Fractal Matrix Multiplication: a Case Study on Portability of Cache Performance

Gianfranco Bilardi[1], Paolo D'Alberto[2], and Alex Nicolau[2]

[1] Dipartimento di Elettronica e Informatica, Università di Padova, Italy.
bilardi@dei.unipd.it ***

[2] Information and Computer Science, University of California at Irvine
{paolo,nicolau}@ics.uci.edu †

Tech Report #00-21

Fractal Matrix Multiplication: a Case Study on Portability of Cache Performance

Gianfranco Bilardi[1], Paolo D'Alberto[2], and Alex Nicolau[2]

[1] Dipartimento di Elettronica e Informatica, Università di Padova, Italy.
bilardi@dei.unipd.it ***
[2] Information and Computer Science, University of California at Irvine
{paolo,nicolau}@ics.uci.edu †

Tech Report #00-21

Abstract. In this paper we demonstrate the practical portability of a simple version of matrix multiplication designed to exploit maximal and predictable locality at all levels of the memory hierarchy, with no *a priori* knowledge of the specific organization of the memory system for any particular machine. We show that memory hierarchies portability does not sacrifice floating point performance, which is always a significant fraction of peak and, at least on one machine, is higher than ATLAS and vendor multiplication.
We present a proof of concept of the fact that the theoretical conclusions on locality exploitation yield practical implementations with the desired properties.

1 Introduction

The ratio between main memory access time and processor cycle in microprocessor systems has been continuously increasing, up to values of a few hundreds nowadays. The increase in ILP has been a significant feature in this; considering that current CPUs can typically issue four to six instructions per cycle, the cost of a memory access in term of performance is today an increasingly high toll on overall performance of super-scalar/VLIW processors. The architectural response to this state of affairs has been an increase in the size and number of caches, with a second level being available on most machines, and a third level becoming now popular. However, the memory hierarchy is of no help to performance unless the computation exhibits a sufficient amount of locality (code/data). Unfortunately, the amount of locality that is naturally found in code is generally no longer sufficient to ensure good performance; algorithm design and compiler optimization need to explicitly take locality into account. A number of studies have begun to explore these issues. An early paper by Aggarwal, Alpern, Chandra, and Snir [1] introduced the Hierarchical Memory Model (HMM) of computation, as a basis to design and evaluate memory efficient algorithms. This approach has been developed and extended in a number of papers

*** This work was supported, in part, by CNR and MURST of Italy
† Supported by AMRM DABT63-98-C-0045.

(see, e.g., [2], [32], [26],[3]). Compiling for locality has also received considerable attention (see, e.g., [38], [39], [21], [4], [9], [29]).

Besides being very burdensome and labor intensive for the user, if it is done by hand, algorithm optimization for increased locality has often led to solutions that are highly dependent upon the parameters of the specific memory system, such as the number, the size, and the speed of the cache levels. In addition to introducing a non trivial source of complexity, this dependence makes the performance of the resulting code non optimally portable across different platforms. This paper addresses the question whether it is possible to develop a code which, unchanged, runs with (near) optimal performance on a wide class of machines. Of course, if the answer to this question is negative, it is worthwhile to develop a parameterized code which can be tuned to the machine by a suitable adjustment of the parameters. However, there are several reasons to investigate the feasibility of fixed, portable code. A first reason is the simplicity of the solution, when achievable. A second reason is its robustness: in time shared environments, the amount of cache space effectively available to a process varies dynamically; a code whose performance is not predicated on careful matching of program and architectural parameters is more likely to be robust. A third reason is of a more methodological nature: by exposing the mechanisms that imply lack of portability in a fixed code, we can obtain valuable insights as to what kind of space the code parameters should span to adapt to the hierarchy.

In this paper we demonstrate the practical portability of a simple version of matrix multiplication designed explicitly to ensure maximum of locality at all levels of the memory hierarchy, with no *a priori* knowledge of the specific organization of the memory system for any particular machine. It is our contention that such reworking of the code is, and will become, increasingly critical for execution on modern ILP, and even more so for massively parallel machine. It is our ultimate belief that such reworking of the code to expose/enhance locality can be done automatically under compiler control, with some sacrifices in optimality. However this is beyond the scope of this paper, which present a proof of concept of the fact that the theoretical conclusions on locality exploitation yield practical implementations with the desired properties. There are indeed various potential sources for discrepancy between theory and practice, especially for current (ILP) machines. One is the asymptotic nature of the theoretical analysis; in contrast, we are interested in comparing the portable performance of our approach against that of highly tuned algorithms, where tuning on actual machines is usually responsible for performance between 50% and 90% of peak. Other sources of discrepancy derive from the differences between real super-scalar/VLIW and the model of computation; for example, the theoretical models do not fully incorporate the effects of cache line length and of the limited degree of associativity of real caches. An additional goal is to get insights on the interactions between cache behavior and processor behavior, especially register allocation and vertical/horizontal parallelism of instruction scheduling.

We follow an approach to matrix multiplication which we call the *fractal* approach, since it aims at exposing locality at all temporal scales. The approach carefully combines a number of known ideas and techniques as well as some novel ones, leading to the following key results:

1. We show that a particular version of matrix multiplication can indeed be implemented on modern ILP machines so as to achieve excellent, portable cache performance, incurring a low number of misses at the various levels of caches on a set of 7 different machines.
2. We then show that the overall running time is also very good in practice, compared to the lower bound implied by peak performance and to the running time of the best known code (Automatically Tuned Linear Algebra Software, ATLAS, [36]). This indicates that the strategy adopted does not sacrifice overall performance.
3. While the main motivation to develop the fractal approach was provided by the goal of portability, at least on some machines such as the R5000 IP32, the fractal approach yields the fastest known algorithms.

2 Related Work

A few studies have investigated issues related to portability of performance across memory hierarchies. In [1], asymptotically optimal implementations are proposed for a number of algorithms for matrix multiplication, FFT, and sorting on the HMM model of computation. In this model, the time to access a location x is a function $f(x)$; the authors observe that optimality of their algorithms is achieved for a wide family of functions f. More recently, similar results have been obtained for a different model, with automatically managed caches [20]. The optimality is established by deriving a lower bound to the access complexity $Q(S)$, i.e., to the number of accesses that necessarily miss any given set of S memory locations. Lower bounds techniques were pioneered in [26] and recently extended in [5, 7]. The question whether arbitrary computations admit optimally portable implementations has been investigated in [6] where it is shown that the answer is generally negative, by exhibiting computations such that no schedule for the operations can be simultaneously optimal for all memory hierarchies of a given class. Nevertheless, it appears of interest to further investigate those computations that admit portable implementations, particularly so since they are likely to include relevant classes such as linear algebra kernels.

Matrix multiplication is a key kernel in linear algebra, and near optimal performance can be achieved by a tuned routine ([27], [28]). A sequence of studies have aimed at reducing the number of operations from the straightforward $2n^3$, e.g., to $O(n^{\log_2 7})$ [33], or to $O(n^{2.376})$ [12] (although the latter is not actually used by any library). Wide attention has been given to the impact on performance of several issues: number of instructions, data layout, data locality, latency hiding, register allocation, instruction scheduling and instruction parallelism, (e.g., [36], [8], [14], [15], [16]).

The most common data locality optimization performed on ijk-loop algorithm is loop tiling ([29], [31], [38], [30], [40]). Loop tiling increases time locality, hence reducing the so called *capacity misses.* But tile sizes are machine dependent parameters, based on the cache sizes and on any technique to reduce *self interference* ([21], [15]). Vendor libraries exploit their knowledge of the platform and determine tile sizes, scheduling instruction and other optimizations. Automatically tuned packages (see [36] and [8] for matrix multiplication and [19] for FFT) measure machine parameters by interactive tests and then produce machine tuned code, generally with better performance than the vendor libraries. This approach achieves *portability* at the level of the package, rather than of the actual application code. It is quite promising, although it requires considerable development effort and some installation effort on each platform.

