USING PHIPAC TO SPEED ERROR BACK-PROPAGATION LEARNING

Jeff Bilmes[*,†], *Krste Asanović*[*,†], *Chee-whye Chin*[*], *and Jim Demmel*[*]
{*bilmes,krste,cheewhye,demmel*}*@cs.berkeley.edu*

[*]Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94704, USA
[†]International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704, USA

ABSTRACT

We introduce PHiPAC, a coding methodology for developing portable high-performance numerical libraries in ANSI C. Using this methodology, we have developed code for optimized matrix multiply routines. These routines can achieve over 90% of peak performance on a variety of current workstations, and are often faster than vendor-supplied optimized libraries. We then describe the bunch-mode back-propagation algorithm and how it can use the PHiPAC derived matrix multiply routines. Using a set of plots, we investigate the tradeoffs between bunch size, convergence rate, and training speed using a standard speech recognition data set and show how use of the PHiPAC routines can lead to a significantly faster back-propagation learning algorithm.

1. INTRODUCTION

Signal processing algorithms such as neural network learning, convolution, cross-correlation, IIR filtering, etc., are often computationally time-consuming. These algorithms, often used in time-critical applications, should therefore be coded as efficiently as possible.

One way to achieve high efficiency is to code in assembly language, but it is then difficult to make a full exploration of a routine's design space, and the resulting code might be unusable or sub-optimal on different platforms. Alternatively, the algorithms could be written in a high-level language and fed to an optimizing compiler. While there is a large literature on relevant compiler techniques [14, 10, 11, 1, 6, 12] that can be used to generate reasonably good code in general, they tend not to generate near-peak code for any one operation. Moreover, it takes significant time and investment before compiler research appears in production compilers, so these capabilities are often unavailable. Furthermore, a high-level language's semantics combined with a casual coding style might obstruct aggressive compiler optimizations even if they are available.

We have developed a methodology, named PHiPAC, for developing Portable High-Performance numerical libraries in ANSI C. Our goal is to produce, with minimal effort, high-performance numerical libraries for a wide range of systems. Using this methodology, we have produced a portable, BLAS-compatible [7], matrix multiply generator. The resulting code can achieve over 90% of peak performance on a variety of current workstations, and is often faster than the vendor-supplied optimized libraries. We have used the resulting code to produce an optimized back-propagation learning program that is much more efficient than many existing programs.

In Section 2., we briefly describe the PHiPAC methodology used to generate the matrix multiply code. In Section 3., we use the resulting matrix multiply code to implement a bunch-mode back-propagation training program that uses multiple rather than one training pattern for each weight update. We investigate the tradeoffs between bunch size, convergence rate, and training speed using a standard speech recognition data set. Finally, in Section 4. we conclude and describe future work.

2. PHIPAC

The PHiPAC methodology for producing high performance code consists of three primary components.

The first component is a generic model of current microprocessors and their C compilers – this provides guidelines for producing portable high-performance ANSI C code. By analyzing a range of machines such as workstations and microprocessor-based shared-memory multiprocessor (SMP) and massively parallel processor (MPP) nodes, we've created a list of common micro-architectural features. In addition, we've found that production ANSI C compilers usually perform reasonable register allocation, instruction selection, and instruction scheduling. However, more sophisticated optimizations, including pointer alias disambiguation, register and cache blocking, loop unrolling, and software pipelining, are best performed manually. The actual micro-architectural features along with the resulting C coding guidelines are fully described in [5].

The second component of the PHiPAC methodology is to, rather than hand-code particular routines, write parameterized generators [1, 11] that produce code according to our guidelines. A generator has several advantages over a single instance of a routine. First, the algorithm's entire design space can be explored by varying the generator parameters and timing the resulting routines. Second, and perhaps surprisingly, a generator can often be easier to write than a single optimized routine, especially when the optimized routine contains a high degree of register blocking and loop unrolling. Third, a generator's end result is *performance portable* since different versions of the code can easily be created for different microarchitectures.

The third PHiPAC component is a search script that automatically tunes code for a particular system by varying a generator's parameters and benchmarking the resulting routines. For each combination of generator parameters and compilation options, the search script calls the generator, compiles the resulting routine, links it with timing code, and benchmarks the resulting executable. We assume that we have all machine specific information, such as the number of integer and floating-point registers and sizes of each cache level, available at the start of the search.

The combination of these three components yields a technique for quickly obtaining highly optimized routines for a range of architectures. We have applied these techniques to the development of a generator and search scripts for the matrix multiply operation.

