Performance Analysis of Applications in the Context of Architectural Rooflines

Boyana Norris
University of Oregon
norris@cs.uoregon.edu

Wyatt Spear
University of Oregon
wspear@cs.uoregon.edu

Allen Malony
University of Oregon
malony@cs.uoregon.edu

ABSTRACT

Intuitive visual representations of architecture capabilities and the performance of applications are critical to enabling effective performance analysis, which in turn guides optimizations. The Roofline Model and its derivatives provide such an intuitive representation of the best achievable performance on a given architecture. The Roofline Toolkit project is a collaboration among researchers at Argonne National Laboratory, Lawrence Berkeley National Laboratory, and the University of Oregon and consists of three principal components: hardware characterization, software characterization, and data manipulation, which includes a visualization interface. These components address the different aspects of performance data acquisition and manipulation required for performance analysis, modeling and optimization of applications. In this paper we introduce an implementation of the third component, a system for visualizing roofline charts and managing roofline performance analysis data. We demonstrate analysis of an application use case within this framework and outline future directions for this type of performance analysis and visualization.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

performance, analysis, visualization

1. INTRODUCTION

The Roofline model [17] enables programmers to visualize the performance potential of algorithms by introducing a simple way to quantify the computation's locality and parallelism and present them in the context of a given architecture's capabilities. Until recently Roofline models were typically laboriously created through (1) collection of hardware performance data, e.g., with micro benchmarks; (2) manual code analysis to determine the arithmetic intensity of the algorithm(s) being studied; and (3) visualizing both the architectural rooflines and the kernel's expected performance under different optimization assumptions. Automating most of this process has been the goal of the Roofline Toolkit Project. The development of portable microbenchmarks that automate the first step is discussed in [13]. In this paper we first introduce the current state of the data representation and visualization infrastructure required to automate the third step. The main contribution is a *usable software framework* for generating and visualizing architectural rooflines. We also discuss future research on automation performance bottleneck analysis and the generation of *accurate, fine-grained, understandable* performance models that can significantly improve the current state of the art in terms of precision, usability, and efficiency.

ICPE'17, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3030207.3030232

2. BACKGROUND

We briefly overview the roofline model and the tools that were leveraged in our research and implementations to date.

2.1 Roofline Analysis

For any machine model, we can evaluate the upper bound on performance by using the roofline model introduced by Williams et al [17]. Given the arithmetic intensity of an algorithm, the roofline model defines an upper limit on kernel performance P_k with the equation, $P_k = minP_f, BA_i$ where P_f is the peak hardware floating-point performance, B is peak bandwidth, and A_i is the arithmetic intensity, typically expressed as the ratio of floating-point operations to bytes transferred to/from memory.

The Roofline model and its extensions (e.g., for energy [9]) provide a compact representation of the *architectural* capabilities as a context that enables visualization of the current and *potential* performance of a computational kernel within its algorithmic and architectural constraints. As operational intensity increases memory bandwidth's limiting influence decreases until the flat, peak bound of processor GFLOPS is reached. Placement of compute kernels on the graph indicates their relationship with the system's theoretical peak performance given their operational intensity.

2.2 Eclipse

Eclipse [1] is a popular and extensible software development platform. The default set of plugins is designed for Java development, but the Eclipse community has provided

support for other languages including C/C++ and Fortran. Support for high performance computing has also been provided via the Parallel Tools Platform (PTP) [2]. The portability and extensibility of the Eclipse platform motivated our choice to use it as the basis of the Roofline visualization framework.

3. CURRENT STATE

In this section we discuss the current state of the ongoing efforts to automate performance analysis and modeling in the context of architectural rooflines.

3.1 Visualization Implementation

The initial roofline visualizations were implemented using general-purpose scientific charting tools such as Gnuplot [18]. This was adequate for developing and testing the roofline system and and for the performance analysis activities of experts. It was clear, however, that general adoption of the roofline system would benefit greatly from a simpler automated means of visualizing the performance data. Moreover, given the intended major use case of comparing the performance of multiple applications or functions to establish best-case performance behavior with respect to the roofline model, a framework that allows rapid and easy analysis would be of benefit even to experts with other visualization techniques at their disposal.

We implemented the roofline visualization system using JavaFX [3]. The new graphing functionality provided in the JavaFX API allows reasonably sophisticated visualizations without relying on external libraries as long as a relatively recent version of Java is available.