Another approach, called *auto-blocking,* has the potential to yield portable performance for the individual code. Informally, one can think of tiles whose size is not determined by any *a priori* information but arises automatically from a recursive decomposition of the problem. This approach has been advocated in [23], with applications to LAPACK, and its asymptotic optimality is discussed in [20]. Our fractal algorithms belong to this framework. Recursion-based algorithms often exploit various features of non standard layouts, *recursive layouts* ([11], [10], [35], [18], [37], [24] and [17]). Conversion from and to standard (i.e., row-major and column-major) layouts introduces $O(n^2)$ overheads, usually negligible, except for matrices small enough for the n^2/n^3 ratio to be insignificant, or large enough to require disk access. Recursive algorithms are often based on power of two matrixes (with padding, overlapping, or peeling) because of closure properties of the decomposition and a simple index computation. In this paper, we use a non padded layout for arbitrary square matrices, thus saving space and maintaining the conceptual simplicity of the algorithm, while developing an approach to burst the recursion and save index computations. Register allocation and instruction scheduling are still bottlenecks. A small number of loads and stores reduces the traffic from/to the cache and latency hiding of loads and stores avoid to stall deep pipelined CPUs ([15], [36]). Currently, the practical way tiling/blocking is combined with other optimizations is function of the level of memory hierarchy which the code is written for. But for recursive algorithms no compiler is so smart to perform unfolding of last calls (*leaves*) and performs optimizations on the code.

The algorithms proposed in this paper *do not* compromise numerical stability. Indeed, for machines without extended precision accumulator and with register file, the order of the computation should not affect the worst case error estimation (Lemma 2.4.1 [22] or Lemma 3.4 [25]).

3 Fractal Layout of Matrices

In this section, we propose a recursive approach to embed a two-dimensional array A into a one-dimensional array $\mathbf{a}$. Any $m \times n$ matrix A is composed of four ordered-by-row sub-arrays, A_0, A_1, A_2 and A_3. The composition is defined *balanced* and $A_0 = \{a_{ij} : 0 \leq i < \lceil m/2 \rceil, 0 \leq j < \lceil n/2 \rceil\}$, $A_1 = \{a_{ij} : 0 \leq i < \lceil m/2 \rceil, \lceil n/2 \rceil \leq j < n\}$, $A_2 = \{a_{ij} : \lceil m/2 \rceil \leq i < m, 0 \leq j < \lceil n/2 \rceil\}$ and $A_3 =$

$\{a_{ij} : \lceil m/2 \rceil \leq i < m, \lceil n/2 \rceil \leq j < n\}$. The logical order of the sub-blocks is

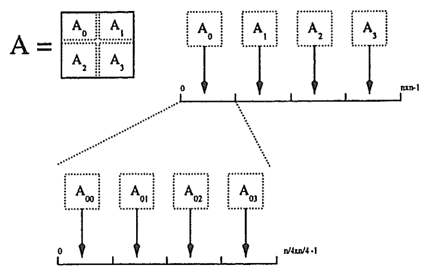

Fig. 1. Partition of a matrix and recursive definition of its fractal layout.

respected by the layout: A_0 is in $\mathbf{a}[h]$ with $h \in [0, \lceil m/2 \rceil \lceil n/2 \rceil - 1]$, A_1 is in $\mathbf{a}[h]$ with $h \in [\lceil m/2 \rceil \lceil n/2 \rceil, n\lceil m/2 \rceil - 1]$, A_2 is in $\mathbf{a}[h]$ with $h \in [n\lceil m/2 \rceil, n\lceil m/2 \rceil + \lfloor m/2 \rfloor \lceil n/2 \rceil - 1]$ and A_3 is in $\mathbf{a}[h]$ with $h \in [n\lceil m/2 \rceil + \lfloor m/2 \rfloor \lceil n/2 \rceil, mn - 1]$. Each A_i is decomposed and stored recursively. When $m = 1$ or $n = 1$ there is no further decomposition. A $m \times n$ matrix is said *near square* when $|n - m| \leq 1$. If A is a near-square matrix, so are the blocks A_0, A_1, A_2, and A_3 of its balanced decomposition. Indeed, a straightforward case analysis ($m = n-1, n, n+1$ and m even or odd) shows that, if $|n - m| \leq 1$ and $S = \{\lfloor m/2 \rfloor, \lceil m/2 \rceil, \lfloor n/2 \rfloor, \lceil n/2 \rceil\}$, then $\max(S) - \min(S) \leq 1$.

4 Fractal Algorithms for Matrix Multiplication

In this section, we introduce a class of procedures for matrix multiplication, dubbed fractal algorithms, all variants of a common scheme. In fact, we target the slightly more general operation of matrix multiply-and-add (MADD) $C = C + AB$, also denoted $C+ = AB$. Let A, B, and C be matrices of respective sizes $m \times n$, $n \times p$, and $m \times p$. The *fractal scheme* to perform the matrix multiply-and-add $C+ = AB$ is recursively defined as follows:

fractal(A, B, C)

1. If $|A| = |B| = 1$, then $C = C + A * B$ (all matrices being scalar).
2. Else, execute - in any serial order - the calls **fractal**(A', B', C') for

$$(A', B', C') \in \{(A_0, B_0, C_0), (A_1, B_2, C_0), (A_0, B_1, C_1), (A_1, B_3, C_1),$$

$$(A_2, B_0, C_2), (A_3, B_2, C_2), (A_2, B_1, C_3), (A_3, B_3, C_3)\}.$$

Of particular interest, from the perspective of temporal locality, are those orderings where there is always a sub-matrix in common between consecutive calls, which increases data reuse.

The problem of finding such orderings can be formulated by defining an undirected graph whose vertices correspond to the 8 recursive calls in the fractal scheme, and whose edges join calls that share exactly one sub-matrix (observe that no two calls share more than one sub-matrix). This graph is easily recognized to be a 3D binary cube. An ordering that maximizes data reuse corresponds to an Hamiltonian path in this cube (See Fig. 2). Even when restricting our attention to Hamiltonian orderings, there are many possibilities. The exact performance of each of them depends on the specific structure and policy of the machine cache(s) in a way too complex to evaluate analytically and too time consuming to evaluate experimentally. In this paper, we shall focus on two Hamiltonian orderings, one reducing write misses and the other one reducing read misses, which ought to be representative of the entire class.

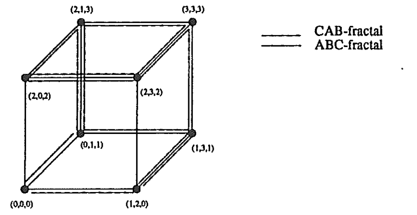

Fig. 2. The cube of calls of the fractal scheme: the Hamiltonian path defining CAB-fractal and ABC-fractal.

We call **CAB-fractal** the algorithm obtained from the fractal scheme when the recursive calls are executed in the following order: (A_0,B_0,C_0), (A_1,B_2,C_0), (A_1,B_3,C_1), (A_0, B_1, C_1), (A_2, B_1, C_3), (A_3, B_3, C_3), (A_3,B_2,C_2), (A_2, B_0, C_2). The label "CAB" underlines the fact that sub-matrix sharing between consecutive calls is maximum for C (4 cases), medium for A (2 cases), and minimum for B (1 case). It is reasonable to expect that CAB-fractal will tend to better reduce write misses, since C is the matrix being written.

In a similar vein, but with a stress on reducing read misses, we consider the algorithm **ABC-fractal** obtained from the fractal scheme when the recursive calls are executed in the following order: (A_0, B_0, C_0), (A_0, B_1, C_1), (A_2, B_1, C_3), (A_2, B_0, C_2), (A_3, B_2, C_2), (A_3, B_3, C_3), (A_1, B_3, C_1), (A_1, B_2, C_0).

