Author Retrospective for
Optimizing Matrix Multiply using PHiPAC: a Portable High-Performance ANSI C Coding Methodology

Jeff Bilmes
EE Department
University of Washington, Seattle
bilmes@ee.washington.edu

Krste Asanović
EECS Department
University of California, Berkeley
krste@eecs.berkeley.edu

Chee-Whye Chin
Department of Mathematics
National University of Singapore
cheewhye@math.nus.edu.sg

Jim Demmel
EECS Department
University of California, Berkeley
demmel@cs.berkeley.edu

Original paper: http://dx.doi.org/10.1145/263580.263662

1. INTRODUCTION

PHiPAC was an early attempt to improve software performance by searching in a large design space of possible implementations to find the best one. At the time, in the early 1990s, the most efficient numerical linear algebra libraries were carefully hand tuned for specific microarchitectures and compilers, and were often written in assembly language. This allowed very precise tuning of an algorithm to the specifics of a current platform, and provided great opportunity for high efficiency. The prevailing thought at the time was that such an approach was necessary to produce near-peak performance. On the other hand, this approach was brittle, and required great human effort to try each code variant, and so only a tiny subset of the possible code design points could be explored. Worse, given the combined complexities of the compiler and microarchitecture, it was difficult to predict which code variants would be worth the implementation effort.

PHiPAC circumvented this effort by using code generators that could easily generate a vast assortment of very different points within a design space, and even across very different design spaces altogether. By following a set of carefully crafted coding guidelines, the generated code was reasonably efficient for any point in the design space. To search the design space, PHiPAC took a rather naive but effective approach. Due to the human-designed and deterministic nature of computing systems, one might reasonably think that smart modeling of the microprocessor and compiler would be sufficient to predict, without performing any timing, the optimal point for a given algorithm. But the combination of an optimizing compiler and a dynamically scheduled microarchitecture with multiple levels of cache and prefetching results in a very complex, highly non-linear system. Small changes in the code can cause large changes in performance, thwarting "smart" search algorithms that attempt to build models of the performance surface. PHiPAC instead used massive, diverse, offline, and only loosely informed grid or randomized search, where each point was evaluated by empirically measuring the code running on the machine, which succeeded due to its indifference to the shape of the performance surface.

Of course, a clever human must first create the set of possible design spaces in the first place, as well as efficient search, timing, and code-generation methods, and update everything as technologies change. This is as it should be, with humans operating at higher levels of abstraction where creativity can have broader payoffs.

In retrospect, the idea of using a computer instead of a human to search a combinatorially large design space seems obvious. Techniques analogous to autotuning were of course used previously by many researchers in software and hardware, for example, design-space exploration in CAD. Intelligent search had also already become an integral part of many of the algorithms used to solve problems in artificial intelligence [10]. At the time, however, PHiPAC was received with some skepticism. As we explain below, PHiPAC was able to quickly achieve performance that was on par with some of the best vendor-supplied numerical libraries available at the time. Therefore, the tides quickly turned, and autotuning is now widely used in many fields [5].

2. HISTORY

The PHiPAC project grew out of work in the early 1990s within the Realization Group at the International Computer Science Institute (ICSI), who were using large artificial neural networks (ANN) for speech recognition. The training algorithms for these ANNs were very computationally intensive, but could be formulated using a few dense linear algebra kernels. The ICSI group had built custom DSP multiprocessors (the RAP machine [9]) to accelerate these algorithms, and was building a custom chip, the T0 vector microprocessor [13], to provide higher performance at lower cost.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).
ICS 25th Anniversary Volume. 2014
ACM 978-1-4503-2840-1/14/06.
http://dx.doi.org/10.1145/2591635.2591656 .

In addition to the custom computing machines, the speech researchers at ICSI regularly used their network of desktop SPARC workstations to run ANN training jobs, with a locally customized version of pmake used to distribute jobs among all the idle machines. Although all the workstations were binary compatible, we noted that six different SPARC microarchitectures were in use around the institute with widely varying floating-point performance. This was also a period of rapid development in RISC workstation architectures, and we wanted to be able to quickly and effectively exploit any new machines as they became available. Although the BLAS library interface was well-established by this time, high-performance library implementations were only available for a few machines and often came at significant additional cost. Freely available portable matrix-vector libraries contained naïve code with low performance. In addition, we wanted to retain a compatible interface to the T0 vector microprocessor, and so we defined our own simple floating-point vector library "fltvec".