The data for visualizing Roofline architectural profiles is generated by a collection of portable micro benchmarks [13]. By default, a single set of architectural roofline data is displayed. Multiple roofline datasets may be loaded simultaneously, either from the local filesystem or from a remote repository enabling rapid switching of views or overlaying rooflines for comparison. The intersection points and the inflection points on the roofline chart may be selected to display the specific recorded metric values.

Figure 1 shows the rooflines generated for the NERSC Cray XC30 Edison supercomputer[1]. The left side of Figure 1 shows just the non-interactive architectural roofline plot generated by the ERT [13]. The top GFLOP/s rate achieved by the micro-benchmark data is indicated with the top orange line. The right side of Figure 1 is a screenshot of the interactive roofline visualization Eclipse-based tool we developed, showing the arithmetic intensity of the functions in the MiniFE proxy application [12] that take a significant portion of the execution time. In some cases lines corresponding to different hardware components overlap. Different rooflines reflect peak capabilities of different hardware components with respect to the operational intensity (x-axis) of the computation. We can also visualize different versions of the same functions in a single plot, which enables study of the effects of optimizations on the operational intensity and performance.

At present application developers can evaluate their application performance at the level of individual routines with respect to the architectural roofline models of systems where it is being or will be run. This will provide valuable insight into the question of attainable performance gains. Application performance profiles stored in TAUdb databases are searchable from the Eclipse interface and plotted on the chart of the selected roofline. The Eclipse UI supports selection of code elements from the project source tree which will facilitate examination of the performance of specific routines with respect to the architecture roofline. This requires accurate measurement of GFLOPS/byte when generating application profiles. Collecting this data is not trivial, as shown in [16], but a capability we hope will be provided in future releases of TAU and other performance analysis systems and then integrated into the Roofline visualization framework.

[1]Edison has 5576 nodes, each with two 12-core Intel "Ivy Bridge" processor at 2.4 GHz and 64 GB DDR3 1866 MHz memory, 460.8 Gflops/node peak per node.

3.2 Roofline Data Management

There are two primary elements to roofline visualization data. The systems where the roofline is being modeled are typified by the benchmarked upper bounds on memory throughput and computational intensity. These values and metric names are stored as name-value pairs. Memory throughput values are typically subdivided into the FLOPs per byte capacity supported by the different cache layers. Additional metrics representing the technical specifications may be included along with the empirically collected data. The simplicity of these data accommodate a wide range of data presentation options. We have selected JSON [4] for roofline data storage because it is simple to work with and there is strong support for it in Java and Python.

In addition to the core metrics of the architectural and application performance data, the data format must allow integration of metadata for systems and experimental trials. This is necessary to establish the provenance of collected data, to avoid duplicate trials and to allow searching and comparison of task specific data from what may be a very large general collection of system models. A robust metadata system also supports more advanced analytical features under development, such as comparison between rooflines of different systems or between the same system with altered software or hardware parameters.

The nature of roofline analysis lends itself to central, publicly available data repositories. Because the system benchmarks are useful to all developers working in the same environment it makes sense to make these a common resource. Motivated by this community oriented use case, the roofline visualization system supports accessing roofline data from a remote repository. A roofline data library is being assembled, hosted by the University of Oregon. As of this writing it is organized using simple name based selection of rooflines from the available systems. It is also possible to use metadata values as criteria to search the repository.

4. FUTURE DIRECTIONS

While viewing arithmetic intensity at the level of function and loop granularity is a useful capability, all current tools *do not* provide a comprehensive and usable modeling infrastructure. In this section we discuss outstanding research and implementation challenges.

4.1 Performance Experiments, Analysis, and Model Generation

Generating fine-grain performance models automatically is essential for fast and accurate identification of sources of

