Summer Report: Software Tools for Variable-Precision Computing

Michael O. Lam

August 11, 2017
LLNL-TR-736999

Summary

This summer I worked in the tools group of the variable-precision computing effort at LLNL. I provided support for existing tools and built a new tool based on the application of automatic differentiation to floating-point precision analysis. The tool itself along with documentation and examples are provided as artifacts in a software repository [1]. 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 Tools Assessment

1.1.1 General Floating-Point Analysis Tools

At the beginning of the summer, we spent some time assessing existing tools for floating-point precision analysis. This included an updated literature review of the area, and I added several new projects and papers to the comprehensive list of floating-point analysis research that I maintain on my website [2].

Unfortunately, we found that the tools in this area are very brittle and difficult to use. I was able to build and use CRAFT [9] and SHVAL [10] (my own tools) on several DOE applications throughout the summer,

[1] `ssh://git@cz-bitbucket.llnl.gov:7999/vpc/adtests.git`
[2] `https://w3.cs.jmu.edu/lam2mo/fpanalysis.html`

but none of the other tools that we looked at (Precimonious [11], FPTuner [2], Rosa [4], etc.) proved usable on DOE codes on LC machines. This is due to a variety of factors including unresolved runtime errors (e.g., Precimonious), commercial software dependencies (e.g., FPTuner), and non-standard languages (e.g., Rosa).

To help remedy this situation, I added additional documentation and a one-click install script for CRAFT. I also submitted a BoF proposal for the Supercomputing conference this year, where I hope to bring several tool developers together to talk about increasing usability and making these tools relevant to a broader audience of users in HPC.

1.1.2 Automatic Differentiation Tools

We also spent some time assessing various tools for *automatic differentiation* (AD) [6], a technique that computes the derivative of a computer program as if it were a numeric function using the chain rule. We wished to use the partial derivatives produced by AD as a means to calculate the required precision of program inputs and intermediate calculations. Previous work [5] showed that this method worked for small bit-width examples in the chip design domain, and we wished to extend this to floating-point precision in the HPC domain. Because there are many existing robust AD libraries and tools, we decided to find and to leverage one (or more) of them for our analysis rather than start from scratch.

Initially, we tried many AD tools and frameworks, but eventually narrowed our focus to three of the most promising: 1) TAPENADE [7], a proven source-to-source tool, 2) Adept [8], a very easy-to-use expression template library, and 3) CoDiPack [1], a currently-maintained expression template library built with HPC in mind. However, we have included in the repository our notes on all of the libraries and tools that we tested.

Although TAPENADE had good performance characteristics because it is a source-to-source tool, we found that it was difficult to use for our purposes because it was rather inflexible with respect to the form of the input code. Additionally, it was difficult and tedious to extract the adjoints for intermediate results given the opaque nature of its auto-generated code. Finally, it does not support C++.

Adept and CoDiPack were easier to use for our purposes because they provide hooks that allow us to retrieve intermediate values, and because they require minimal modifications to the original source code thanks to the template and operator overloading features in C++. We found that Adept, while easy to use, was significantly slower and had a higher memory overhead, which prevented us from running on larger examples. CoDiPack was much more efficient (comparable to TAPENADE in the forward mode and actually faster in the reverse mode), even though it was a bit more difficult to use. Thankfully, the developer was very responsive and answered several questions throughout the summer with thorough and helpful responses.

Finally, there are two modes for AD: forward and reverse. Forward analysis calculates partial derivatives for a single input variable, and reverse analysis calculates partial derivatives for a single output variable. Reverse mode calculations are generally more accurate [6], but require recording all calculations on a "tape" for the backwards chain rule calculations. CoDiPack supports both modes, and it compares very favorably with TAPENADE in terms of performance, which is impressive considering that TAPENADE is a source-to-source tool and CoDiPack is implemented using expression templates.

1.2 CoDiPack Wrapper

We wrote a wrapper for CoDiPack that both encapsulates the precision analysis logic and makes it much easier to insert AD instrumentation into existing programs. This wrapper consists of a set of additional header files (CoDiPack is a header library) and a couple of additional helper utilities. Together, they provide a simple and unified interface that abstracts away the CoDiPack internals and allows the user to compile and run both the forward and reverse modes without modifying the code in between (which is not possible with the original version of CoDiPack).