4.1 Cache Performance

Fractal multiplication algorithms can be implemented with respect to any memory layout of the matrices. For an ideal fully associative cache with least recently used replacement policy and with cache lines holding exactly one matrix entry, the layout is immaterial to performance. The key property of the fractal approach is that it makes good use of the cache, irrespective of its size s, measured in matrix entries. To estimate performance, let us focus on the highest level of recursion such that all three matrix blocks being processed by calls at that level fit in cache simultaneously. Approximately, such blocks will be of size $s/3$, will cause s misses while being loaded in cache, and their entries will participate in $(\sqrt{s/3})^3 = s\sqrt{s}/3\sqrt{3}$ scalar madds, leading to an estimate of $\mu = (3\sqrt{3}(/(2\sqrt{s}) \approx 2.6/\sqrt{s}$ misses per flop. (This is within a constant factor of optimal, as a consequence of Corollary 6.2 of [26].)

For a real machine, the above analysis needs to be refined, keeping into account the effects of cache-line length ℓ (in matrix entries) and of a typically low degree of associativity. Here, the fractal layout, which stores relevant matrix blocks in contiguous memory locations, takes full advantage of cache-line effects and avoids self interference (for blocks that fit in cache, as those considered in the above analysis), even in direct mapped caches. The misses per flop can then be estimated at $\mu = 2.6\gamma/\ell\sqrt{s}$, where the factor γ accounts for cross interference between different matrices and other fine effects not captured by our analysis. In general, for a given fractal algorithm, γ will depend on matrix size (n), on the relative positions of the fractal arrays in memory, and on the degree of associativity of the cache. When interference is negligible, we can expect $\gamma \approx 1$.

On recent machines, typical values for a first-level data cache could be $s = 2^{12}$ and $\ell = 4$ double-precision words (32KB and 32B, respectively), leading to $\mu_1 \approx \gamma_1 0.01$. For a 2MB second-level cache with a 32B line, we have $\mu_2 \approx \gamma_2 0.0013$. With a penalty of 15 cycles for missing at the first level and a penalty of 100 cycles for missing at the second level, and assuming $\gamma_1 = \gamma_2 = 1$,we get an estimate of 0.15+0.13=0.28 cycles per flop due to cache misses. For a machine capable of one flop per cycle and which stalls on misses, performance would be at most 78% of FP peak, even in the absence of any other loss. In general, performance will suffer more when more FP units are available and will suffer less when the memory system is pipelined and several misses can be processed at the same time.

4.2 The Structure of the Call Tree

When pursuing efficient implementations of fractal algorithms, we are faced with the issue of managing the recursion, both to reduce its overheads and to establish a framework for good register utilization. With this goal, in this section, we study the structure of the call tree, exposing some very useful properties.

Definition 1. *Given a fractal algorithm $\mathcal{A}$, its* call tree *$T = (V, E)$ with respect to input (A, B, C) is an ordered, rooted tree defined as follows. V contains one node for each call. The root of T corresponds to the main call* **fractal***(A,B,C). The ordered children $v_1, v_2, \ldots, v_8$ of a internal node v correspond to the calls made by v in order of execution. A leaf has no children.*

If A is $m \times n$ and B is $n \times p$, we shall say that the input is of *type* $< m, n, p >$. If one among m, n, and p is zero, then we shall say that the type is *empty* and use also the notation $< \emptyset >$. It is clear that the structure of T is uniquely determined by type of the root. We will focus on square matrices, i.e., on inputs of type $< n, n, n >$. For such inputs, it is easy to see that the tree has depth $\lceil \log n \rceil + 1$ and $8^{\lceil \log n \rceil}$ leaves, n^3 of which have type $< 1, 1, 1 >$ and correspond (from left to right) to the n^3 madds executed by the algorithm. The remaining leaves have an empty type. Internal nodes are essentially responsible for performing the problem decomposition, their specific computation depending on the way matrices are represented. An internal node has eight children, typically non empty, except when its type has one or two components equal to 1, e.g., $< 2, 1, 1 >$ or $< 2, 2, 1 >$, in which case the non empty children are 2 and 4, respectively. While the call tree has about n^3 nodes, most of them have the same type. To deal with

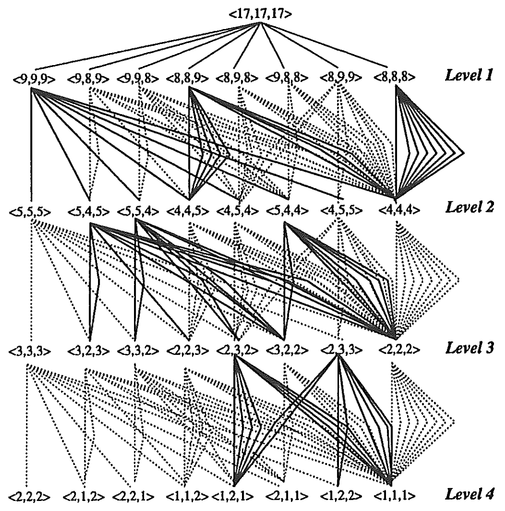

Fig. 3. Call-type dag for Matrix Multiplication $< 17, 17, 17 >$

this issue systematically, we introduce the concept of type dag, representing the set of types that arise from a given input size, together with the types of the children. Given a fractal algorithm $\mathcal{A}$, an input type $< m, n, p >$, and the corresponding call tree $T = (V, E)$, the *call type dag* $D = (U, F)$ is a directed acyclic graph, where the arcs with the same source are ordered, such that: 1) U contains exactly one node for each type occurring in T, the node corresponding to $< m, n, p >$ is called the *root* of D; 2) F contains, for each $u \in U$, the ordered set of arcs $(u, w_1), \ldots, (u, w_8)$, where $w_1, \ldots, w_8$ are the types of the (ordered) children of any node in T with type u.

Next, we study the size of the call-type dag D for the case of square matrix multiplication. It turns out that this size grows only logarithmically with the matrix size, opening interesting avenues for efficient implementation/removal of recursion. We begin by showing that there are at most 8 types of input for the calls of a given level of recursion.

Proposition 1. *For any integers $n \geq 1$ and $d \geq 0$, let n_d be inductively defined as $n_0 = n$ and $n_{d+1} = \lceil n_d/2 \rceil$. Also, for any integer $q \geq 1$, define the set of types $Y(q) = \{< r, s, t >: \ r, s, t \in \{q, q-1\} \ \}$. Then, in the call tree corresponding to a square matrix multiplication of type $< n, n, n >$, the type of each call-tree node at distance d from the root belongs to the set $Y(n_d)$, for $d = 0, 1, \ldots, \lceil \log n \rceil$.*

Proof. The statement trivially holds for $d = 0$ (the root), since $< n, n, n > \in Y(n) = Y(n_0)$. Assume now inductively that the statement holds for a given level d. From the closure property of the balance decomposition and the recursive decomposition of the algorithm, it follows that all matrix blocks at level $d+1$ have dimensions between $\lfloor (n_d - 1)/2 \rfloor$ and $\lceil n_d/2 \rceil$. From the identity $\lfloor (n_d - 1)/2 \rfloor =$

$\lceil n_d/2 \rceil - 1$, we have that all types at level $d+1$ belong to set $Y(\lceil n_d/2 \rceil) = Y(n_{d+1})$.

We are now in a position to give an accurate estimate of the size of the call-type dag.