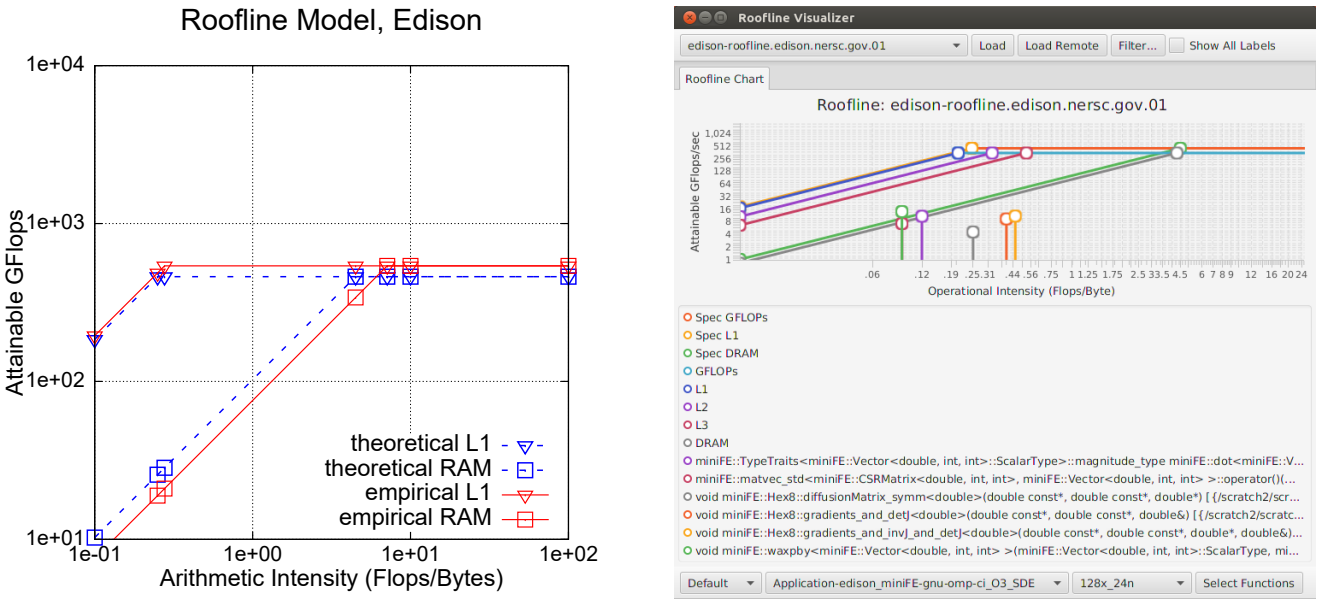

Figure 1: Left: Architectural roofline plot for Edison, a Cray XC30 supercomputer. Nodes are 12-core Intel "Ivy Bridge" processors (2.4 GHz) with 64 GB memory. Right: Application analysis for the MiniFE mini-app [12].

performance degradation and strategies to optimize the corresponding computations. For example, Calotoiu et al. [8, 7] show that it is possible to generate scalability models from empirical data. These models require only wall-clock time but even this relatively simple measurement involves running an application at varying scales. Selecting problem sizes for each experiment are typically user-defined tasks, which can be both error-prone (because of lack of complete knowledge on how to scale problem sizes in complex codes) and potentially wasteful (e.g., running too many large-scale experiments that are not necessary for constructing the model).

When modeling single-node performance, a number of additional complications arise. Consider estimating arithmetic intensity as defined in the roofline model, which requires measurement of floating-point operations and main memory traffic. Even homogeneous platforms do not provide consistent interfaces for obtaining accurate measurements of these quantities. On each platform, the user must learn how to install and use a mix of tools with different interfaces and information granularities. In some cases, there are no hardware counters for the quantity of interest (e.g., floating-point operations on Intel Haswell servers). The alternatives are manual estimates for time-intensive portions of the application (an approach taken by some HPC numerical packages such as PETSc [5, 6], or using static source code analysis to estimate such quantities [15, 11]. The static approach allows the generation of more abstract models, which can then be modified by the user to reflect planned or necessary optimizations, with the resulting performance visualized in the context of architectural rooflines.

Arithmetic intensity is just one way of viewing the efficiency of computations. Other metrics of interest include energy, or domain-specific quantities such as number of vertices or edges processed per second in graph computations. When used in the context of the architectural roofline, arithmetic intensity or a similar measure of *useful work* is a good first step in identifying functions that should be targeted for optimizations. Arithmetic intensity by itself, however, does not provide information to inform the optimization decisions, e.g., are the GFLOP/s lower because a significant loop was not vectorized or because of conflict misses in a multithreaded computation? To identify root causes, more detailed (automated) measurements are required. Again, the lack of common interfaces and tool infrastructure across platforms makes this a challenging and time-consuming task. Increasing heterogeneity at node level is also not addressed by existing measurement approaches – users are expected to use separate tools for CPUs and GPUs. Merging data from different tools is also non-trivial because of differences in sampling rates or types of measurements available. Our Autoperf framework [10] implements some initial steps toward automating complete performance modeling (and eventually optimization) workflows. Much remains to be done before performance experiment automation is widely available in support of performance analysis and model generation on modern heterogeneous platforms.

