Is Predictable Performance Possible?

William D. Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

Programming for performance remains more an art than a science. While some progress has been made in the development of predictive models and algorithms based on those models, the best results are still often obtained by trial and error. This situation is illustrated by recent work of several groups on perhaps the simplest problem for which high floating-point performance can be expected: matrix-matrix multiply.

1 Introduction

The gap between peak and achieved performance on many important scientific applications is large; sustained performance figures of 5 to 25% are common. In some cases this reflects a limitation in the hardware (Section 2), but often it is due to the complexity of the architecture and the resulting difficulty in producing effective code. This is illustrated with the very simple case of dense matrix-matrix multiply in Section 3. The results show that it is very difficult to *predict* the performance of an algorithm; hence, application writers and algorithm developers are often reduced to a trial-and-error approach. Developers fall back on performance *understanding*; but as discussed in Section 4, even with the support of hardware counters, this is very difficult.

2 Sparse Matrix-Vector Multiply

The most efficient algorithms (in terms of floating-point operations) for the solution of large, sparse systems of linear equations are preconditioned iterative methods. A key operation in these methods is a matrix-vector multiply. In actual practice, the performance of this matrix-vector product is disappointing. We can understand this performance by using a very simple model.

Let the sparse matrix be stored in AIJ format. In this format, there is an array `A` containing the non-zero elements of the matrix, an array `J` containing the column indices of each element, and an array `I` containing information on the number of non-zero matrix entries in each row of the matrix. If there is no other structure, this is the minimum amount of information that must be stored. The code for a matrix-vector multiply then looks like

```
for (k=0;k<n;k++) {
    nrow = i[k+1] - i[k];
    sum  = 0;
    while(nrow--)
        sum += *a++ * x[*j++];
    *y++ = sum;
    }
```

Assume that the problem is large enough that it does not fit in cache. We can set an upper bound on the performance of this operation by considering only the main memory bandwidth. Here are the operations:

1. every row, fetch `ia[k+1]`
2. fetch nrow values from `j`, `a`
3. fetch every element of `x` (perfect cache)
4. perform a floating multiply-add with each element of `a`
5. every row, store `y[k]`.

Let there be m non-zeros and n rows. The total data volume is

$$\begin{array}{rl} n*(sizeof(int)+2*sizeof(double)) & (ia,x,y) \\ m*(sizeof(int)+sizeof(double)) & (ja,a) \end{array}$$

There are m floating-point multiply-adds. A typical simple problem will have $m \approx 5n$, and with sizeof(int) = 4 and sizeof(double) = 8, we have

$$\begin{aligned} \text{bytes} &= n*(4+2*8)+m*(4+8) \\ &= n*(4+2*8+5*(4+8)) \\ &= n*(4+16+5*12) \\ &= 80*n \\ &= 16*m(2 \text{ doubles per MA}) \end{aligned}$$

This really isn't too bad, except: Consider a 200 MHz processor capable of a MA (multiply-add) each cycle (roughly a SGI Origin2000) or a 100 MHz processor capable of two MAs each cycle (roughly an IBM P2SC). The computation will require 16 bytes per MA, or 3.2 GB/sec bandwidth. This is (roughly) 3–6 times what is actually available. Note that the calculation assumes that data is read only once, particularly the vector. If the vector elements, because of cache misses, must be read several times (such as another $2n$ times), the necessary bandwidth increases.

The real situation is even worse, of course. This analysis doesn't include the effects of limitations of cache associativity or the latency of the memory system. These must also be taken into account. Differences between loads and stores are also ignored.

We can reverse the calculation. Given a memory bandwidth, we can calculate the maximum number of flops that can be supported. For example, if the main memory bandwidth is 0.5 GB/s, then the maximum flop rate is 31 MAs or 62 MFlops (out of 400 peak MFlops). These numbers do, in fact, match the the observed performance on several systems.

In many ways, this is just a rediscovery of the fact that matrix-vector operations constrain the opportunities for high performance. So, for operations that are constrained by a single resource such as main memory bandwidth, performance is indeed predictable (if disappointing).