`mm_gen` is a generator that produces blocked matrix multiply code [8], following the PHiPAC coding guidelines. It generates code for the operation $C = \alpha op(A) op(B) + \beta C$ where $op(A)$, $op(B)$, and C, are respectively M×K, K×N, and M×N matrices, α and β are scalar parameters, and $op(X)$ is either $transpose(X)$ or just X. Details of the resulting code are described in [5].

Figure 1 shows the performance of naive matrix-multiply code, the vendor supplied optimized and assembly coded BLAS routine, and our matrix multiply code. See [5] for a set of additional plots showing similar PHiPAC performance advantages.

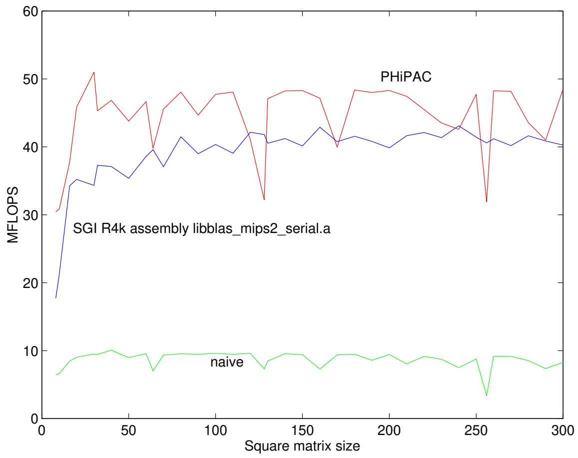

Figure 1. Performance of the naive code (3 nested loops), an SGI supplied matrix library, and PHiPAC matrix multiply for square matrices on an SGI Indigo R4K 100 MHz.

In this paper we concentrate on matrix multiplication and back-propagation, but we have produced other generators including convolution, dot-product, and AXPY, which have similarly demonstrated portable high performance.

3. BUNCH-MODE BACK-PROPAGATION LEARNING

In [9], two modes of back-propagation learning were defined, on-line mode where only one training pattern is used at a time to update the weight matrices, and batch mode where all training patterns are used simultaneously to update the weight matrices. An alternate strategy, which we call *bunch-mode* (also called block-mode [2, 3]), uses more than one training pattern simultaneously to update the weight matrices. Let n_p equal the number of training patterns being processed simultaneously (i.e., the bunch size). When $n_p = 1$, the back-propagation learning algorithm inherently uses matrix-vector operations. When $n_p > 1$, however, the algorithm can be formulated to use matrix-matrix operations. It is well known that matrix-matrix operations can be coded much more efficiently, especially for larger matrix sizes [7]. Therefore, larger bunch size back-propagation learning should have a speed advantage.

Using the PHiPAC derived matrix-multiply routines, we have therefore implemented a bunch-mode back-propagation learning algorithm. Let n_i equal the number of network inputs, n_h equal the number of hidden units, n_o equal the number of outputs. Also, let I be the $n_p \times n_i$ matrix of input patterns, H (resp. ΔH) be the $n_p \times n_h$ matrix of hidden units (resp. delta hiddens), O (resp. ΔO, $\mathcal{T}$) be the $n_p \times n_o$ matrix of output units (resp. delta outputs, target patterns), W_{ih} be the $n_h \times n_i$ input to hidden weight matrix, and W_{ho} be the $n_o \times n_h$ hidden to output weight matrix.

The following is a simplified version of a single bunch-mode learning step for a three-layer multi-layer perceptron with logistic hidden and output units. In the following (as in Matlab) $*$ denotes normal matrix multiply, $.*$ denotes element wise matrix multiply, and $g(x)$ denotes logistic function application to each element in the matrix x.

$$
\begin{aligned}
&H = g(I * W_{ih}^T)\\
&O = g(H * W_{ho}^T)\\
&\Delta O = O. * (1 - O). * (\mathcal{T} - O)\\
&\Delta H = H. * (1 - H). * (\Delta O * W_{ho})\\
&W_{ho} = W_{ho} + \eta \Delta O^T * H\\
&W_{ih} = W_{ih} + \eta \Delta H^T * I
\end{aligned}
$$

As can be seen, this algorithm can be implemented with the matrix operations $C = A * B^T$, $C = A * B$, and $C = C + \alpha A^T * B$ all of which can be generated by `mm_gen`.

