Lightweight, Reusable Models for Dynamically Tuning Data-Dependent Code

David Beckingsale, Olga Pearce and Todd Gamblin
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, Livermore, CA 94550
Email: {david, olga, tgamblin}@llnl.gov

Performance portability is extremely important for modern simulation codes. Algorithms must run efficiently on an increasing number of target architectures, but choosing the fastest threading model, block size, thread schedule , and other tuning parameters for each architecture is a daunting task. The performance of modern scientific codes depends not only on the target architecture, but also on data-dependent aspects of the code.

Depending on a code's input, different physics modules may call the same kernel in many ways. Different materials in the same mesh element may require different types of physics to be executed by the same kernel, or the kernel may handle a wide range of patch sizes in an adaptive mesh refinement (AMR) application. Moreover, such behaviors may evolve and change over the course of a run. These dynamic applications require *on-line* tuning to achieve the fastest performance, as tuning parameters may need to be set based on the state of each kernel invocation.

Large codes contain thousands of independent kernels, and developing and maintaining multiple versions of each one is infeasible. Templated portability layers such as RAJA [1] and Kokkos [2] have emerged to fill this gap; they allow developers to separate the concerns of tuning and correctness. Developers write each kernel only once, and they set tuning parameters independently for each architecture at compile time.

I. Related Work

Auto-tuning is a broad field, and there has been much existing work on tuning the performance of codes. Extensive work has been done in the area of compile-time auto-tuning [3, 4, 5] as well as run-time adaptation and compilation [6, 7, 8, 9, 10]. Existing approaches typically perform a large, guided search of a performance parameter space, running dummy invocations of the kernel to determine the best assignment of tuning parameters. Despite a host of techniques that prune the search space and reduce the required number of trials, the fastest runtime searches today still take minutes to run. Existing tuning approaches work only if the code's behavior changes slowly. The tuner must either be run infrequently, to amortize the large cost, or on separate resources from the application. With caching, decisions can be made faster, but a search must generally be run for each kernel. AMR applications evolve much more dynamically, and many different tunings of the same kernel may be needed within a single time step.

II. Approach

To cope with this complex tuning problem, we use machine-learning to generate classifiers that can rapidly predict the fastest parameter values for a kernel at runtime. Using measurement collected from training runs, we gather features such as the total number of iterations the kernel will perform, or the number of instructions in the kernel body. We use these to pre-train a decision tree classifier, and we generate light-weight code for the classifier that can be used on-line to dynamically select tuning parameters for each kernel. Our approach avoids the costly hill-climbing and search algorithms associated with other dynamic auto-tuning approaches. To the best of our knowledge, this is the first technique to approach tuning at this granularity with this level of generality *and* responsiveness.

Our work is implemented in *Apollo*, an extension of the RAJA performance-portability library. RAJA allows static tuning through specialized *policy* template parameters, that allow the programmer to select the programming model back-end an application kernel will be executed by. We introduce a unified interface to collect training data from trial runs, and to collect dynamic measurements as input to our runtime decision models.

III. Results

We apply Apollo to two mini-applications and one production multi-physics application. To generate training data, we run each application and problem configuration at a range of global problem sizes, capturing the range of computational performance attributed to hardware specifications such as cache size. From the training data collected we generate decision models that can be compiled and loaded by the application at runtime to make tuning decisions.

Figure 1 shows the speedup provided by the models for each parameter in both applications running on a single node. For LULESH and CleverLeaf, these predictions speed up execution by as much as 4.8x. The speedup for ARES is 1.15x. It is important to note that in these are total application wallclock time speedups, and only one physics component of ARES currently uses RAJA.

IV. Conclusion

In this poster, we introduce Apollo, a framework to generate and use lightweight, dynamic decision tree models to tune application parameters at runtime. We have used our framework

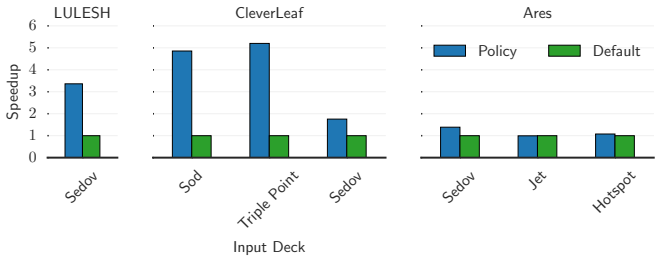

Fig. 1: Speedups when using the online models for each parameter in each application.

in both a proxy application, and with several different input problems to a production multi-physics code. Apollo models are accurate when cross-validated (up to 0.98 accuracy) and achieve real-world speedups from 1.2x to 4.8x. The main limitation of our technique is that it requires model training to be performed ahead of time on a range of representative training data. Our approach can be generalized across performance-portability frameworks and furthermore, can be extended to support additional application parameters easily.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-697318).

References

[1] R. D. Hornung and J. A. Keasler, "The RAJA Poratability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, Sep. 2014.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3202–3216, Dec. 2014.

[3] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, Jan. 2001.

[4] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[5] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, Feb. 2004.

[6] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," in *SC '02: Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, 2002, p. 44.

[7] I.-H. Chung and J. K. Hollingsworth, "Using Information from Prior Runs to Improve Automated Tuning Systems," in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, 2004, pp. 30–30.

[8] A. Tiwari and J. K. Hollingsworth, "Online Adaptive Code Generation and Tuning," *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pp. 879–892, 2011.

[9] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame, "Auto-Tuning Full Applications: A Case Study," *International Journal of High Performance Computing Applications*, vol. 25, no. 3, pp. 286–294, Aug. 2011.

[10] I.-H. Chung and J. K. Hollingsworth, "A Case Study Using Automatic Performance Tuning for Large-Scale Scientific Programs," *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pp. 45–56, 2006.