3 Dense Matrix-Matrix Multiply

Computing the product $C = A * B$ of two large, dense (all entries non-zero) matrices is a common operation that is both important to a class of scientific applications (it is the building block for both solving linear equations and finding eigenvalues) and has been heavily studied as an archetypical kernel for compiler code generation (see, e.g., [1]). In addition, it has the property that there are n^3 floating-point operations for n^2 data elements.[1] Because each data item is reused roughly n times, main memory bandwidth is no longer a performance limitation, and carefully crafted code can often reach a significant fraction of peak performance for the operation of matrix-matrix multiply. In fact, this operation is a key part of the level 3 BLAS [2] and is performed by the BLAS routine DGEMM. Computer vendors often provide carefully tuned implementations of the BLAS; DGEMM is particularly important because it is a common building block for other software.

[1]This ignores Strassen's algorithm, which lowers the floating-point operations to $n^{2.8}$.

An indication of the lack of predictability is the emergence of projects that perform a direct search over the space of possible implementations. Two such projects are ATLAS [3] and PHiPAC [4, 5]. These parameterize the basic matrix-matrix multiply algorithm, and then measure the performance of the resulting code. The results are startling: ATLAS, applied to seventeen different systems (six major processor families) outperforms the vendor version in nine of the seventeen cases (sometimes by more than 50%), and is never significantly slower [6].

Even more startling is the poor performance of the original Fortran implementation of DGEMM. On one system, the performance of the original Fortran code is a factor of eight slower than the ATLAS version [7].

These results serve as a reality check on the hope that compilers will be able to handle the complexities of modern systems in the near future. An example of the sort of analysis that a compiler might need to undertake can be found in [8]; even here, the effects of cache associativity are ignored. Clearly, the difficulty in predicting performance makes it extremely difficult for a compiler to deliver good performance on even relatively simple computational kernels. Some interesting comments on architectural features such as the conflicting shape of TLB and cache can be found in [9].

4 Testing Performance Hypothesis

Given that performance prediction is so difficult, most developers must fall back on performance *understanding.* To aid in this, many systems now provide access to hardware counters that provide counts for cache misses, floating-point instructions, and so on. Unfortunately, these are often of only limited use.

The key difficulty with hardware counters is the difficulty in relating them to the user's application. Put another way, a performance measurement is an experiment that tests a hypothesis about the source of a performance problem. Unfortunately, the hypotheses that the user may form are not easily answered by simple hardware counts. Some example hypotheses are:

1. Cache thrashing caused by too many variables for the amount of cache associativity

2. Extraneous loads caused by conservative variable loading by the compiler

3. Specific important branches mispredicted

4. Specific variables not cached effectively (e.g., `x` in the matrix-vector multiply example)

The practice of some vendors of not providing assembly code listings at high optimization levels also eliminates a kind of experiment: did the compiler generate code that was as tight as the user expected?

Because of the lack of integration between the language (and compiler) and the hardware counters, it is difficult for users to construct tests that unambiguously test a performance hypothesis.

5 Conclusions

Predicting or even explaining performance of real applications on many current systems is difficult unless performance is limited by a single resource such as main memory bandwidth or instruction issue rate. The difficulty in generating efficient code for matrix-matrix multiply shows that the complexity of current systems exceeds the ability of production compilers to handle that complexity. A more integrated, system-oriented approach to performance predictability is needed.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] 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*, Apr. 91.

[2] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, "A set of level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 1–17, Mar. 1990.

[3] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software." http://www.netlib.org.

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

[5] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C. 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] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software." `http://www.cs.utk.edu/~rwhaley/ATL`, 1998.

[7] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software, figure 3." http://www.netlib.org.

[8] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante, "Quantifying the multi-level nature of tiling interactions," *International Journal of Parallel Programming*, 1998.

[9] N. Mitchell, L. C. N., and J. Ferrante, "A compiler perspective on architectural evolutions," Feb. 1997. http://www.cs.ucsd.edu/users/carter/Papers/icca97.ps.gz; Workshop on Interactions between Compilers and Computer Architectures.