The bunch size, controlled by the parameter n_p, affects two aspects of the training algorithm. As n_p changes, the inherent convergence rate — i.e., the number of epochs required to achieve a given epoch sum-of-squared error (ESSE) — can change. On the other hand, as n_p increases from 1, the time per epoch will decrease. Therefore, there is typically some n_p which achieves the best tradeoff between the two for a given learning task.

The following graphs display 3-layer MLP training performance results for a standard speech data set.[1] The network consists of 153 (9 frames of 17 features) input, 200 hidden, and 61 phonemic output units. There are roughly 75,000 training patterns. All training was done on an IBM RS6000/590, which has a peak speed of 266 MFLOPS.

Figure 2 graphs the MCUPS (millions of connection updates per second) as a function of the bunch size. While we are guaranteed an increase in MCUPS as the bunch size increases from 1 because of better matrix blocking, we are

[1] A telephone quality database of digits and control words from Bellcore that we call "digits+".

not in general guaranteed that the convergence rate will not offset these gains and result in a poorer performing training algorithm. The following results show, however, that in this task there is a significant advantage to larger bunch sizes.

All of the following plots use ESSE as an error measure. For a particular task, it might be more relevant to use a different measure such as cross-validation error or some higher level classification error. Nevertheless, we chose to use ESSE for two reasons. First, it is a general error measure unspecific to any particular application. Second, and more importantly, the back-propagation learning algorithm we tested attempts to minimize the following cost function:

$$J = \frac{1}{2}\sum_n (T_n - O_n)(T_n - O_n)^T$$

where T_n is the target and O_n is the output for the n^{th} pattern. By observing the evolution of ESSE for a given bunch size, we therefore can see how this cost function decreases over epoch presentations.[2]

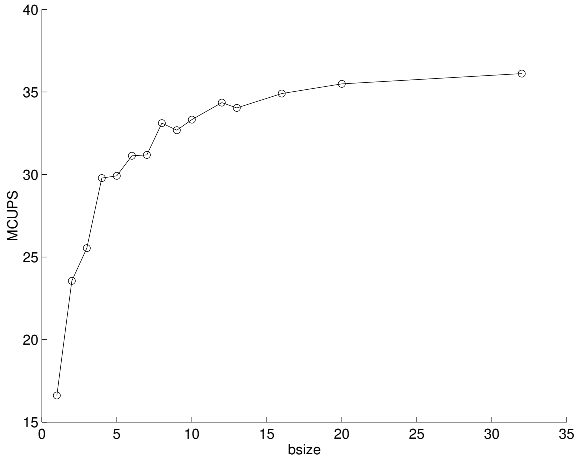

Figure 2. MCUPS vs. bunch size.

Figure 3 graphs the number of epochs required to reach a certain ESSE. The graph shows that for this task convergence is faster for smaller bunch sizes although the effect is not large. In other words, larger bunch sizes tend to take more epochs to reach a certain ESSE.

Figure 4 combines the information given in Figures 2 and 3 by plotting the *time* required to achieve a given ESSE as a function of bunch size. Observe that for all the ESSE values listed, the speed advantage of larger bunch sizes outweighs the corresponding decrease in algorithmic convergence. Therefore, this plot shows runtime advantage for using greater than unity bunch sizes.

Figure 5 graphs the speedup, or the time for unity bunch size divided by the time for each bunch size, vs. the log of ESSE. Note that in this plot, time moves forward from right to left as log ESSE gets smaller. We see that at best, a bunch size of 4 (highlighted in Figures 3 and 4) results in a 2.2 speedup over a unity bunch size.

[2] ESSE would therefore be inappropriate for other cost functions, such as cross entropy.

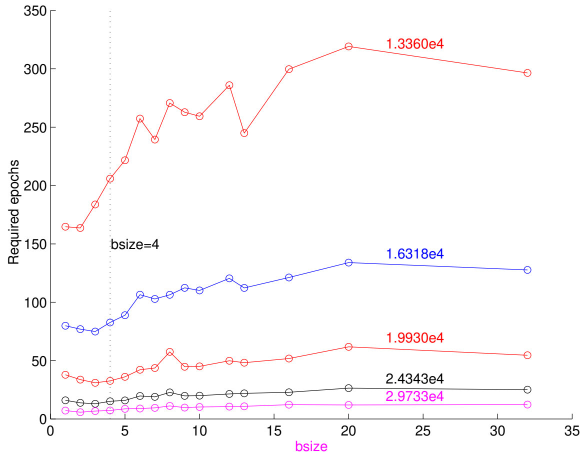

Figure 3. Epochs to reach a given ESSE vs. bunch size.

