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 †

Abstract. The practical portability of a simple version of matrix multiplication is demonstrated. The multiplication algorithm is designed to exploit maximal and predictable locality at all levels of the memory hierarchy, with no *a priori* knowledge of the specific memory system organization for any particular machine. By both simulations and execution on a number of platforms, we show that memory hierarchies portability does not sacrifice floating point performance; indeed, it is always a significant fraction of peak and, at least on one machine, is higher than the tuned routines by both ATLAS and vendor. The results are obtained by careful algorithm engineering, which combines a number of known as well as novel implementation ideas. This effort can be viewed as an experimental case study, complementary to the theoretical investigations on portability of cache performance begun by Bilardi and Peserico.

1 Introduction

The ratio between main memory access time and processor clock cycle has been continuously increasing, up to values of a few hundreds nowadays. The increase in Instruction Level Parallelism (ILP) has been a significant feature: current CPUs can issue four/six instructions per cycle and the cost of a memory access is an increasingly high toll on overall performance of super-scalar/VLIW processors. The architectural response 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. The memory hierarchy helps performance only to the extent to which the computation exhibits data and code locality. The necessary amount of locality becomes greater with steeper hierarchies, an issue that algorithm design and compiler optimization increasingly need to take 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 (extended in [35, 3]). In this model, the time to access a location x is a function $f(x)$; the authors observe that optimal algorithms are achieved for a wide family

*** This work was supported, in part, by CNR and MURST of Italy

† Supported by AMRM DABT63-98-C-0045

of functions f. More recently, similar results have been obtained for a different model, with automatically managed caches [22]. 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 [28] and recently extended in [6, 9]; these techniques are crucial to establish the existence of portable implementations for some algorithms, such as matrix multiplication. The question whether arbitrary computations admit optimally portable implementations has been investigated in [7, 8]. Even though the answer is generally negative, the computations that admit portable implementations do include relevant classes such as linear algebra kernels ([29, 30]).

This work focuses on matrix multiplications algorithms with complexity $O(n^3)$ (rather than $O(n^{\log_2 7})$ [36] or $O(n^{2.376})$ [14]) investigating the impact on performance of data layout, latency hiding, register allocation, instruction scheduling, instruction parallelism, (e.g., [39, 10, 16–18])[1] and their interdependences. The interdependence between tiling and sizes of caches is probably the most investigated [31, 34, 41, 32, 43, 23, 17]. For example, vendor libraries (such as BLAS from SGI and SUN) exploit their knowledge of the destination platform and determine very efficient routines, but non optimally portable across different platforms. Automatically tuned packages (see [39, 10] matrix multiply and [21] FFT) measure *machine parameters* by interactive tests and then produce machine tuned code. This approach achieves optimal performance and *portability* at the level of package, rather than the actual application code. Another approach, called *auto-blocking*, has the potential to yield portable performance for the individual code. Informally, one can think of a tile 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 [25], with applications to LAPACK, and its asymptotic optimality is discussed in [22]. Our fractal algorithms belong to this framework. Recursion-based algorithms often exploit various features of non-standard layouts, *recursive layouts* ([13, 12, 38, 20, 40, 26, 19]). Conversion from and to standard (i.e., row-major and column-major) layouts introduces $O(n^2)$ overheads[2]. 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 ([17, 39]); for recursive algorithms the problem is worse because no compiler is capable of unfolding the calls in order to expose larger sets of operations to aggressive optimizations. We propose a pruning of the recursion tree to circumvent this problem.

[1] See [41, 42, 23, 4, 11, 31] for more general locality approaches suitable at compile time and used for linear algebra kernels.

[2] The overheads are negligible, except for matrices small enough for the n^2/n^3 ratio to be insignificant, or large enough to require disk access.

Our approach, hereafter *fractal* approach, combines a number of known ideas and techniques as well as some novel ones to achieve the following results.

1) There exists a matrix multiplication implementation for modern ILP machines achieving excellent, portable cache performance, and we show it through simulations of 7 different machines. 2) The overall performance (FLOPS) is very good in practice, and we show it by comparison with the upper bound implied by peak and performance of the best known code (Automatically Tuned Linear Algebra Software, ATLAS, [39]). 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. Among the techniques we have developed, those in Sections 2.2 and 2.3 lead to efficient implementations of recursive procedures. They are especially worth mentioning because they are likely to be applicable to many other hierarchy-oriented codes. In fact, it can be argued with some generality that recursive code is naturally more conducive to express temporal locality than code written in the form of nested loops. Numerical stability is not considered in this paper (Lemma 2.4.1 resp. 3.4 in [24] resp. [27]).

