machine code "instructions," which was the base level at which CRAFT originally performed tuning. After verifying that the general search approach would still work, we added first-class support for variables.

One benefit of doing the search at the variable level with a source-to-source tool is that speedups can be verified empirically by compiling and running a mixed-precision configuration. Previously, the instrumentation overhead involved in building the configuration far outweighed any benefit of the precision replacement. In addition, the compiler did not have a chance to perform optimizations like vectorization that would have exposed performance improvements at lower precision. With source-level transformations neither of these problems manifest, and for the first time CRAFT can directly search for a speedup.

Thus, we pursued two metrics for measuring the quality of a configuration: 1) the total number of variables replaced (which is a metric that was used previously for instructions) and 2) the actual speedup achieved by the configuration.

A secondary change to CRAFT was the addition of several new search strategies. Because variables do not have as deep of a structural hierarchy as instructions ("function $\rightarrow$ variable" rather than "module $\rightarrow$ function $\rightarrow$ basic block $\rightarrow$ instruction"), the old hierarchical strategy was less useful. Over the course of the summer, we added (or updated) three new search strategies:

1. **Combinational** - This is a brute-force strategy that simply tries all potential combinations of variables. This strategy is not viable for more than a few variables because it will test $2^n - 1$ configurations given n variables (it does not need to try the configuration where zero variables are replaced). However, it is useful for establishing a baseline for comparison and for finding a global maximum replacement count and speedup for small numbers of variables.

2. **Compositional** - This strategy tries replacing every variable individually and then attempts to build better configurations using compositions of already-passing configurations. It does this by taking every passing configuration with k replacements and building new configurations by taking the union with all k-cardinality and 1-cardinality configurations. This approach will generally find the global maximum replacement count and speedup, but it is not guaranteed to do so. However, it also does not necessarily try all possible combinations. In practice, it avoids large areas of the search space dominated by variables that cannot be replaced and provides results that are usually globally optimal.

3. **Delta debugging** - This strategy is based on the algorithm described in the original Precimonious paper [6]. It uses a binary search on the list of program variables and examines an asymptotically smaller space than either of the other approaches. However, it is also not guaranteed to find global maxima for either replacement count or speedup.

Finally, we are extending CRAFT with a "wizard" mode that provides an interactive interface for running the entire system and the various pieces of the pipeline. This mode asks the user several questions and walks them through creating the various scripts necessary for the rest of the system. It then runs the various pieces of the system, describing them as they run. The actual run scripts are saved so that the user can re-run various stages over again if they wish (or if there is a problem that they need to fix). Figure 3 shows an edited excerpt from a session with this wizard.

1.4 TypeForge Improvements

Several improvements were added to TypeForge:

1. **Base type replacement** - Previously, TypeForge was only able to replace type literals (e.g., "`double*`" was changed to "`float`" rather than "`float*`". This is a valid use case for TypeForge but would have required a very tedious listing of types in the context of this project. Thus, a "replace base type" option was added to TypeForge that allows us to specify that only the floating-point part of a type declaration should be changed, not any pointer or array components.

