Automatic benchmarking and optimization of codes: an experience with numerical kernels*

José R. Herrero, Juan J. Navarro
Computer Architecture Department, Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Mòdul D6, E-08034 Barcelona, (Spain)
{josepr,juanjo}@ac.upc.es

Abstract *New algorithms are constantly developed in search of better or faster results. Many variants of code are often tried while searching for the best solution. When the number of code variants or possible input parameters is very high, the process of benchmarking and analysis of results can become cumbersome and error prone. We present a tool for automatic benchmarking which manages a database of possible parameters and the results obtained for them. This tool can automatically choose the optimum code and create a target library. Using it we have generated a library specialized in the operation on very small matrices which is useful in multimedia codes.*

Keywords: Automatic benchmarking, tuning, performance, small matrices

1 Automatic benchmarking: motivation

Some times code developers need to compare multiple variants of a program. Each variant can be chosen either at compilation time or at execution time. Conditional compilation can be done in case a particular variant amongst several has to be chosen at compilation time. Conditional compilation is performed by means of preprocessor directives. Depending on whether a symbol is defined or not, or the value it has at compilation time, certain parts of code are included or excluded, or take one value or another. In order to know the exact variant of code we are using we need to keep track of all parameters specified at compilation time. Similarly, when a program is executed it can often have multiple input parameters.

Consequently, the results of the execution of a program should be kept associated to all input parameters, i.e. to all parameters used at compilation time to generate the object plus information on all inputs used at execution time. Unless we keep all this information we will not be able to do accurate studies with those results. A result can become useless if we cannot know exactly how it was obtained.

As the number of parameters grows, the number of combinations tested can become very large. Keeping all necessary data for each combination can be tedious work and error prone. Thus, it is desirable and advisable to automate this process.

2 Automatic performance optimization of libraries

In this paper we present a tool we have developed and how it has been used to optimize a library of routines specialized in operating on small matrices. We call this tool BMT, which stands for *BenchMarking Tool*.

We only know of one tool which has some similarity with BMT. We refer to a commercial product called ST-ORM [1]. This tool allows

*This work was supported by the Ministerio de Ciencia y Tecnología of Spain and the EU FEDER funds (TIC2001-0995-C02-01)

http://people.ac.upc.edu/josepr/

for stochastic analysis. It can be used to launch executions. It maintains a database of results and offers statistical and graphical treatment of these results. However, as far as we know, it does not offer the possibility to generate optimized libraries.

2.1 Features of BMT at a glance

BMT is a program written in Perl that can launch compilation of programs as well as its posterior execution. In each case, BMT prepares the parameters to be used, analyzes the compilation and execution processes searching for errors, and keeps the results in a database. Statistics and plots can be obtained. The tool can also inform about the optimal combination of parameters amongst a set of combinations. Using this information it can produce a library of optimal routines.

The user can specify a set of benchmarks to run. For each benchmark one can specify parameters for compilation, including definition of preprocessor symbols, and parameters for execution. For each parameter the user can specify a set of values to use.

2.2 Important aspects of automatic benchmarking

Handling a variable number of parameters

When several parameters have to be used, we need a way to generate all their combinations. The common way to do that is to define several nested loops. Each loop is associated to one parameter. Then, the induction variable in each loop gets its values from the list of values that its associated parameter can have.

In general, we need a number of nested loops that equals the number of parameters to use. However, a general benchmarking application must be able to deal with any number of parameters: the number of parameters used for one benchmark can be different from that used for another benchmark. Thus, we cannot hardcode a particular number of nested loops in the benchmarking tool since that would limit its applicability. Instead, we simulate any number of loops. We generate all combinations of the input parameters, one by one, with a single loop. For each parameter, we keep information about the next value that must be used from its list of values in generating the next combination.

Checking the correctness of an execution

When hundreds or thousands of executions are launched the possibility of getting errors increases. Some of them can be transient errors. Others can be permanent (real) errors. For instance, if a process is killed externally, a disk quota is exceeded or a machine crashes, the execution finishes abruptly. However, launching the process a second time can produce correct results. In this case we have a transient error.