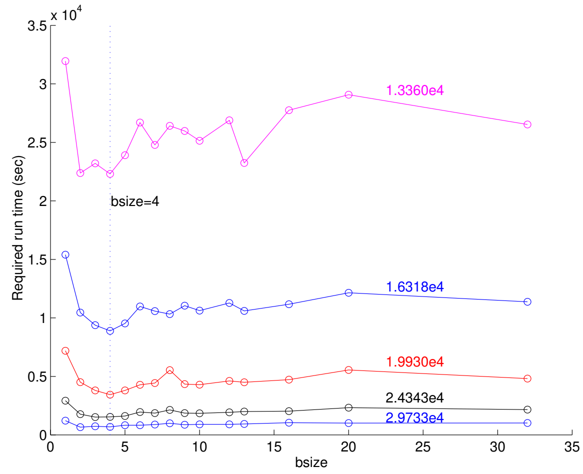

Figure 4. Time required to achieve a given ESSE vs. bunch size.

The above results show that, even in a case where the algorithmic convergence is adversely effected by increasing the bunch size, the corresponding increase in speed obtained by the faster matrix multiply routine can result in an overall reduction in the wall-clock time necessary to reach a given ESSE. Note that as n_p increases from one, the set of operations performed during a training doesn't change. In order to further decrease training time by changing the algorithmic convergence, additional techniques such as dynamic learning rate adjustment, random pattern presentation, momentum, etc [13] can be used together with the bunch-mode backpropagation algorithm. Some combination of both algorithmic techniques and the use of matrix-matrix multiply will probably result in an optimal back-propagation learning algorithm for a given task. In fact, PHiPAC matrix multiply code has been successfully integrated into a more general neural network simulation program, QuickNet, that is in common use here at ICSI for our speech recognition tasks.

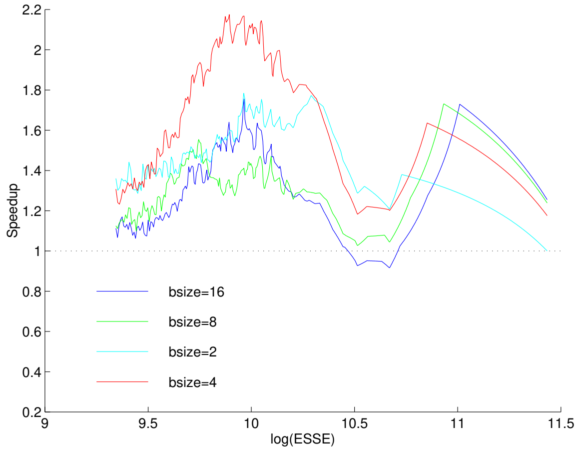

Figure 5. Speedup over unity bunch size.

4. CONCLUSIONS

We have described how the PHiPAC coding methodology can be used to produce performance portable matrix-matrix multiply routines and how these routines can be used to speed the execution of the back-propagation learning algorithm.

In the future, we plan to write search scripts for the convolution generator, write generators for more operations such as cross-correlation, FFTs, IIR filtering, and matrix-transpose, and demonstrate the viability of this technique to more general signal processing and numerical algorithms.

The PHiPAC matrix multiply generator, search scripts, and a demonstration of bunch-mode back-propagation code in C are all available from the PHiPAC WWW site [4].

REFERENCES

[1] B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: A methodology for portable high-performance. In *International Working Conference on Massively Parallel Programming Models*, 1995.

[2] D. Anguita and B. Gomes. MBP on T0: mixing floating- and fixed-point formats in BP learning. Technical Report 94-038, ICSI, 1994.

[3] D. Anguita, G. Parodi, and R. Zunino. An efficient implementation of BP on RISC-based workstations. *Neurocomputing*, 6:57–65, 1994.

[4] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. The PHiPAC WWW home page. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[5] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. LAPACK working note 111, University of Tennessee, 1996.

[6] L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.

[7] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[8] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[9] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley, 1991.

[10] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[11] J.D. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix algorithms. In R. Geist and S. Junkins, editors, *Proceedings of the 33rd Annual Southeast Conference*, pages 195–204, New York, NY, March 1995. Association for Computing Machinery, ACM.

[12] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the 10th International Parallel Processing Symposium.* IEEE Computer Society, April 15–19 1996.

[13] Dilip Sarkar. Methods to speed up error back-propagation learning algorithm. *ACM Computing Surveys*, 27(4):519–544, December 1995.

[14] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.