2. **Pragma instrumentation** - A new mechanism was added for inserting instrumentation based on pragmas. This allows ADAPT calls to be inserted at user-specified locations in the original program. This is useful in a few situations where it is difficult or impossible to infer information automatically (e.g., name and error tolerance of outputs).

```
== CRAFT Wizard ==

To search in parallel automatically, the system must be able to:
  1) acquire a copy of your code,
  2) build your code using generic CC/CXX variables,
  3) run your program using representative inputs, and
  4) verify that the output is acceptable.

How would you like to acquire a copy of your code?
  a) Recursive copy from a local folder
  b) Clone a git repository
Choose an option above: a
Enter project root path:  [default='.']
Acquisition script created: /g/g19/lam26/src/adtests/sum2pi_x/simple/.craft/wizard_acquire

How is your project built?
  a) "make"
  b) "./configure && make"
  c) "cmake ."
  d) Custom script
Choose an option above: a
Build script created: /g/g19/lam26/src/adtests/sum2pi_x/simple/.craft/wizard_build

Enter command(s) to run your program with representative input.
If you need to save the output for verification purposes, please
write it to "stdout" in the current folder. Enter an empty line
to finish.

./sum2pi_x >stdout

Run script created: /g/g19/lam26/src/adtests/sum2pi_x/simple/.craft/wizard_run

How should the output be verified?
  a) Exact match with original
  b) Contains a line matching a regex
  c) Contains no lines matching a regex
  d) Custom script
Choose an option above: a
Verify script created: /g/g19/lam26/src/adtests/sum2pi_x/simple/.craft/wizard_verify

Finding variables to be tuned.
typeforge --spec-file initial.json --compile -g -O3 -Wall -Wno-unknown-pragmas -c sum2pi_x.cpp
typeforge --spec-file initial.json --compile -g -O3 -Wall -Wno-unknown-pragmas -c main.cpp
typeforge --spec-file initial.json --compile -o sum2pi_x sum2pi_x.o main.o -lm
Initial configuration created: /g/g19/lam26/src/adtests/sum2pi_x/simple/.craft/craft_initial.json

If you wish, now we can run your program with ADAPT
instrumentation. This ill most likely cause the search to
converge faster, but your program must be compilable using
'--std=C++11' and you must have included all of the appropriate
pragmas (see documentation).
Do you wish to run ADAPT? [default='n']

CRAFT supports several search strategies:
  a) Combinational - try all combinations (very expensive!)
  b) Compositional - try individuals then try to compose passing configurations
  c) Delta debugging - binary search on the list of variables
Which strategy do you wish to use for the search? c
How many worker threads do you want to use? [default=72] 32

CRAFT 1.1
Initializing standard search:  strategy=ddebug
Performing baseline performance test ... Done.  [Base error: 0.0  walltime: 0:00:00]
Initializing search strategy ... Done.  [7 candidates]
[...]
Candidate queue exhausted.  [Max queue length: ~2 item(s)]

          Top instrumented (passed):               Runtime (s)          Speedup (X)
            - acc_answer_diff_error                0.01                 1.1

          Speedup achieved! (max: 1.1x, baseline: 0.01s)

Total candidates:                   7
Total configs tested:               3
  Total passed:                     1
  Total failed:                     2
  Total aborted:                    0
Done.  [Total elapsed walltime: 0:02:02]
```

Figure 3: Excerpt from CRAFT tuning “wizard”

	SUM2PI	**DFT**	**CG**	**FFT**	**MG**
Candidate variables	4	10	16	23	59
Configurations tested					
Combinational	15	1023	65535	8.39e6	5.76e17
Compositional	8	638	918	-	8000+
Delta debugging	3	11	16	11	230
Highest number of replacements					
Combinational	3	9	-	-	-
Compositional	3	9	7	-	-
Delta debugging	3	9	4	-	4
Best speedup					
Combinational	1.0	1.0	-	-	-
Compositional	1.0	1.0	0.9	-	-
Delta debugging	1.0	1.0	0.9	1.2	1.0

Table 1: Comparison of search strategies

	w/o ADAPT	**w/ ADAPT**
Combinational	127	7
Delta debugging	22	11

Table 2: Impact of ADAPT results on SUM2PI search

TypeForge also had to be modified to support the new JSON format for both input and output. We designed the format with significant input from the TypeForge developers and they will be adopting it as their primary input format.

1.5 ADAPT Improvements

ADAPT required relatively few changes for this project. The most significant was a major cleanup in preparation for public release (pending review). A significant amount of the original source code was removed, consisting of experimental features and unused code from last year's work. This included the entire forward analysis mode as we found it to be not useful for this project. The only significant new features added for this project were support for the new JSON output format and support for multiple dependent (output) variables.

1.6 Preliminary Results

As of the end of summer, the minimal changes to all three components in order to cooperate have been completed, and we have successfully applied the whole system to several examples and benchmarks.

Table 1 shows some preliminary results comparing the three search strategies described above. Clearly, the combinational approach does not scale, and the compositional approach has a similar problem (although to a lesser extent). Delta debugging is much more efficient, but does not always find all possible replacements.

Table 2 shows some preliminary results demonstrating the impact of ADAPT analysis on the search phase. If ADAPT info is present, the combinational search strategy will use it to narrow the field of valid candidates for replacement and consider only those that ADAPT recommends replacing. If ADAPT info is present, the delta debugging search strategy will sort the variables by ADAPT-reported error, improving the search convergence by grouping low-error variables together. The compositional strategy does not currently incorporate ADAPT information.

2 Future Work

2.1 User Interface Improvements

The new "wizard" mode is a step towards making the entire tuning process seamless, and we have attempted to make sure all of the tools are well-integrated and documented. However, there is always room to make the tool more accessible to application developers, including a more robust wizard and the addition of a GUI (where usable).