2 Fractal Algorithms for Matrix Multiplication

We use the following recursive layout of an $m \times n$ matrix A into a one-dimensional array $\mathbf{a}$ of size mn. If $m = 1$, then $\mathbf{a}[h] = a_{0h}$, for $h = 0, 1, \ldots, n-1$. If $n = 1$, then $\mathbf{a}[h] = a_{h0}$, for $h = 0, 1, \ldots, m-1$. Otherwise, $\mathbf{a}$ is the concatenation of the layouts of the blocks A_0, A_1, A_2, and A_3 of the following *balanced* decomposition. $A_0 = \{a_{ij} : 0 \leq i < \lceil m/2 \rceil, 0 \leq j < \lceil n/2 \rceil\}$, $A_1 = \{a_{ij} : 0 \leq i < \lceil m/2 \rceil, \lceil n/2 \rceil \leq j < n\}$, $A_2 = \{a_{ij} : \lceil m/2 \rceil \leq I < m, 0 \leq j < \lceil n/2 \rceil\}$ and $A_3 = \{a_{ij} : \lceil m/2 \rceil \leq i < m, \lceil n/2 \rceil \leq j < n\}$. 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 $\max(S) - \min(S) \leq 1$. The fractal layout just defined can be viewed as a generalization of the Z-Morton layout for square matrixes [12], [20] or as a special case of the Quad-Tree [19] layout.

We introduce now the fractal algorithms, a class of procedures all variants of a common scheme, for the operation of matrix multiply-and-add (MADD) $C = C + AB$, also denoted $C+ = AB$. For near square matrices, the *fractal scheme* to perform $C+ = AB$ is recursively defined as follows, with reference to the above balanced decomposition.

fractal(A, B, C)

- If $|A| = |B| = 1$, then $C = C + A * B$ (all matrices being scalar).
- 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. The vertices correspond to the 8 recursive calls in the fractal scheme. The 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. 1).

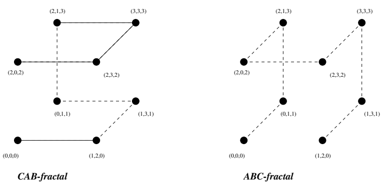

Fig. 1. The cube of calls of the fractal scheme: the Hamiltonian path defining CAB-fractal and ABC-fractal.

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 orderings: one reducing write misses and one reducing read misses. 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).

2.1 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 (LRU) and with cache lines holding exactly one matrix entry, the layout is immaterial to performance. The fractal approach exploits temporal locality for any cache independently of its size s (in matrix entries). Indeed, consider the case when at the highest level of recursion all calls use matrix blocks that fit in cache simultaneously. Approximately, the matrix blocks are of size $s/3$. Each call load will cause about s misses. Each call computes up to $(\sqrt{s/3})^3 = s\sqrt{s}/3\sqrt{3}$ scalar MADDs. The ratio misses per FLOP is estimated as $\mu = (3\sqrt{3}(/(2\sqrt{s}) \approx 2.6/\sqrt{s}$. (This is within a constant factor of optimal, Corollary 6.2 [28].)

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 a 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 has no self interference for blocks that fit in cache. The misses per flop is estimated as $\mu = 2.6\gamma/\ell\sqrt{s}$, where 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), relative fractal arrays positions in memory, cache associativity and, sometimes, register allocation. When interference is negligible, we can expect $\gamma \approx 1$.

2.2 The Structure of the Call Tree

Pursuing efficient implementations for the fractal algorithms we face the usual performance drawbacks of recursion: overheads and poor register utilization (due to lack of code exposure to the compiler). To circumvent such drawbacks, we carefully study the structure of the call tree.

Definition 1. *Given a fractal algorithm $\mathcal{A}$, its* call tree $T = (V, E)$ *w.r.t. 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 an internal node v correspond to the calls made by v in order of execution.*

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 $< \emptyset >$. The structure of T is uniquely determined by type of the root. We focus on square matrices, i.e. type $< n, n, n >$ for which the tree has depth $\lceil \log n \rceil + 1$ and it has $8^{\lceil \log n \rceil}$ leaves. n^3 leaves have type $< 1, 1, 1 >$ and correspond (from left to right) to the n^3 MADDs of the algorithm. The remaining leaves have empty type. Internal nodes are essentially responsible for performing the problem decomposition; their specific computation depends on the way matrices are represented. An internal node has typically eight non-empty children, except when its type has at least one components equal to 1, e.g., $< 2, 1, 1 >$ or $< 2, 2, 1 >$, in which 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 this issue systematically, we introduce the concept of *type DAG*. 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 DAG, 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. See Figure 2 for an example. Next, we study the size of the call-type DAG D for the case of square matrix multiplication. We begin by showing that there are at most 8 types of input for the calls of a given level of recursion.