If, for instance, a routine expects a positive integer as input and we feed it with a negative integer we will repeatedly get an incorrect execution. In this case we should not keep trying but, instead, keep track of the incorrectness of the parameter combination tried as input. In this way we could avoid future retries.

Enforcing robustness of programs

We want to be able to check that a program worked correctly and all the parameters it used in its execution were those we expect. To make sure that the process ended as expected we have it to output a given string just before it finishes execution, e.g. *End of execution.* Thus, we can check whether this string appears in the output of the execution. In this way we can detect unexpected termination of programs. When such an error is detected we can decide whether to launch the execution again or not.

In order to make sure that a program is using the very same parameters as we expect it to use, we force it to output all the parameters that affect its execution. Both parameters used at compilation time and parameters used at execution time. For each parameter we have it to write the parameter name and its value.

Then, after an execution is completed we can parse its output and check all parameter-value pairs. If all of them match the current parameter combination then we consider the execution correct and accept to keep its results in the database. Otherwise those results are discarded and an error flagged.

Dealing with concurrency

We believe it is important to allow for concurrent compilation and execution of programs. Concurrency affects the way we handle compilations, executions and accesses to the database.

We use a different (temporary) directory for each compilation. Thus, several compilations can proceed at once. Once we obtain the executable we rename it with a name that includes all parameters used at compilation time and store it in a directory where we store all executable files handled by BMT. When each execution is launched we redirect the standard output and standard error to a file whose name starts with the name of the executable followed by all the input parameters used for its execution. Finally, the output file is analyzed looking for the information we want to store in the database of results. When the database is managed we ensure mutual exclusion by using the usual lock procedures.

3 An application: Generation of a Small Matrix Library

This section shows how efficient codes operating on very small matrices (that fit in the first level – L1 – cache) can be produced. Different matrix sizes or target platforms may require different codes to obtain good performance. We write a set of codes for each matrix operation using different loop orders and unroll factors. Then, for each matrix size, we automatically compile each code fixing matrix leading dimensions and loop sizes, run the resulting executable and keep its Mflops. The combination producing the highest number of Mflops is then used to produce the object introduced in a library. Thus, a routine for each desired matrix size is available from the library.

Using routines in our library can improve the performance of codes which perform a large number of operations on small matrices. Examples of such programs are sparse matrix and multimedia codes (like 3D graphics engines, signal processing or video encoding/decoding algorithms).

3.1 Generation of efficient code

Creation of efficient code has traditionally been done manually using assembly language and based on a great knowledge of the target architecture. Such an approach, however cannot be easily undertaken for many target architectures and algorithms. Alternatively, codes specially optimized for a particular target computer can be written in a high level language [2, 3]. This approach avoids the use of the assembly language but keeps the difficulty of manually tuning the code. It still requires a deep knowledge of the target architecture and produces a code that, although portable, will rarely be efficient on a different platform.

A cheaper approach relies on the quality of code produced by current compilers. The resulting code is usually less efficient than that written manually by an expert. However, its performance can still be extremely good and some times it can even yield better code. We have taken this approach for creating our Small Matrix Library (SML). For each desired operation, we have written a set of codes in Fortran. For instance, for a matrix multiplication we have codes with different loop orders (*kji, ijk*, etc.) and unroll factors. Using *BMT*, we compile each of them using the native compiler trying several optimization options. For each resulting executable, we automatically execute it and register its performance. These results are kept in a database and finally employed to produce a library using the best combination of parameters.

By fixing the leading dimensions of matrices and the loop trip counts we have managed to

obtain very efficient codes for matrix multiplication on small matrices. Since several parameters are fixed at compilation time the resulting object code is only useful for matrix operations using these fixed values. Actual parameters of these routines are limited to the initial addresses of the matrices involved in the operation performed. Thus, there is one routine for each matrix size. Each one has its own name in the library.

We also tried feedback driven compilation using the Alpha native compiler but performance either remained the same or even decreased slightly. We conclude that, as long as a good compiler is available, fixing leading dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels.