So far we have considered mostly empirical performance analysis and modeling. Because of the challenges in obtaining accurate measurements across platforms, we believe that static program analysis will play an increasingly important role in the analysis and model generation process. For example, we are pursuing static approaches that consider both source code and binary analysis (on both CPUs and GPUs) to generate models for metrics such as arithmetic intensity.

4.1.1 *Visualization*

The roofline visualization system remains under development and there are a number of features we anticipate adding as the project proceeds. These will dovetail with the continuing development of the roofline data collection and analytical utilities also under development.

Comparison between systems and between multiple sets of application trial data within a single chart will be useful in performance engineering operations that incorporate

roofline data. Devising good visual representations that are informative rather than cluttered and confusing has been an ongoing challenge. Beyond single metrics, the effectiveness of different approaches to 2-D or 3-D representations [14] of multidimensional data merit future investigation.

We are also planning to increase the integration between the Eclipse framework and the visualization system. Enabling control of performance measurements, analysis and models from within the IDE is a natural next step. Rather than reimplementing existing tools, leveraging approaches (e.g., Autoperf [10]) that interface with existing measurement and analysis tools would be preferable.

The computation and visualization of arithmetic intensity (and other emerging algorithmic metrics) can be integrated into the Eclipse source code views, so that users can easily visualize the current and potential performance of selected computations as they are developing them. To accomplish this we will implement two main components of the Roofline Toolkit—developer-aided static model generation and empirical performance data integration.

In its ultimate form the visualization system, in conjunction with other roofline data collection mechanisms, should enable system designers to easily compare and typify fundamental performance characteristics for proposed and existing hardware, help software developers to explain and tune the performance of their computational kernels in a hardware-aware fashion and encourage sharing and use of system and application level performance data.

5. CONCLUSION

We have introduced a new Eclipse-based performance visualization system that shows loop-level performance information in the context of architectural rooflines. The generation of the hardware roofline profiles is fully automated. Most of the application performance data gathering on Intel architectures is also automated, as is the computation of the derived metrics required for displaying performance in the context of the rooflines. We then discuss the ways in which this framework can be extended to enable more complete automation of performance measurement, analysis, and model generation on modern heterogeneous platforms.

Acknowledgments

This work was supported in part by DOE Grant DE-SC0004510.

6. REFERENCES

[1] Eclipse IDE. http://www.eclipse.org. Accessed: 2014-10-20.

[2] Eclipse Parallel Tools Platform. http://www.eclipse.org/ptp/. Accessed: 2014-10-20.

[3] JavaFX. http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html. Accessed: 2014-10-20.

[4] JSON. http://www.json.org. Accessed: 2014-10-20.

[5] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2016.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[7] A. Calotoiu, D. Beckingsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast Multi-Parameter Performance Modeling. Oct. 2016. Accepted at IEEE International Conference on Cluster Computing (Cluster'16).

[8] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 45:1–45:12, New York, NY, USA, 2013. ACM.

[9] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 661–672, May 2013.

[10] X. Dai, B. Norris, and A. D. Malony. Autoperf: Workflow support for performance experiments. In *Proceedings of the Workshop on Challenges in Performance Methods for Software Development (WOSP-C'15), Austin, Texas*, 1 2015.

[11] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, , and W. Jalby. MAQAO: Modular assembler quality analyzer and optimizer for Itanium 2. In *Proceedings of the Workshop on Explicitly Parallel Instruction Computing Techniques, Santa Jose, California, March*, 2005.

[12] M. A. Heroux, D. W. Doerer, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Sept. 2009.

[13] Y. Lo, S. Williams, B. Van Straalen, T. Ligocki, M. Cordery, N. Wright, M. Hall, and L. Oliker. *Roofline Model Toolkit: A practical tool for architectural and program analysis*, volume 8966 of *Lecture Notes in Computer Science*, pages 129–148. Springer Verlag, 2015.

[14] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. 3dyrm: a dynamic roofline model including memory latency information. *The Journal of Supercomputing*, 70(2):696–708, 2014.

[15] S. H. K. Narayanan, B. Norris, and P. D. Hovland. Generating performance bounds from source code. In *Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)*, 9 2010.

[16] G. Ofenbeck, R. Steinmann, V. C. Cabezas, D. G. Spampinato, and M. PÃijschel. Applying the roofline model. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, 2014.

[17] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[18] T. Williams, C. Kelley, and many others. Gnuplot 4.4: an interactive plotting program. http://gnuplot.sourceforge.net/, March 2010.