In the reverse mode, CoDiPack is used to record all computation and then the tape is evaluated multiple times at the end, once for each output variable. In the forward mode, the program itself is run once for each input variable and the results are aggregated using a post-processing utility. Because of the multiple executions, forward mode incurs a higher runtime cost. However, it does not require recording all computation (as does the reverse mode), which is prohibitively expensive for reasonably-sized HPC benchmarks or proxy apps.

CoDiPack allows you to choose the data type for gradient calculations. In order to ensure that these calculations are as accurate as possible, we currently use the "long double" data type, which is 80 bits wide and provides a higher precision than the standard IEEE double precision format. We found in our experiments that this does not substantially increase the analysis overhead but provides a noticeable improvement in accuracy.

Variable	Bits Required	Min Exponent	Max Exponent
"acc"	23	1	2
"sum"	31	2	10
"tmp"	22	-22	1
"pi"	31	2	2

Table 1: Precision level recommendations for `sum2pi_x` example

```
FUNC: 0 "sum2pi_x"                              [3 instruction(s)]
    BBLK: 0
        INSN: 0 "acc [sum2pi_x.cpp:61]  Exp: (1,2)"
        INSN: 0 "sum [sum2pi_x.cpp:65]  Exp: (2,10)"
        INSN: 0 "tmp [sum2pi_x.cpp:58]  Exp: (-22,1)"
FUNC: 0 "main"                                  [1 instruction(s)]
    BBLK: 0
        INSN: 0 "pi [sum2pi_x.cpp:76]  Exp: (2,2)"
```

Figure 1: Visual results for `sum2pi_x` example

The wrapper also tracks dynamic range information for each variable of interest, reporting the lowest and highest value at the end of execution. This is useful for determining whether the variable's dynamic range is compatible with conversion to a lower precision. We found that adding range tracking did not introduce a significant additional overhead on top of the existing analysis.

All of the information tracked by the wrapper is saved to human-readable text files at the end of execution, and it also creates an output file in the format used by the graphical viewer from CRAFT.

1.3 Experiments

We ran our wrapper-based analysis on a variety of examples. We started with a few sanity-check examples that were small enough for us to work out the correct answers by hand. We also tested on the `sum2pi_x` example from CRAFT, the `arclength` example from Precimonious, and the `DFT` example from the bitwidth paper [5]. We performed scaling analysis using a simple gaussian elimination kernel as well as the LULESH DOE proxy application. We also tested on several other examples of interest, such as Euler2d, the Sequoia and Mantevo benchmarks, and FPBench [3].

In all cases, we found the wrapper very easy to use, and it rarely took more than an hour to do the initial program edits to make it compatible with the AD analysis. In some cases (e.g., `sum2pi_x` and `arclength`), we were able to replicate or confirm previous reported mixed-precision results. In other cases, further investigation will be necessary to find mixed-precision configurations. This is often because a program uses many variables and intermediates, and it is difficult to identify which variables should be the focus of our analysis. We could avoid this difficulty by simply analyzing all intermediates, but this poses a severe scaling challenge as discussed in Section 1.4.

As an example of our analysis, we present our results on the `sum2pi_x` case study included in CRAFT. Table 1 and Figure 1 show the precision recommendations and exponent ranges for the four major variables in this program, and Figure 2 shows the source code with colored dots indicating the precision requirements of each annotated variable (here we show a simplified mode which depicts only green for "below single precision" and red for "above single precision"). Both of these images were produced by the CRAFT viewer (which was designed for binary analysis), and so the references to basic blocks ("`BBLK`") and instructions ("`INSN`") should be ignored in this context.

As shown in the figures, the `sum` variable requires higher precision than the `acc` or `tmp` variables (the `pi` "variable" is actually a constant). This supports the results of the CRAFT analysis, and a manual conversion confirmed that a mixed-precision version of the example achieved the desired accuracy while keeping `acc` and `tmp` in single precision.