3.2 Results

In this section we present one of the operations implemented in our library: the matrix multiplication. This operation is frequently used as internal kernel in numerical algorithms. Table 1a shows the algorithms tried for the $C = A \times B^T$ matrix multiplication operation where $A(i \times k)$, $B(j \times k)$ and $C(i \times j)$. For simplicity, each algorithm is coded with a number. The notation for the order of the loops (forms) was introduced in [4]. kji means that loop k (direction k in the iteration space) is specified as the outer loop while loop i is found in the inner loop. $4\tilde{i}$ means that a loop with 4 iterations in direction i has been fully unrolled. *BCreg* means values in matrices B and C are kept in local variables in an attempt to improve register reuse. $i(\ldots)$ means tiling is done in the i dimension.

Table 1b shows the best algorithm found for the matrix multiplication operation *mxmts_fix* for several matrix sizes on the Alpha 21164 and R10000 processors. Matrix sizes are expressed as *i_j_k*. There is a large variation in the optimal algorithm chosen. 7 out of 11 algorithms resulted to be the best for some particular combination of sizes and target platform. Choosing a single algorithm for all cases would result in an important performance loss.

Code of Algorithm	form
1	*jik_Creg*
2	*jik*
3	*kji_Breg*
4	$i(jk4\tilde{i}$*_BCreg*$)$
5	$i(jk4\tilde{i}$*_Breg*$)$
6	$i(jk4\tilde{i})$
7	*ijk*
8	*kji*
9	*jki*
10	$jik4\tilde{k}$*_Creg*
11	$jik8\tilde{k}$*_Creg*

a)

Matrix sizes	*Alpha 21164*	*R10000*
4_4_4	2	8
4_4_8	3	5
4_4_16	3	3
4_4_32	3	3
8_8_8	10	6
8_8_16	11	8
8_8_32	11	5
16_16_16	11	2
16_16_32	11	5
32_32_32	11	6

b)

Table 1: **a)** List of algorithms **b)** Best algorithm found for the matrix multiplication operation *mxmts_fix* on an Alpha 21164 and an R10000.

Figure 1a shows the performance of different routines for matrix multiplication for several matrix sizes on an Alpha-21164. The matrix multiplication performed in all routines benchmarked uses: the first matrix without transposition (n); the second matrix transposed (t); and subtracts the result from the destination matrix (t). Thus, we call the BLAS[1] routine *dgemm_nts*.

The vendor BLAS routine *dgemm_nts* yields very poor performance for very small matrices getting better results as matrix dimensions

[1] BLAS stands for Basic Linear Algebra Subroutines.

grow towards a size that fills the L1 cache (8 Kbytes for the Alpha-21164). This is due to the overhead of passing a large number of parameters, checking for their correctness, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when the operation is performed on large matrices. However, it is notable when small matrices are multiplied. Also, since its code is prepared to deal with large matrices, further overhead can appear in the inner code by the use of techniques like strip mining.

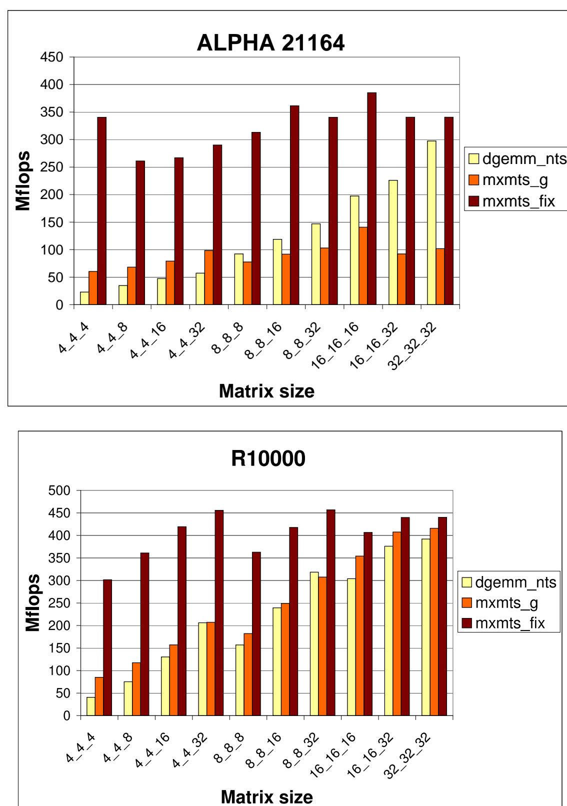

Figure 1: Comparison of performance of different routines for several matrix sizes: **a)** on an Alpha **b)** on an R10000.