Proposition 2. *Let n be of the form $n = 2^k s$, with s odd. Let $D = (U, F)$ be the call-type dag corresponding to input type $< n, n, n >$. Then, $|U| \leq k + 1 + 8(\lceil \log n \rceil - k)$.*

Proof. It is easy to see that, at level $d = 0, 1, \ldots, k$ of call tree nodes have type $< n_d, n_d, n_d >$, with $n_d = n/2^d$. For each of the remaining $(\lceil \log n \rceil - k)$ levels, there are at most 8 types per level, according to Proposition 1.

Thus, we always have $|U| = O(\log n)$, with $|U| = \log n + 1$ when n is a power of two, with $|U| \approx 8\lceil \log n \rceil$ when n is odd, and with $|U|$ somewhere in between for general n.

4.3 Bursting the Recursion

If v is an internal node of the call tree, the corresponding call receives as input a triple of blocks of A, B, and C, and produces as output the input for each child call. When matrices A, B, and C are *fractally* represented by the corresponding one-dimensional arrays a, b, and c, the input triple is uniquely determined by the type $< r, s, t >$ and by the initial positions i, j, and k of the blocks in their respective arrays. Specifically, the block of A is stored in $a[i, \ldots, i + rs - 1]$, the block of B is stored in $b[j, \ldots, j + st - 1]$, and the block of C is stored in $c[k, \ldots, k + rt - 1]$. The call at v is then responsible for the computation of the type and initial position of the sub-blocks processed by the children. For example, for the A-block $r \times s$ starting at i, the four sub-blocks have respective dimensions $\lceil r/2 \rceil \times \lceil s/2 \rceil$, $\lceil r/2 \rceil \times \lfloor s/2 \rfloor$, $\lfloor r/2 \rfloor \times \lceil s/2 \rceil$, and $\lfloor r/2 \rfloor \times \lfloor s/2 \rfloor$. They also have respective starting points i_0, i_1, i_2, and i_3, of the form $i_h = i + \Delta i_h$, where: $\Delta i_0 = 0$, $\Delta i_1 = \lceil r/2 \rceil \lceil s/2 \rceil$, $\Delta i_2 = \lceil r/2 \rceil s$, $\Delta i_3 = \lceil r/2 \rceil s + \lfloor r/2 \rfloor \lceil s/2 \rceil$. In a similar way, one can define the analogous quantities $j_h = j + \Delta j_h$ for the sub-blocks of B and $k_h = k + \Delta k_h$ for the sub-blocks of C, for $h = 0, 1, 2, 3$. During the recursion and in any node of the call tree, every Δ value is computed twice. Hence, a straightforward implementation of the fractal algorithm is bound to be rather inefficient. Two avenues can be followed, separately or in combination.

First, rather than executing the full call tree down to the n^3 leaves of type $< 1, 1, 1 >$, one can execute a pruned version of the tree. This approach reduces the recursion overheads and the straight-line coded leaves are amenable to aggressive register allocation, a subject of the next section.

Second, the integer operations are mostly the same for all calls. Hence, these operations can be performed in a preprocessing phase, storing the results in an auxiliary data structure built around the call-type dag D, to be accessed during the actual processing of the matrices. Counting the number of instructions per node, we can see a reduction of 30%.

5 Register Issues

The impact of register management on overall performance is captured by the number ρ of memory (load or store) operations per floating point operation, required by a given assembly code. In a single-pipeline machine with at most one FP or memory operation per cycle, $1/(1+\rho)$ is an upper limit to the achievable fraction of FP peak performance. The fraction lowers to $1/(1+2\rho)$ for machines where madd is available as a single-cycle instruction. To achieve 50% of peak, $\rho \leq 1$ and $\rho \leq 1/2$ is required, respectively. For machines with parallel pipes, say 1 load/store pipe every f FP pipes, an upper limit to the achievable fraction of FP peak performance becomes $\max(1, f\rho)$, so that memory instructions are not a bottleneck as long as $\rho \leq 1/f$. As for matrix multiplication, a naive implementation where each madd reads the three operands from memory and writes the result back to memory leads to $\rho = 2$. In this section, we explore two techniques which, for the typical number of registers of current RISC processors, lead to values of ρ approximately in the range 1/4 to 1/2. The general approach consists in stopping the recursion at some point and formulating the corresponding leaf computation as a straight-line code. All matrix entries are copied into a set of scalar variables, whose number R is chosen so that any reasonable compiler will permanently keep these variables in registers (*scalarization*). For a given R, the goal is then to choose where to stop the recursion and how to sequence the operations so as to minimize ρ, i.e., to minimize the number of assignments to and from scalar variables.

Fractal Sequences. One approach consists in sequencing the operations in the order that arises from the fractal scheme when the recursive process is followed all the way down to $< 1, 1, 1 >$ leaves. We have heuristically explored sequences that arise from changing the order of the subproblems at different nodes of the recursion trees (e.g., from ABC to CAB), generalizing a trick proposed in [18] (for caches). A systematic analysis will be given in the full paper. As an indication, for a $< 4, 4, 4 >$ leaf we obtain $\rho = 0.5$ (with $R \geq 28$) and for a $< 32, 32, 32 >$ leaf we obtain $\rho = 0.33$ (with $R \geq 32$).

C-tiling Sequences. The C-tiling approach, which generalizes the register allocation proposed in [36], partitions the result matrix of a generic $< m, n, p >$ leaf multiplication into rectangular tiles. An $r \times s$ tile of C is the product of an $r \times n$ sub-matrix of A and an $n \times s$ sub-matrix of B and hence can be expressed as the sum of n terms, each term is a product of a column of the A sub-matrix by a row of the B sub-matrix. If $R \geq rs + r + 1$ registers (scalar variables) are available one can: (i) load the C tile (into rs registers), (ii) load one at the time the n A-sub-columns (into r registers), (iii) load one at the time the elements of the corresponding B-sub-column and execute the r madds involving it and the elements of A currently in registers, and (iv) store back the C tiles. The number of accesses is $2rs + n(r + s)$ and the number of FP operations is $2rsn$, yielding $\rho = \frac{1}{n} + \frac{1}{2r} + \frac{1}{2s}$. The value of ρ for the full $< m, n, p >$ product is a sort of average over the chosen tiles, which might be of different sizes especially at the boundaries of the tiled sub-matrices. As an indication, for an $< 8,8,8 >$ leaf we obtain $\rho = 0.50$ (with $R \geq 21$) and for a $< 32,32,32 >$ leaf we obtain $\rho = 0.25$ (with $R \geq 32$).

6 Experimental Results

We have studied experimentally both the cache behavior of fractal algorithms, in terms of misses, and the overall performance, in terms of running time.

6.1 Cache Misses

The results of this section are based on simulations performed (on an SPARC Ultra 5) using the *Shade* software package for Solaris, of Sun Microsystems. Codes are compiled for the SPARC ultra2 processor architecture (V8+) and then simulated for various cache configurations, chosen to correspond to those of a number of commercial machines. Thus when we refer, say, to the R5000 IP32, we are really simulating a ultra2 CPU with the memory hierarchy of the R5000 IP32.

In fractal codes, (i) the recursion is stopped when the size of the leaves is strictly smaller than problem $< 32,32,32 >$; (ii) the recursive layout is stopped when a sub-matrix is strictly smaller than 32×32; (iii) the leaves are implemented with C-tiling register assignment using $R = 24$ variables for scalarization (this leaves the compiler 8 of the 32 registers to buffer multiplication outputs before they are accumulated into C-entries). The leaves are compiled with cc WorkShop 4.2 and linked statically. The recursive algorithms, i.e. ABC-Fractal and CAB-Fractal, are compiled with gcc 2.95.1.

We have also simulated the code for ATLAS DGEMM obtained by installation of the package on the Ultra 5 architecture. This is to have another term of reference, and generally fractal has fewer misses. However, it would be unfair to regard this as a competitive comparison with ATLAS, which is meant to be efficient by adapting to the varying cache configuration.