Figure 2 also demonstrates the function calls provided by the AD wrapper (beginning and ending the analysis, and registering independent/intermediate/dependent variables). Not shown in this excerpt is the use

```
43  real pi = PI;
44  real tmp, acc;
45  sum_type sum = 0.0;
46
47  real answer = (real)OUTER * PI;  /* correct answer */
48
49  void sum2pi_x()
50  {
51      int i, j;
52      for (i=0; i<OUTER; i++) {
53          acc = 0.0;
54          for (j=1; j<INNER; j++) {
55
56              /* accumulatively calculate pi */
57              tmp = (real)pi / pow(2.0,j);
58              AD_INTERMEDIATE(tmp, "tmp");
59
60              acc += tmp;
61              AD_INTERMEDIATE(acc, "acc");
62          }
63
64          sum += acc;
65          AD_INTERMEDIATE(sum, "sum");
66      }
67  }
68
69  int main()
70  {
71      printf("===  Sum2PI_X  ===\n");
72      printf("sizeof(real)=%lu\n",     sizeof(real));
73      printf("sizeof(sum_type)=%lu\n", sizeof(sum_type));
74
75      AD_begin();
76      AD_INDEPENDENT(pi, "pi");
77
78      sum2pi_x();
79
80      AD_DEPENDENT(sum, "sum_final", 6);
81      AD_end();
82
83      double err = ABS(AD_value(answer)-AD_value(sum));
84
85      printf("  RESULT:  %.16e\n", AD_value(sum));
86      printf("  CORRECT: %.16e\n", AD_value(answer));
87      printf("  ERROR:   %.16e\n", AD_value(err));
88      printf("  THRESH:  %.16e\n", EPS*AD_value(answer));
89
90      if (AD_value(err) < (double)EPS*AD_value(answer)) {
91          printf("SUM2PI_X - SUCCESSFUL!\n");
92      } else {
93          printf("SUM2PI_X - FAILED!!!\n");
94      }
95
96      AD_report(true);
97
98      return 0;
99  }
```

Figure 2: Source code view for `sum2pi_x` example

	Original		CoDiPack (1-var reverse)				CoDiPack (1-var forward)			
Size	Time (s)	MRS (kb)	Time (s)	X	MRS (kb)	X	Time (s)	X	MRS (kb)	X
10	0.2	2084	6.1	29	4030672	1934	1.1	5.2	2896	1.4
15	1.4	3812	29.5	21	15988152	4194	7.2	5.1	6486	1.7
20	5.2	7396	94.4	18	45881108	6204	25.6	4.9	13208	1.8
25	15.8	13164	235.5	15	99337396	7546	66.0	4.2	24192	1.8
30	27.5	21708	(aborted after using >119G memory)				138.5	5.0	40472	1.9

Table 2: LULESH scaling results

of the provided `AD_real` type for all real-valued variables of interest, which is accomplished using a `typedef` at the top of the file.

1.4 Scaling Analysis

Performance overhead is still a significant challenge for this kind of analysis. There are two sources of scaling issues: 1) the size of the computation record or "tape" (in reverse mode only), and 2) the number of independent variables (in both modes).

The first issue is that the reverse mode of AD must record all computation in order to back-propagate gradients from the dependent variables. CoDiPack does apply some fairly aggressive optimization to compress this tape, but it still renders the analysis infeasible for larger applications. Table 2 shows scaling results for LULESH, analyzing a single independent and a single dependent variable (in both cases, the origin energy) and reporting the runtime in seconds and the maximum resident set size in kilobytes. It is clear that the standard reverse mode is not feasible for analyzing this application at full scale.

The second issue is that the runtime and memory overhead of both modes increases with the number of independent variables (which for our analysis could include all intermediate results). We have several ideas and at least two prototypes for reducing this overhead, which are discussed in Section 2.1.

Another related issue that we encountered is that as the number of variables increases, the bit precision requirements increase. This is because the precision requirement calculations are based on dividing the dependent variable tolerable error by the number of independent variables (which in our case includes all intermediates). These high recommendations are not incorrect, but they are overly pessimistic. Over the course of the summer, we developed a few heuristics to modify the suggestions. All of these heuristics can be toggled on and off independently, and none are enabled by default.

The first heuristic is to examine each independent variable value to see if it contains less information in the lower-order bits than what was calculated by the bit precision analysis. We do this by finding the number of trailing zeroes in the significand and calculating the number of bits of non-zero data, then comparing it to the bit precision requirement and reporting the lower of the two numbers. This does not preserve independence between intermediate variables, but it was helpful in some cases such as replicating the Precimonious results on the "arclength" example. In that case, a floating-point number was only being used to store integer powers of two, and thus required only the implicit "1" in floating-point because all the necessary information is encoded in the exponent.