A simple matrix multiplication routine *mxmts_g* which avoids any parameter checking and scaling of matrices can outperform the BLAS for very small matrix sizes. Finally, our matrix multiplication code *mxmts_fix* with leading dimensions and loop limits fixed at compilation time gets excellent performance for all block sizes ranging from 4x4 to 32x32. The latter is the maximum value that allows for a good use of the L1 cache on the Alpha unless tiling techniques are used.

Figure 1b shows the performance of different routines for several matrix sizes on the R10000 processor. Results are similar to those of the Alpha with the only difference that the *mxmts_g* performs very well. This is due to the ability of the MIPSpro F77 compiler to produce software pipelined code, while the Alpha compiler hardly ever manages to do so.

Though our codes were adapted to the underlying architecture, they were written in a high level programming language (FORTRAN).

3.3 The need for a benchmarking tool

Automation of the previous process is absolutely necessary due to the large amount of combinations tested and data generated. For example, we tried over 400 combinations of algorithms and compiler optimization flags just for the matrix multiplication operation *mxmts_fix* on the Alpha for the sizes reported in this paper. When we look at all the operations included in the library the number of combinations tried amounts to several thousands. Using *BMT* we could obtain an efficient Small Matrix Library (SML) for Alpha, R10000 and Intel Pentium based systems. Generating SML in other platforms should be straightforward.

3.4 Related work

Iterative compilation [5] consists in a repetitive compilation of code using different parameters. Program transformations like loop tiling and loop unrolling are very effective techniques to exploit locality and expose instruction level parallelism. The authors claim that finding the optimal combination of tile size and unroll factor is difficult and machine dependent. Thus, they propose an optimization approach based on the creation of several versions of a program and decide upon the best by actually executing

them and measuring their execution time. Our approach for obtaining high performance code is similar with the difference that, while they apply a set of transformations, we use simple codes and let the compiler do its best.

Several projects were targeted at producing efficient BLAS routines through automatic tuning [6, 7, 8]. The difference with our work is that they are not focused on operations on small matrices.

4 Conclusions

There are a number of applications (like sparse matrix and multimedia codes) which perform many operations on small matrices. The performance of routines which operate on them is critical for the overall performance obtained by those applications. Since no single algorithm was the best for all matrix sizes in any platform, we conclude that an exhaustive search is necessary to get the best one for each matrix size. The large number of possibilities tested widens the possibility of getting good performance. At the same time this makes the optimization process time consuming and error prone. For this reason we have implemented a *BenchMarking Tool* (BMT) which automates this process.

BMT can control the compilation of multiple variants of code as well as the execution of each variant for a set of input parameters. Results are stored in a database and can be used to determine the optimal combination of parameters. These parameters are used to obtain the best possible routine. This routine is finally introduced in a library.

Using BMT we have generated a Small Matrix Library (SML) on several systems obtaining very efficient codes specialized on operations on small matrices. These routines outperform the vendor's BLAS routine for small matrix sizes. We have shown that fixing dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels.

References

[1] ST-ORM User's Manual, 1999.

[2] C. Kamath, R. Ho, and D.P. Manley. DXML: A high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, 1994.

[3] Juan J. Navarro, E. García, and José R. Herrero. Data prefetching and multilevel blocking for linear algebra operations. In *Proceedings of the 10th international conference on Supercomputing*, pages 109–116. ACM Press, May 1996.

[4] Juan J. Navarro, Antonio Juan, and Tomas Lang. MOB forms: A class of Multilevel Block Algorithms for dense linear algebra operations. In *Proceedings of the 8th international conference on Supercomputing*. ACM Press, 1994.

[5] T. Kisuki, P.M.W. Knijnenburg, and M.F.P O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

[6] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM Press, 1997.

[7] J. Cuenca, D. Gimenez, and J. Gonzalez. Towards the design of an automatically tuned linear algebra library. In *Proceedings. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 201–208, 2002.

[8] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, pages 211–217. IEEE Computer Society, Nov 1998.