We have simulated 7 different cache configurations (Table 1). Notationally, I= Instruction cache, D=Data cache, and U=Unified cache. We have measured the number $\mu(n)$ of misses per flop and compared it against the value of the estimator (Section 4.1) $\mu(n) = 2.6\gamma(n)/(\ell\sqrt{s})$, where s and ℓ are the number of (64 bit) words in the cache and in one line, respectively, and where we expect values of $\gamma(n)$ not much greater than one. In Table 1, we have reported the value of $\mu(1000)$ measured for CAB-fractal and the corresponding value of $\gamma(1000)$ (last column). More detailed simulation results are given in the Appendix (Figures 4 to 10). We can see that γ is generally between 1 and 2; thus, our estimator gives a reasonably accurate prediction of cache performance. This performance is consistently good on the various configurations, indicating efficient portability. For completeness, we have also reported simulation results for code misses: although the comparatively large size of the leaf procedures does increase such misses, they remain negligible with respect to data misses.

Table 1. Summary of simulated configurations

Simulated Configuration		Size (Bytes/s)	Line (Bytes,ℓ)	Associativity/ Write Policy	$\mu(1000)/\gamma(1000)$
SPARC 1					
	U1	64KB / 8K	16B / 2	1 / through	2.65e-2 / 1.84
SPARC 5					
	I1	16KB	16B	1 /	
	D1	8KB / 1K	16B / 2	1 / through	5.96e-2 / 1.47
Ultra 5					
	I1	16KB	32B	2 /	
	D1	16KB / 2K	32B / 4	1 / through	2.51e-2 / 1.75
	U2	2MB / 256K	64B / 8	1 / back	1.05e-3 / 1.66
R5000 IP32					
	I1	32KB	32B	2 / back	
	D1	32KB / 4K	32B / 4	2 / back	1.06e-2 / 1.04
	U2	512KB / 64K	32B / 4	1 / back	3.61e-3 / 1.42
Pentium II					
	I1	16KB	32B	1 /	
	D1	16KB / 2K	32B / 4	1 / through	2.50e-2 / 1.74
	U2	512KB / 64K	32B / 4	1 / back	3.98e-3 / 1.57
HAL Station					
	I1	128KB	128B	4 / back	
	D1	128KB / 16K	128B / 16	4 / back	2.65e-3 / 2.09
ALPHA 21164					
	I1	8KB	32B	1 /	
	D1	8KB / 1K	32B / 4	1 / through	3.75e-2 / 1.85
	U2	96KB / 12K	32B / 4	3 / back	5.81e-3 / 0.99

6.2 Running Time

While portability of cache performance is desirable, it is important to explore the extent to which it can be combined with optimizations of CPU performance. We have tested the fractal approach on the four different processors listed in Table 2, using always the same code for the recursive decomposition (which is essentially responsible for cache behavior) and varying the code for the leaves, to adapt the number of scalar variables R to the processor: $R = 24$ for Ultra2i (Ultra 5), $R = 8$ for Pentium II, and $R = 32$ for R5000 (IP32) and SPARC64 (HAL Station). We compare the running time (or, equivalently, the MFLOPS) of fractal algorithms in double precision with peak performance and with the performance of ATALS-DGEMM, if available. Fractal achieves performances comparable to those of ATLAS, being at most 2 times slower (on PentiumII) and a little faster on SGI R5000 IP32. Since no special adaptation to the processor has been performed on the fractal codes, except for the number of scalar variables, we conclude that the portability of cache performance can be combined with overall performance. More detailed running time results are reported in the Appendix (Figures 12 to 11.)

Table 2. Processor Configurations

Processor	Ultra 2i (Ultra 5)	PentiumII	R5000 (IP32)	SPARC64 (HAL Station)
Registers Structure	32 64-bit register file	8 80-bit stack file	32 64-bit register file	32 64-bit register file
Multiplier Adder	distinct	distinct	single FU	single FU
Latency FP (Cycles)	3	8	2	4
Peak (MFLOPS)	666	400	360	200
Peak of CAB-Fr. / matrix size	425 / 444×444	187 /400×400	133 / 504×504	168 / 512×512
Peak of ATLAS / matrix size	455 / 220×220	318 / 848×848	113 / unknown	not available

7 Conclusions

In this paper, we have developed a close study of matrix multiplication showing that suitable algorithms can efficiently exploit the cache hierarchy without taking cache parameters into account, thus ensuring portability of cache performance. Clearly, performance itself does depend on cache parameters and we have provided a reasonable estimator for it. We have also experimentally shown that, with a careful implementation of recursion, high performance is achievable. We hope the present study will motivate extension in various directions, both in terms of results and in terms of techniques. In [13], we have already used the fractal multiplication codes and recursive code optimizations of this paper to obtain implementation of other linear algebra algorithms, such as those for LU decomposition of [34], with overall performance higher than other multiplication-based algorithms.

References

1. A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir: A model for hierarchical memory. Proc. of 19th Annual ACM Symposium on the Theory of Computing, New York, 1987,305-314.
2. A. Aggarwal, A.K. Chandra and M. Snir: Hierarchical memory with block transfer. 1987 IEEE.
3. B. Alpern, L. Carter, E. Feig and T. Selker: The uniform memory hierarchy model of computation. In *Algorithmica*, vol. 12, (1994), 72-129.
4. U.Banerjee, R.Eigenmann, A.Nicolau and D.Padua: Automatic program parallelization. Proceedings of the IEEE vol 81, n.2 Feb. 1993.
5. G.Bilardi and F.Preparata: Processor-time tradeoffs under bounded-speed message propagation. Part II: lower bounds. Theory of Computing Systems, Vol. 32, 531-559, 1999.

25. N.J.Higham: Accuracy and stability of numerical algorithms ed. SIAM 1996
26. Hong Jia-Wei and T.H.Kung: I/O complexity :The Red-Blue pebble game. Proc.of the 13th Ann. ACM Symposium on Theory of Computing Oct.1981,326-333.
27. B.Kågström, P.Ling and C.Van Loan: Algorithm 784: GEMM-based level 3 BLAS: portability and optimization issues. ACM transactions on Mathematical Software, Vol24, No.3, Sept.1998, pages 303-316
28. B.Kågström, P.Ling and C.Van Loan: GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. ACM transactions on Mathematical Software, Vol24, No.3, Sept.1998, pages 268-302.
29. M.Lam, E.Rothberg and M.Wolfe: The cache performance and optimizations of blocked algorithms. Proceedings of the fourth international conference on architectural support for programming languages and operating system, Apr.1991,pg. 63-74.
30. S.S.Muchnick: Advanced compiler design implementation. Morgan Kaufman
31. P.R.Panda, H.Nakamura, N.D.Dutt and A.Nicolau: Improving cache performance through tiling and data alignment. Solving Irregularly Structured Problems in Parallel Lecture Notes in Computer Science, Springer-Verlag 1997.
32. John E.Savage: Space-Time tradeoff in memory hierarchies. Technical report Oct 19, 1993.
33. V.Strassen: Gaussian elimination is not optimal. Numerische Mathematik 14(3):354-356, 1969.
34. S.Toledo: Locality of reference in LU decomposition with partial pivoting. SIAM J.Matrix Anal. Appl. Vol.18, No. 4, pp.1065-1081, Oct.1997
35. M.Thottethodi, S.Chatterjee and A.R.Lebeck: Tuning Strassen's matrix multiplication for memory efficiency. Proc. SC98, Orlando,FL, nov.1998 (http://www.supercomp.org/sc98).
36. R.C.Whaley and J.J.Dongarra: Automatically Tuned Linear Algebra Software. http://www.netlib.org/atlas/index.html
37. D.S.Wise: Undulant-block elimination and integer-preserving matrix inversion. Technical Report 418 Computer Science Department Indiana University August 1995
38. M.Wolfe: More iteration space tiling. Proceedings of Supercomputing, Nov.1989, pg. 655-665.
39. M.Wolfe and M.Lam: A Data locality optimizing algorithm. Proceedings of the ACM SIGPLAN'91 conference on programming Language Design and Implementation, Toronto, Ontario,Canada,June 26-28, 1991.
40. M.Wolfe: High performance compilers for parallel computing. Addison-Wesley Pub.Co.1995