The ANN training codes used primarily matrix-vector operations (BLAS2) at this time, and we began handwriting some optimized C matrix-vector routines for fltvec in early 1994. During this process, we figured out most of the portable high-performance C coding rules that eventually appeared in Section 2 of the paper [2], as an explicit goal of the library was to attain good performance across the range of workstations in use at ICSI. The name "PHiPAC" ("feepack", following the Greek pronunciation of ϕ) was actually initially coined to refer just to this coding style, though we eventually used it to cover the entire autotuning process.

Later in 1994, the speech group was seeking greater performance and wanted to use faster matrix-matrix operations (BLAS3) on both the workstations and the T0 accelerator. By December 1994, we had completed an initial matrix-multiply routine in C that was parameterized using a very large fixed set of preprocessor macros, and began tuning and evaluating its performance on various machines.

In the spring of 1995, we taught the CS267 "Applications of Parallel Computers" class at Berkeley. In this class, students entered a matrix-multiply contest and had to compare their results with the current version of our matrix-multiply code. We continued to add optimizations and code variants to the matrix-multiply code, but found that the preprocessor macro approach didn't scale due to the combinatorial explosion when combining different options. We decided to change our approach and instead write a program to generate the code variant directly, including only the specific macros that were needed for the given set of blocking parameters. The first version of this new matrix-multiply code generator, which would eventually form the core of the PHiPAC release, was completed around the end of April 1995.

Using the new code generator, we ran timings on many more machines including an SGI MIPS R4000-based Indigo and an IBM RS6000. The IBM RS6000 came with the ESSL BLAS libraries, which were extensively manually tuned and well regarded, and we were pleasantly surprised when we found we could quickly achieve similar performance using our generated code. At this time, we were still manually changing parameters to explore the design space, but realized that it should be possible to automate the search.

In the summer of 1995, Dominic Lam, an undergraduate, joined the project and began work on Perl scripts to automatically search (i.e., generate code and then time instances within) the design space of the generator. We used simple heuristics to limit the design space the autotuner would explore, but within the feasible set we performed a simple exhaustive grid search that could take days to complete. To help us more quickly get a sense of the tuning space, we added a random search strategy, and were surprised at how often this would quickly find a good point, if not the best point, in the space. The summer was spent refining the search strategy, which was complicated by the need to tune for each level of the memory hierarchy, from innermost (register blocking) to outermost (level 2 cache blocking).

Each new machine we tried would usually require some additional optimization or improvements in search strategy to get better performance, though the current version would usually give reasonable performance when first run on a new machine. As an example, in November 1995, Greg Henry started using the PHiPAC generator on the newly introduced Intel Pentium Pro machines, which represented the first x86 machines with performance competitive to the RISC workstations. When run on the Pentium Pro, PHiPAC automatically selected a very different register blocking than for the register-rich RISC workstation floating-point units. Although the initial PHiPAC-generated code was reasonably fast (around 60% of peak), hand-tuned assembly code was running significantly faster (>80% of peak). PHiPAC relies on a optimizing compiler to take care of low-level scheduling, and at the time, the code quality of the available x86 compilers was not as high as that from the RISC workstation vendors. Later modifications to PHiPAC, and better x86 compilers, led to much improved performance on x86 platforms in our later release.

We released the first PHiPAC code in December 1995, and started to receive growing interest in the code and the technique. We also experienced a lot of skepticism from the compiler community, who thought compilers could generate good code for matrix multiply, and library authors, who thought only human-tuned code could attain near-peak performance. Our experiences were that even the most highly regarded commercial compilers failed dismally on matrix multiply code, and that PHiPAC often produced code that matched or beat vendor-supplied hand-tuned libraries. Further, for many machines, high-performance libraries weren't available or were prohibitively expensive to license.

We continued to improve the search strategy and the code generation. The search strategy was made more intelligent to focus on just the runs needed to tune at each level of the memory hierarchy, while the code generator was similarly improved to generate code specialized for each level of the memory hierarchy, the lowest level including special features like explicit software pipelining [3]. Later in 1996, we prepared a submission to the Supercomputing conference, which was rejected partly because a reviewer believed the ESSL numbers against which we were comparing were from the FORTRAN compiler!

We reworked the paper and submitted it to ICS-1997, where thankfully it was accepted [2]. As we worked on the camera-ready version of the paper, we had a particularly satisfying visit to SGI where Ed Rothberg had kindly made available machines containing MIPS R8000 and R10000 processors. Within a single afternoon, we had the autotuner producing matrix-multiply routines that were competitive with the vendor BLAS libraries for these two very different microarchitectures. Given that we had not previously

worked with these machines or their compilers, this was strong evidence that the autotuning approach could quickly provide high-performance code for new architectures.

