Using PHiPAC to Speed Error Back-Propagation Learning

Jeff Bilmes, Krste Asanovic,
Chee-whye Chin, and Jim Demmel.
CS Division,
University of California at Berkeley
and the
International Computer Science Institute
www.icsi.berkeley.edu/~bilmes/phipac

Abstract.

We introduce PHiPAC, a coding methodology for developing high-performance numerical libraries in ANSI C. Using this methodology, we have developed optimized code for matrix multiply. The resulting routines can achieve over 90% of peak performance on a variety of current workstations, and are often faster than vendor-supplied optimized libraries. We use the PHiPAC matrix multiply routines to implement a "bunch-mode" back-propagation learning algorithm for 2-layer multi-layered perceptrons. A bunch-mode algorithm trains with greater than one pattern at a time. By timing the resulting code, we show that the PHiPAC derived back-propagation code can lead to significantly faster training.

PHiPAC

It is important to have optimized linear algebra and signal processing routines such as matrix-multiplication, convolution, cross-correlation, IIR filtering, FFT, etc. Unfortunately, producing an efficient library routine for a given machine can be a time-consuming task. There are several methods commonly used to produce high-performance code for such routines, including:

Assembly Coding:
The routines could be manually written in assembly code, but fully exploring the design space is difficult. Furthermore, the code might be unusable or sub-optimal on a different system.

High-Level Language, Manual Tuning:
The routines code be coded in a high-level language and manually tuned to match the underlying architecture, but this approach still requires writing machine specific code that is not performance-portable.

High-Level Language, High-Performance Compiler:
Ideally, the routines would be coded once in a high-level language and fed to an optimizing compiler. While this approach can generate good code in general, it is unlikely to generate near-peak code for any one operation. Furthermore, we have seldom found the capabilities of such high-performance optimizing compilers in production compilers available for commercial workstations. And while this situation will improve, we expect it will be many years before a single version of a library routine can be compiled to give near-peak performance on a wide variety of machines.

We have developed a methodology, named PHiPAC, for producing Portable High-Performance linear algebra and signal processing libraries in ANSI C. Our goal is to produce high-performance libraries for a wide range of systems with minimal effort. The PHiPAC strategy consists of the following three components:

1) Machine and Compiler Model:
We have developed a generic model of current C compilers and microprocessors that provides guidelines that help us attain high performance across a range of machine and compiler combinations. We found we could usually rely on reasonable register allocation, instruction selection, and instruction scheduling -- more sophisticated compiler optimizations, however, were usually not effective at producing the highest quality code. From our analysis, we determined a set of PHiPAC coding guidelines [1] that may be used directly by programmers wishing to improve performance.

2) Code Generators:
Rather than hand code individual routines, we write parameterized generators that produce code according to the PHiPAC coding guidelines. A large portion of a routines design space may be produced simply by varying the generators parameters.

3) Search Scripts:
To obtain a high-performance version of a routine, we write scripts that automatically tune code for a particular system by varying the generators' parameters and benchmarking the resulting routines.

Matrix Multiply Generator and Search Scripts

We have produced a generator `mm_gen` that produces C code following the PHiPAC coding guidelines for one variant of the matrix multiply operation C = $\alpha op(A) op(B) + \beta op(C)$ where *op*(A), *op*(B), and C, are respectively MxK, KxN, and MxN matrices, α and β are scalar parameters, and *op*(X) is either *transpose*(X) or just X. `mm_gen` produces a multi-level cache-blocked matrix multiply, restructuring the code for unit stride, and reducing the number of cache misses and unnecessary loads and stores. We also produce a search script that takes parameters describing the machine architecture and then, for each combination of generator parameters, calls the generator, compiles and links the resulting code, and benchmarks the executable. Figure 1 shows the resulting performance for matrix multiply on various machines. The generator and search scripts are available at our WWW site.

Bunch-Mode Back-propagation Learning

In [2], two modes of back-propagation learning for a 2-layer MLP were defined: on-line mode where only one training pattern is used per weight update, and batch mode where all training patterns are used per weight update. An alternate strategy, which we call bunch mode, is to use more than one but less than all training patterns per update.

If we assume the following definitions:

n_p = the bunch size.
n_i = the number of input units.
n_h = the number of hidden units.
n_o = the number of output units.
I = the n_pxn_i matrix of input patterns.
H = the n_pxn_h matrix of hidden units.
ΔH =the n_pxn_h matrix of delta hiddens.
O =the n_pxn_o matrix of output units.
ΔO = the n_pxn_o matrix of delta outputs.
T = the n_pxn_o matrix of target patterns.
W_{ih} = the n_hxn_i input-to-hidden weight matrix.
W_{ho} = the n_oxn_h hidden to output weight matrix.

Then the matlab-like code to the right is a simplified version of a single bunch-mode learning step for a 2-layer MLP where g() is the logistic function, and ".*" denotes element-wise matrix multiply.