2.2 Scaling

We have already shown that the various components of the pipeline scale to HPC benchmarks and small proxy apps. Future work includes pushing the system to work on larger and more complex programs. Here are a variety of obstacles to scaling, including build system complexity, the number of technologies and libraries used, parallel and distributed communication, and of course the amount of computation and the size of data involved. Each of these presents challenges that still need to be addressed.

2.3 Search Strategy Improvements

Currently, only one search strategy (delta debugging) terminates in a reasonable amount of time on programs with a non-trivial number of variables (e.g., 20+). Further developments in search strategies are needed. Here are a few ideas:

- **Invalid configuration identification** - We have observed that often there are "groups" of variables that must be converted atomically (i.e., you must reduce the precision for all of them or none of them). A smart search strategy would attempt to identify these groups early on in the search process so as to save time later by tuning them as a group.
- **Performance estimation** - Currently, the search strategy has no reliable way to determine which configuration(s) will have the best performance. This is determined by trial-and-error. A performance model that could accurately predict the performance of a configuration without actually building and running it would make the search converge much quicker.

2.4 Extension to GPUs

Speedups due to precision reduction are more significant on accelerated platforms. In our ADAPT work, for example, we found a 1.2 speedup on LULESH only with the GPU version. This is because single-precision arithmetic is often not significantly faster on CPUs; any speedup must be achieved by vectorization and reduced memory pressue. Theoretically our system can support tuning GPU code directly, as Rose has recently been extended with CUDA support. Unfortunately, due to time constraints this has been relegated to future work.

2.5 Other Tools

By designing a general JSON standard for data interchange and by implementing as much of the system as possible within the three main components, we hope to be able to easily add more components in the future. This could include components for cancellation and dynamic range detection, fault tolerance analysis, or alternative representation prototyping.

2.6 HPC Benchmark Suite

Now that there are several major projects attempting to perform precision analysis at HPC scales, finding suitable benchmarks with well-defined accuracy requirements can be a challenge. I have initiated an effort to build a shared HPC floating-point benchmark suite, and nearly all major groups have responded indicating their interest in and willingness to contribute to such an effort. I have offered to help coordinate this effort, which will begin during the Fall 2018 semester.

3 Conclusion

This summer, I helped coordinate a new integration effort combining three program analysis components (CRAFT, ADAPT, and TypeForge) into an end-to-end precision tuning system. We have extended all three components to enable the integration, created a new data interchange format to facilitate communication, and tested the system on several examples and bechmarks. We demonstrated that such analysis is feasible and applicable to small-scale HPC workloads. We have also identified several ideas for extension projects, and anticipate that the system will serve as a foundation for many avenues of future work.

Acknowledgements

Given its nature as an integration project, the scope of my work this summer was large and required the cooperation and assistance of many people besides myself.

I would like to acknowledge Scott Lloyd, my LLNL summer scholar technical mentor, who provided invaluable guidance for the project. I would also like to extend a huge thank-you to my student Logan Moody, who became familiar with the various pieces of the project in an astonishingly short period of time and made many contributions over the summer (especially in the design and implementation of the JSON-based data interchange format). I thank Harshitha Menon, who originally proposed the use of algorithmic differentiation for mixed-precision analysis; we are co-developers on ADAPT. I also thank Markus Schordan, the original developer of TypeForge who provided significant consulting and advice for the project, as well as Nathan Pinnow, his summer scholar student who extended TypeForge for this project.

Finally, I gratefully thank Jeff Hittinger for funding this project. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, via LDRD project 17-SI-004. LLNL-TR-756348

References

[1] Lam, M. O. Summer Report: Software Tools for Variable-Precision Computing. Tech. rep., Lawrence Livermore National Laboratory, 2017.

[2] Lam, M. O., and Hollingsworth, J. K. Fine-Grained Floating-Point Precision Analysis. *International Journal of High Performance Computing Applications* (6 2016), 1094342016652462.

[3] Lam, M. O., Hollingsworth, J. K., de Supinski, B. R., and Legendre, M. P. Automatically Adapting Programs for Mixed-Precision Floating-Point Computation. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)* (New York, New York, USA, 6 2013), ACM Press, p. 369.

[4] Lam, M. O., Hollingsworth, J. K., and Stewart, G. Dynamic Floating-Point Cancellation Detection. In *WHIST '11* (2011).

[5] Quinlan, D. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters 10*, 02n03 (2000), 215–226.

[6] Rubio-González, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on (SC'13)* (New York, New York, USA, 11 2013), ACM Press, pp. 1–12.

[7] Sagebaum, M. A. X., Albring, T. I. M., and Gauger, N. R. High-Performance Derivative Computations using CoDiPack. Tech. rep., 2012.