Appendix: Figures

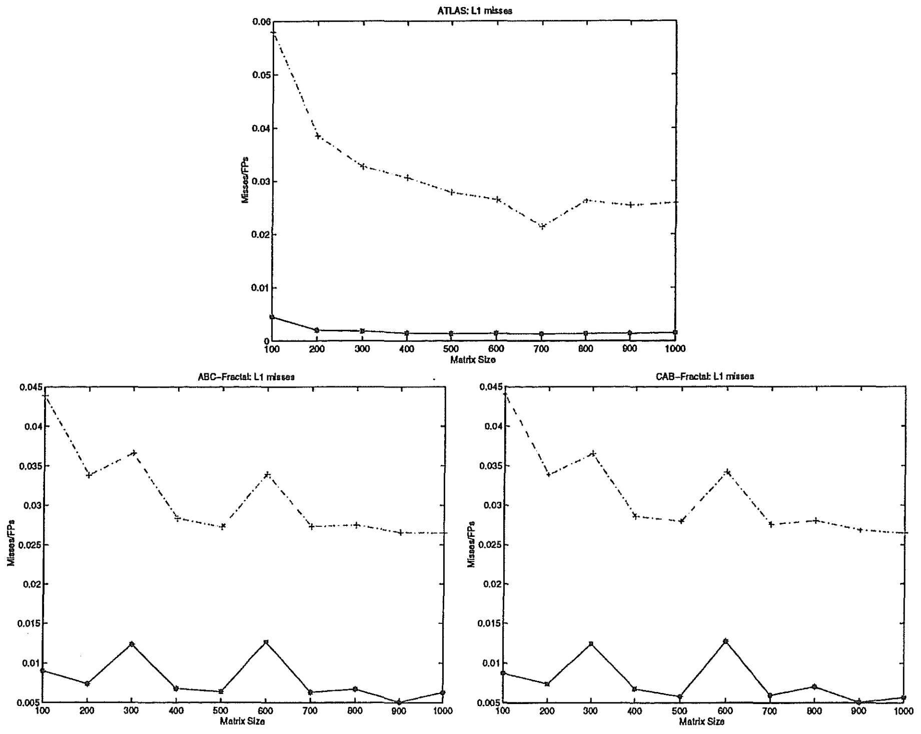

Fig. 4. SPARC 1, (+) normalized number of data miss, $\mu(n)$, and (*) code miss. The algorithms DGEMM from ATLAS, ABC-Fractal and CAB fractal have been simulated with matrixes in double precision. ATLAS has a slightly smaller number of miss than the Fractal approach. ATLAS code size is small, it has good time locality (there is one procedure called the most) and interferes a few times with data. Fractal approaches, instead, have large code size, it changes frequently and it increases interference with the common data.

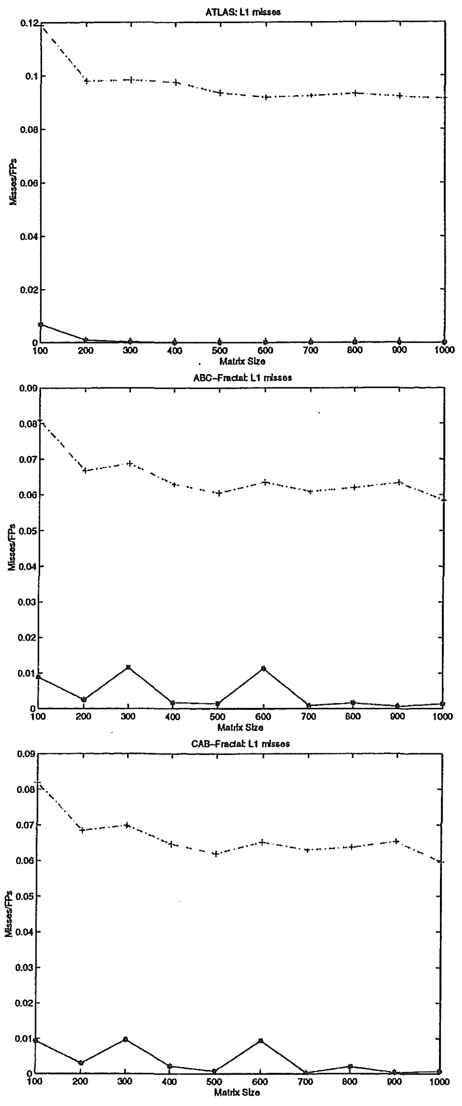

Fig. 5. SPARC 5, (+) normalized number of data miss, $\mu(n)$, and (*) code miss. The algorithms DGEMM ATLAS, ABC-Fractal and CAB fractal have been simulated with matrixes in double precision. ATLAS has a very good code locality but poor data locality.

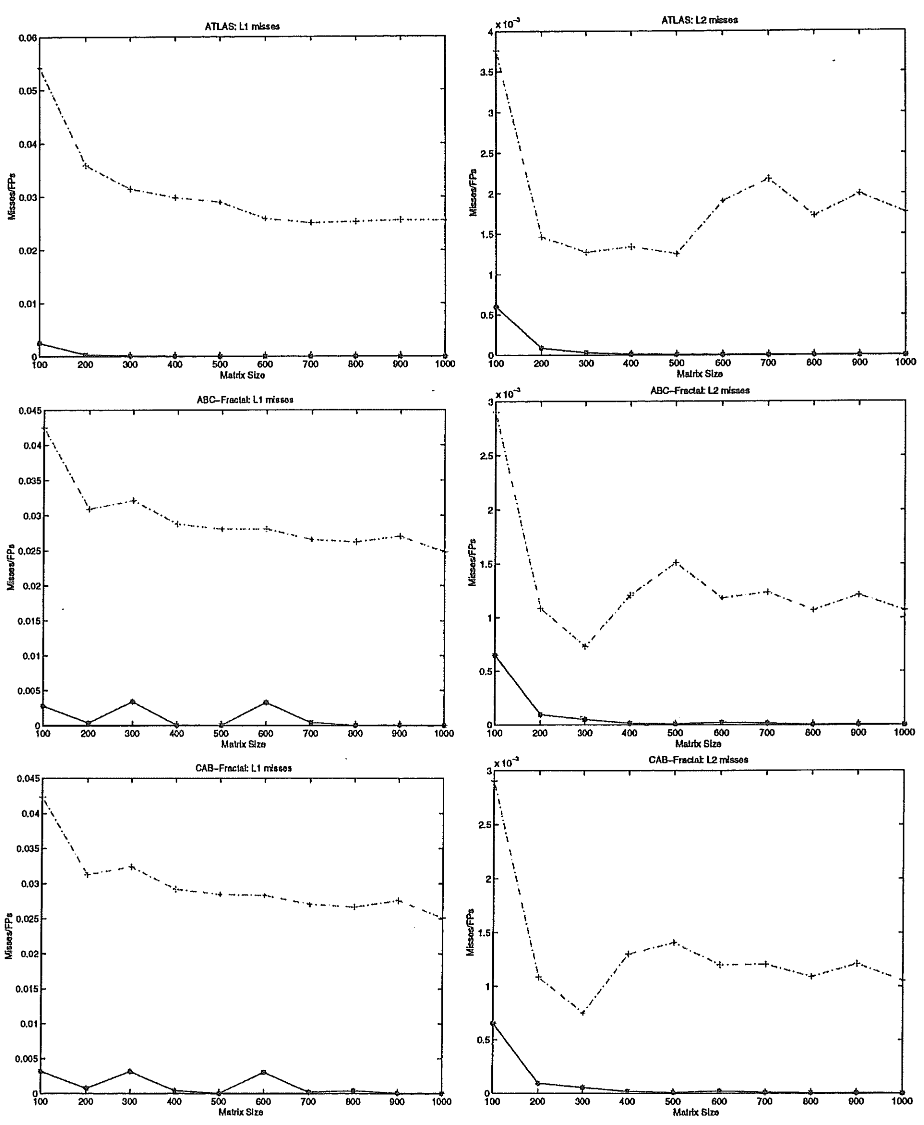

Fig. 6. Ultra 5, (+) normalized number of data miss, $\mu(n)$, and (*) code miss. The algorithms DGEMM ATLAS, ABC-Fractal and CAB fractal have been simulated with matrixes in double precision. All algorithms are designed with particular attention to this architecture and therefore they have very similar performance. The Fractal approaches have a slightly better performance at the second level cache.