The second heuristic is to compare the calculated tolerated error of each independent variable with its value. If the tolerated error is higher than the value, this tends to indicate that the value is not very important (i.e., has little influence on the dependent variables) and that only the magnitude needs to be stored. The bit precision analysis reports a high precision for these cases because if the error really was near the tolerance we would need that many bits in order to determine if it really was under the threshold. However, this is not usually the case in practice.

The third heuristic is to simplify the reporting by only reporting "single" or "double" for each variable. This is done by comparing the bit precision requirement to see if it is under 24 bits and also checking the variable's dynamic range to ensure that its exponent will fit into the range of single precision (-126 to 127). This simplifies the CRAFT viewer output but hides some of the details.

2 Future Work

2.1 Scaling

We have several ideas to address the scaling issues, and initial prototypes for some of them.

2.1.1 Mode-specific

The forward mode is trivially parallelizable because each run is independent; this allows us to tackle the scaling problem using parallel or distributed computing. We developed a prototype of this using GNU Parallel [12] to run multiple passes independently using a worker pool execution model. This could be extended to a distributed system by submitting individual runs (or groups of runs) as batch jobs.

The forward mode also produces a lot of intermediate output that must be post-processed. Currently this output is stored in human-readable plain-text format, but this is inefficient. We could instead compress the data using binary output and/or compression schemes (including floating-point-specific compression methods such as ZFP). It may also be possible to parallelize the post-processing step, but this has not yet been necessary because it already runs quickly compared with the regular forward passes.

It may be possible to extract some intermediate adjoints from other forward mode passes, without having to run forward passes for those intermediates. We have shown that this can be done for very simple cases by creating a system of equations involving these derivatives and solving for the unknowns. We are not currently aware of a method that works in the general case.

For the reverse mode, the challenges of computation tape length could be addressed using checkpointing and/or paging, but in some sense this just defers the problem to a later point in time because the entire tape must be processed after execution to back-propagate the gradients. It is possible that this could be handled more efficiently using an online streaming algorithm such that the entire tape does not need to be loaded at once. We are not aware of any implementation of AD that uses this approach.

2.1.2 Mode-independent

We also have some ideas that would work in both the forward and reverse modes. The first idea is to take advantage of the composable nature of AD by handling individual functions or regions of code as "black boxes," avoiding the overhead of tracking all computation during those regions. This would provide benefit in both modes but is especially attractive in the reverse mode because it reduces the size of the tape. CoDiPack does provide support for this (although the reverse mode interface is a bit tedious to use), and we developed a simple prototype on a single example program as a proof-of-concept. We observed significant performance improvements in both the reverse mode (5x speedup and 99% memory reduction) and in the forward mode (18x speedup). We believe that this is the most promising avenue for scaling this approach to full HPC applications.

However, applying function composition in the general case will be challenging because it requires 1) identifying candidate functions or regions and their inputs and outputs, 2) automatically adjusting the data types or creating alternative versions to de-activate AD analysis inside those regions, and then 3) manually adjusting the gradients. The last step is significantly different between the forward and reverse mode, because in the former we can just perform the gradient calculation after the original calculation, but in the latter we need to store enough information on the tape so that the back-propagation system can calculate the adjoints later.

Implementing composition automatically may require the assistance of a source-to-source or symbolic differentiation tool to detect candidate functions, generate gradient calculations, and perform data type adjustments. It is important to be careful when selecting candidate functions or regions; even in our prototype we observed that composition is only a benefit for functions that have a derivative that is cheap to compute. Integrating these tools with our online analysis could be challenging, and doing it automatically will be even more difficult.

In addition, we hope to improve scaling in both modes by reducing the number of variables that must be analyzed individually. For instance, it is possible that we could analyze a single representative element in a matrix instead of every element. However, this would require some rigorous way to identify these representative values. We plan to investigate the use of various dimensionality-reduction approaches, such as principle component analysis, eigenvalue decomposition, and importance sampling. We may also be able to use other floating-point analysis tools such as CRAFT to detect variables of interest.

2.2 Integration

Currently, our analysis is only semi-automatic; some source code changes must be done manually. We conjecture that these changes could be done automatically by a source-to-source tool, possibly in conjunction with binary instrumentation to insert AD library function calls at appropriate locations.