Returning to the original impetus for the project, we used PHiPAC-generated code to improve neural network training, resulting in significant speedups [4]. Due to their excellent performance in many applications such as speech and computer vision, the many layered "deep" variants [8] of neural networks have recently been experiencing a resurgence of interest, making autotuned implementations [4] perhaps even more relevant today.

We released the second and final version of PHiPAC in spring 1998 [3]. By this time, several other efforts had sprung up to produce autotuners, most notably the ATLAS project [1], which was directly inspired by PHiPAC. Independently, but a little later than PHiPAC, a group at MIT started the FFTW effort to produce autotuned FFTs [7].

3. AUTOTUNING TODAY

The field of autotuning, or automated design-space exploration, has grown in many directions over the years. This short retrospective does not permit us to survey the entire field, or even just the hundreds of papers that have cited our early work on PHiPAC. But we can summarize their common motivation: the design space of different ways to implement an algorithm on a particular piece of hardware is so large, and the impact of even small implementation changes on the running time is so difficult to predict, that it is much more productive to automate the process of searching the design space and to use empirical timing results. This still leaves the programmer the challenging problems of defining the design space, and finding an algorithm to search it efficiently, a much better level of abstraction at which to work. Autotuning is now widely used to generate code not just for dense linear algebra [1] but also sparse linear algebra [5], FFTs [6] more general digital signal-processing algorithms [11], and many other examples [12].

The survey paper [5] summarizes a subset of this subsequent work that has been carried on by some of the PHiPAC authors and other more recent contributors, and gives some pointers to other literature. One interesting example is the continued use of the matrix multiply contest as a homework in our CS267 class. Now the students learn about autotuning before the assignment, and are told that autotuning is a possible (but not required!) way to do the assignment. For the class offering discussed in [5], 3-student teams built autotuners in the 2 weeks they had for the assignment, automatically generating and searching over tens of thousands of lines of code. More recently, matrix multiply has been adopted in UC Berkeley's newly reorganized 3rd semester undergraduate programming class CS61C. Again, some of the sophomores choose to build their own autotuners to explore the design space.

Looking at ongoing and future work, the challenges and opportunities are large and growing. Sometimes tuning cannot be done "offline", taking a large amount of time to search the design space and building a library that is called at run-time, but must be done in part at run-time, when the user's input data is available. This is because the best algorithm may strongly depend on the inputs. For example, for sparse-matrix operations, the best algorithm depends on the sparsity pattern of the matrix. Sometimes the tuning space is so large that it is even impractical to tune it exhaustively offline, requiring clever machine learning techniques to search faster. Finally, sometimes the search space includes not just changes to the software, but also changes to the hardware as well [11], e.g. when one is trying to build specialized hardware to solve a problem, making the search space larger still.

To conclude, our work on PHiPAC has been an example of a larger pattern underlying much successful computer science research: the goal of research project $n+1$ is to automate the successful ideas from research project n.

4. ACKNOWLEDGEMENTS

We would like to thank Dominic Lam for his work on the first release of PHiPAC, and Rich Vuduc for experimenting with PHiPAC after the ICS paper.

5. REFERENCES

[1] ATLAS: Automatically Tuned Linear Algebra Software. math-atlas.sourceforge.net/, 2014.

[2] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, pages 340–347, Vienna, Austria, July 1997. ACM SIGARC.

[3] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. The PHiPAC v1.0 Matrix-Multiply Distribution. Technical Report UCB/CSD-98-1020, Computer Science Division, University of California at Berkeley, 1998. also ICSI technical report TR-98-035.

[4] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel. Using PHiPAC to speed error back-propagation learning. In *Proceedings of ICASSP*, volume 5, pages 4153–4157, Munich, Germany, April 1997.

[5] J. Demmel, J. Dongarra, A. Fox, S. Williams, V. Volkov, and K. Yelick. Autotuning: Accelerating Time-to-Solution for Computational Science and Engineering. In *SciDAC Review, v. 15*, Winter 2009. available at www.scidacreview.org.

[6] FFTW. www.fftw.org, 2014.

[7] M. Frigo. *Portable High-Performance Programs*. PhD thesis, Massachusetts Institute of Technology, June 1999.

[8] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[9] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. The Ring Array Processor (RAP): A Multiprocessing Peripheral for Connectionist Applications. *Journal of Parallel and Distributed Computing*, 14:248–259, April 1992.

[10] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[11] Spiral: Software/Hardware Generation for DSP Algorithms. www.spiral.net, 2014.

[12] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[13] J. Wawrzynek, K. Asanović, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, March 1996.