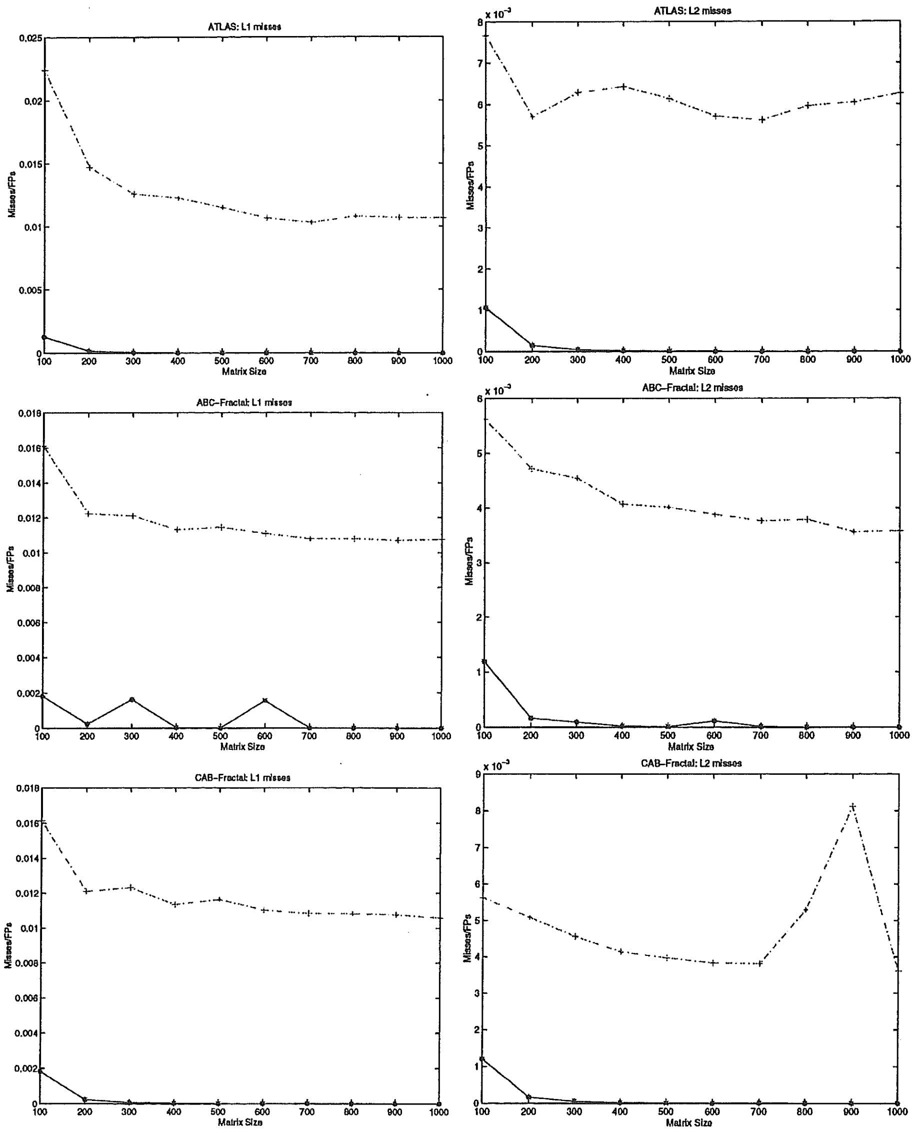

Fig. 7. R5000 IP32,(+) normalized number of data miss, $\mu(n)$, and (*) code miss. The algorithms DGEMM ATLAS, ABC-Fractal and CAB fractal have been simulated with matrixes in double precision. Algorithm CAB-Fractal behaves differently: 1) the erratic peak of number of misses at problem $< 900, 900, 900 >$ is due to write misses, 96% of writes are write misses; 2) the code misses are completely negligible.

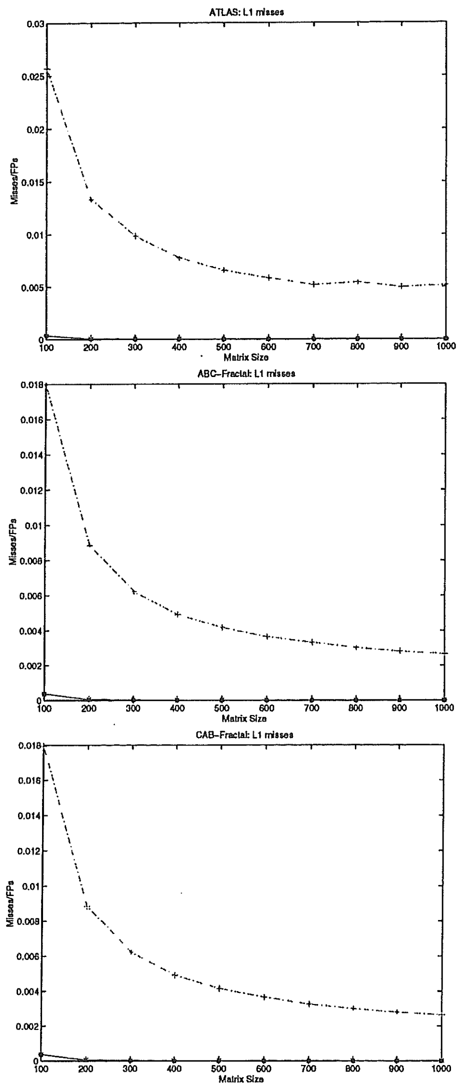

Fig. 9. HAL Station,(+) normalized number of data miss, $\mu(n)$, and (*) code miss. The algorithms DGEMM ATLAS, ABC-Fractal and CAB fractal have been simulated with matrixes in double precision. Code misses are just compulsory misses and Fractal algorithms are always better than ATLAS.