We also plan to investigate the integration of our AD analysis with other mixed-precision analysis tools such as CRAFT, Precimonious, and FPTuner. In particular, we hope to be able to use heuristic search tools like CRAFT or Precimonious to narrow the focus of our AD analysis and reduce the number of variables that need to be analyzed. We also conjecture that static dataflow analysis could help by providing static accuracy guarantees for some portions of the program, leaving AD to handle only the regions that the static analysis cannot handle (or where it is overly pessimistic).

Finally, we have not yet investigated the application of this technique to distributed programs. CoDiPack does claim to support the AdjointMPI interface for distributed AD, but we have not yet tested it. This is partially due to time constraints, but also due to the scaling issues; we see little value in pursuing distributed analysis until we can scale robustly on serial code.

2.3 Miscellaneous

We have identified many additional minor improvements and extensions to this work. We plan to examine the impact of our various heuristics across a range of examples, and we also plan to investigate the impact of compiler optimization flags on our analysis. There are also a variety of floating-point edge cases that we have not explicitly addressed yet, such as underflow, overflow, and denormalized numbers. We also do not currently check for floating-point exceptions.

Finally, we observe that the tolerated error in dependents is currently divided evenly among all independents for the purpose of analysis. This matches the approach taken by previous work [5], but that work also proposes a weighted error distribution model. Unfortunately, the specifics of that model were not described in enough detail to replicate. We conjecture that once we are able to identify representative variables from larger data structures (see Section 2.1), we could weigh each variable's contribution based on the size of its overall data structure. We might also generate weights based on the results of other tools.

3 Conclusion

We have assessed the state of the art in floating-point variable-precision software tools, and we have developed a new tool for mixed-precision recommendations using automatic differentiation. We have demonstrated that this tool works and can reproduce previous mixed-precision results, and we have proposed several approaches for improving its scalability. We have also described many additional avenues for future work. I hope to continue working on this project and related projects over the coming academic year and possibly into next summer as well.

Acknowledgments

First, I would like to acknowledge and thank Jeff Hittinger for funding this work. Second, I am grateful to my technical mentor Scott Lloyd and the AD project lead Harshitha Menon for their guidance, and to my student Garrett Folks for his assistance. I would also like to thank Daniel Osei-Kuffuor and Barry Rountree for providing advice at various points throughout the summer. Finally, I would like to thank Max Sagebaum, a developer and current maintainer of CoDiPack, for answering questions regarding that software package.

References

[1] Codipack – code differentiation package. `http://www.scicomp.uni-kl.de/software/codi/`. Accessed: 2017-06-20.

[2] Chiang, W.-F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., and Rakamari, Z. Rigorous Floating-Point Mixed-Precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)* (New York, NY, USA, 2017), ACM, pp. 300—-315.

[3] Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., and Tatlock, Z. Toward a standard benchmark format and suite for floating-point analysis. In *Numerical Software Verification: 9th International Workshop, NSV 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, S. Bogomolov, M. Martel, and P. Prabhakar, Eds. Springer International Publishing, 2017, pp. 63–77.

[4] Darulova, E., and Kuncak, V. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS) 39*, 2 (2017), 8:1—-8:28.

[5] Gaffar, A. A., Mencer, O., Luk, W., Cheung, P. Y. K., and Shirazi, N. Floating-point bitwidth analysis via automatic differentiation. In *Proceedings - 2002 IEEE International Conference on FieId-Programmable Technology, FPT 2002* (2002).

[6] Griewank, A. On automatic differentiation. In *In Mathematical Programming: Recent Developments and Applications* (1989), Kluwer Academic Publishers, pp. 83–108.

[7] Hascoët, L., and Pascual, V. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software 39*, 3 (2013).

[8] Hogan, R. J. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Trans. Math. Softw. 40*, 4 (July 2014), 26:1–26:16.

[9] 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, jun 2013), ACM Press, p. 369.

[10] Lam, M. O., and Rountree, B. L. Floating-Point Shadow Value Analysis. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools* (Piscataway, NJ, USA, 2016), ESPT '16, IEEE Press, pp. 18–25.

[11] 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, nov 2013), ACM Press, pp. 1–12.

[12] Tange, O. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine 36*, 1 (Feb 2011), 42–47.