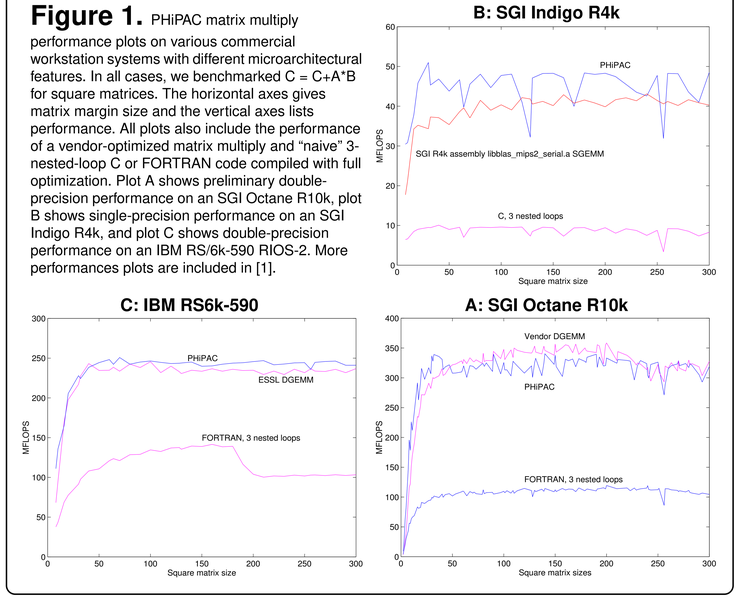
$$H = g(I * W_{ih}^T)$$
$$O = g(H * W_{ho}^T)$$
$$\Delta O = O.*(1-O).*(T-O)$$
$$\Delta H = H.*(1-H).*(\Delta O * W_{ho})$$
$$W_{ho} = W_{ho} + \eta \Delta O^T * H$$
$$Wih = Wih + \eta \Delta H^T * I$$

Since n_p controls the bunch size, n_p = 1 corresponds to standard on-line learning which uses matrix-vector operations. When n_p > 1, however, the algorithm may be implemented with the matrix-multiplication operations C = A*B^T, C = A*B, and C = C+αA^T*B. All of these can be generated by `mm_gen`.

n_p affects both the inherent convergence rate (i.e, the required number of epochs to achieve a given epoch sum-of-squared error (ESSE)) and the running time per epoch. Therefore, some n_p will achieve the best trade-off between the two.

Figure 1.

PHiPAC matrix multiply performance plots on various commercial workstation systems with different microarchitectural features. In all cases, we benchmarked C = C+A*B for square matrices. The horizontal axes gives matrix margin size and the vertical axes lists performance. All plots also include the performance of a vendor-optimized matrix multiply and "naive" 3-nested-loop C or FORTRAN code compiled with full optimization. Plot A shows preliminary double-precision performance on an SGI Octane R10k, plot B shows single-precision performance on an SGI Indigo R4k, and plot C shows double-precision performance on an IBM RS/6k-590 RIOS-2. More performances plots are included in [1].



Bunch-Mode Back-propagation with PHiPAC: Results

Figure 2 shows a series of graphs giving the MLP training performance for a standard speech data set. The benchmarked MLP has 153 input, 200 hidden, and 61 output units. All training was done on an IBM RS6k-590.

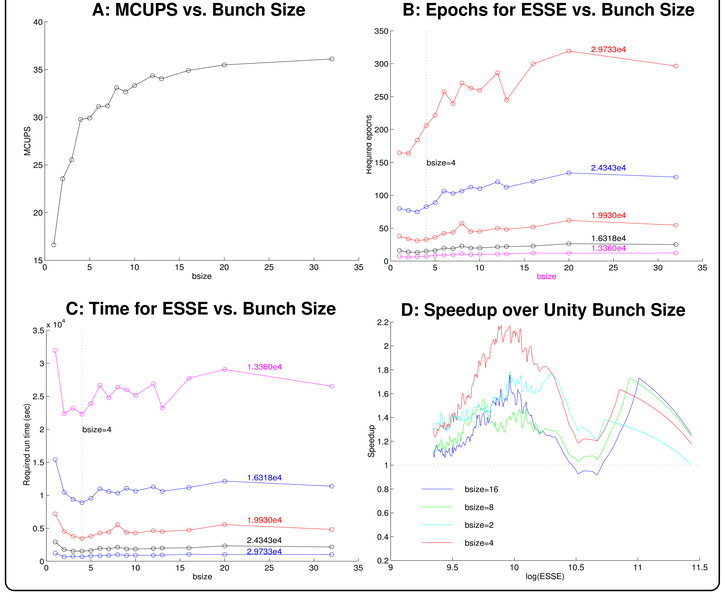
Figure 2A shows training performance in MCUPS (millions of connection-updates per second) vs. the bunch size n_p. As can be seen, as the bunch size is increased, MCUPS performance tends to get better as a result of the greater blocking opportunity given to the PHiPAC matrix multiply routines. This performance increase alone, however, does not guarantee faster training.

Figure 2B shows a plot of the number of epochs required to reach a given ESSE for different bunch sizes. The plot shows that, for this training set, it typically requires more training epochs to reach a given ESSE as the bunch size is increased from unity.

Figure 2C combines the results of Figures 2A and 2B and shows the time required to reach a given ESSE for different bunch sizes. The plot shows that the performance gained by using the PHiPAC matrix multiply routines with a bunch size greater than unity outweighs in absolute time the additional epochs necessary to reach a given ESSE.

Figure 2D shows the speedup over unity bunch size as a function of log(ESSE) for different bunch sizes. We see a greater than unity speedup for almost all the bunch sizes. The plot shows that, in this case, a bunch sizes of 4 is best, and that for all desired ESSE values, there is a greater than unity speedup.

Figure 2.



References:
[1] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. *Optimizing* Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI *C Coding Methodology*. Proc. International Conference on Supercomputing. Vienna, Austria, 1997.
[2] J. Hertz, A. Krogh, and R.G. Palmer. Introduction to the Theory of Neural Computation. Addison Wesley, 1991