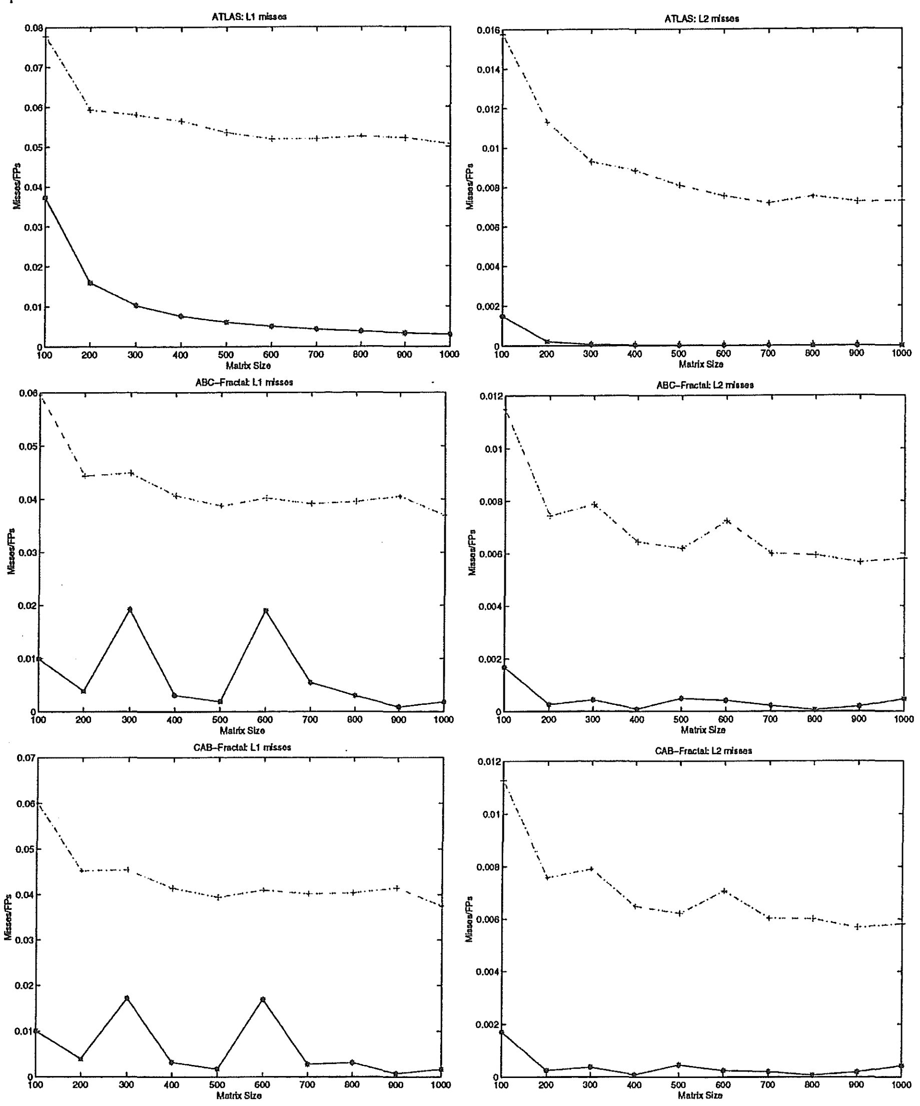

Fig. 10. Alpha 21164, (+) normalized number of data miss, $\mu(n)$, and (*) code miss. The algorithms DGEMM ATLAS, ABC-Fractal and CAB fractal have been simulated with matrixes in double precision. The Fractal algorithms have a better data locality at every level of the memory hierarchy but they have a poor code locality at the first level of cache.

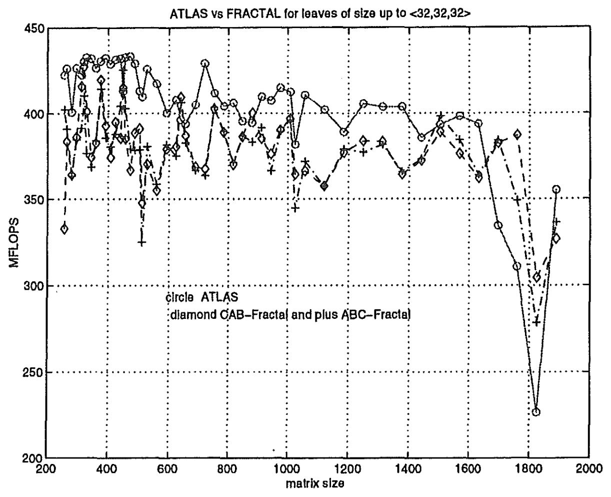

Fig. 11. **Ultra 5** ATLAS has slightly better performance (¡10%) than any Fractal.

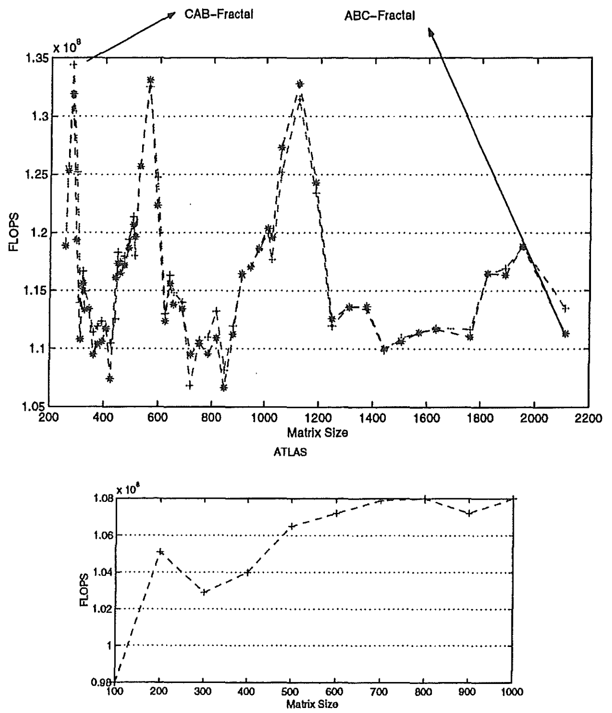

Fig. 12. **R5000 IP_32.** We report DGEMM ATLAS performance according to its authors [36]. CAB-Fractal achieves a somewhat better performance.

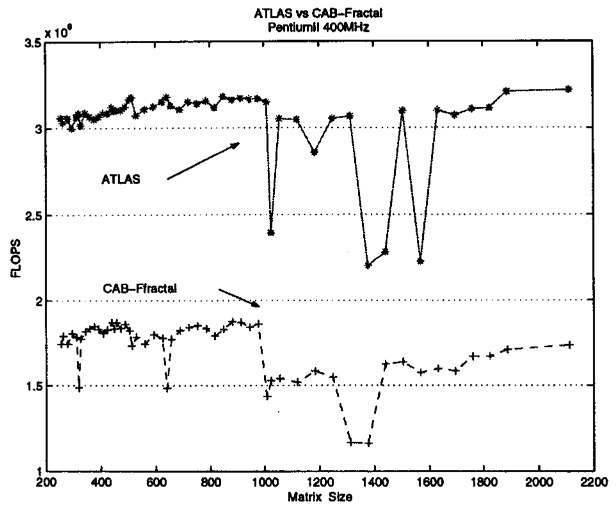

Fig. 13. PentiumII. ATLAS was motivated by the lack of a vendor library for Pentium. ATLAS is twice as fast than any fractal approach. C-tiling does not work well on stack register files. ATLAS uses just 5 registers and exploits the Floating point structure of Pentium II.

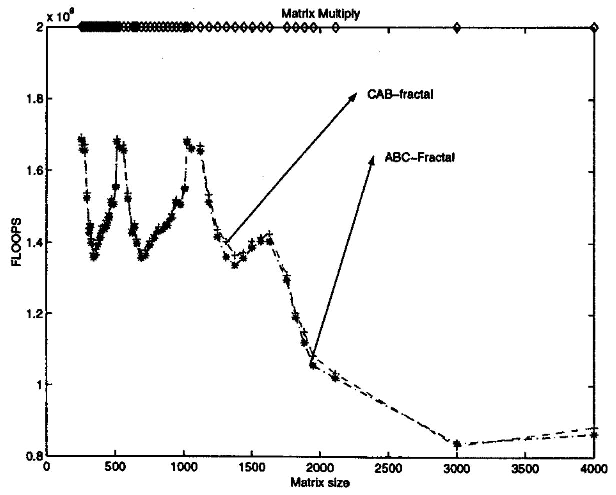

Fig. 14. HAL Station. We were not able to install ATLAS here. When the matrices fit in main memory, performance is above 75% of peak.