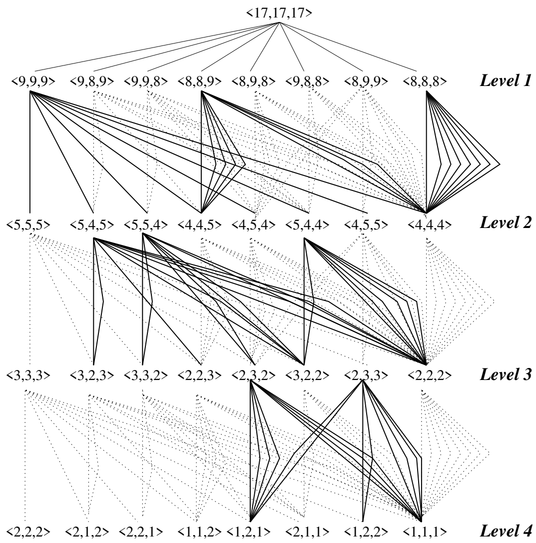

Fig. 2. Example of call-type DAG for Matrix Multiplication $< 17, 17, 17 >$

Proposition 1. *For any integers $n \geq 1$ and $d \geq 0$, let n_d be defined inductively 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 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 $Y(\lceil n_d/2 \rceil) = Y(n_{d+1})$.

Now, we can give an accurate size estimate 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.

2.3 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: $\Delta i_0 = 0$, $\Delta i_1 = \lceil r/2 \rceil \lceil s/2 \rceil$, $\Delta i_2 = \lceil r/2 \rceil s$, $\Delta i_3 = \Delta i_2 + \lfloor r/2 \rfloor \lceil s/2 \rceil$. 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%.

3 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. 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 $\rho \leq 1/f$. 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.

We investigated and implemented two different scalar replacements: *Fractal Sequence* "inspired" by [20] and *C-tiling sequence* inspired by [39] (see [5, 33] for a full description).

4 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.

4.1 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+, no MADD operation available) 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 (as suggested in [39]). 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 used as 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); we use the notation: 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 2.1) $\mu(n) = 2.6\gamma(n)/(\ell\sqrt{s})$, 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 [5]. 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, in [5], we have also

Table 1. Summary of simulated configurations

Simulated	Conf.	Size (Bytes/s)	Line (Bytes,ℓ)	Assoc./WritePol.	$\mu(1000)$/ $\gamma(1000)$
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

reported simulation results for code misses: although these misses do increase due to the comparatively large size of the leaf procedures, they remain negligible with respect to data misses.

4.2 MFLOPS

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 four different processors listed in Table 2. We always use the same code for the recursive decomposition (which is essentially responsible for cache behavior). We vary the code for the leaves, to adapt the number of scalar variables R to the processor: $R = 24$ for Ultra 5, $R = 8$ for Pentium II, and $R = 32$ for SGI R5K IP32 and HAL Station. We compare 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 (which is not a RISC) and a little faster on SGI R5K. 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 [5]

5 Conclusions

In this paper, we have developed a careful study of matrix multiplication implementations, showing that suitable algorithms can efficiently exploit the cache

Table 2. Processor Configurations

Processor	Ultra 2i (Ultra 5)	PentiumII	R5000 (IP32)	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
FP Lat.(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

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 [15], 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 [37], 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, P. D'Alberto, and A. Nicolau: Fractal Matrix Multiplication: a Case Study on Portability of Cache Performance, *University of California at Irvine*, ICS TR#00-21, 2000.
6. G. Bilardi and F.P. Preparata: Processor-time tradeoffs under bounded-speed message propagation. Part II: lower bounds. Theory of Computing Systems, Vol. 32, 531-559, 1999.
7. G. Bilardi, E. Peserico: An Approach toward an Analytical Characterization of Locality and its Portability. *IWIA 2000, International Workshop on Innovative Architectures*, Maui, Hawai, January 2001.
8. G. Bilardi, E. Peserico: A Characterization of Temporal Locality and its Portability Across Memory Hierarchies. *ICALP 2001, International Colloquium on Automata, Languages, and Programming*, Crete, July 2001.

9. G. Bilardi, A. Pietracaprina, and P. D'Alberto: On the space and access complexity of computation DAGs. 26th Workshop on Graph-Theoretic Concepts in Computer Science, Konstanz, Germany, June 2000.
10. J. Bilmes, Krste Asanovic, C. Chin and J. Demmel: Optimizing matrix multiply using PHiPAC: a portable, high-performance, Ansi C coding methodology. International Conference on Supercomputing, July 1997.
11. S. Carr and K. Kennedy: Compiler blockability of numerical algorithms. Proceedings of Supercomputing Nov 1992, pg.114-124.
12. S. Chatterjee, V.V. Jain, A.R. Lebeck and S. Mundhra: Nonlinear array layouts for hierarchical memory systems. Proc. of ACM international Conference on Supercomputing, Rhodes,Greece, June 1999.
13. S. Chatterjee, A.R. Lebeck, P.K. Patnala and M. Thottethodi: Recursive array layout and fast parallel matrix multiplication. Proc. 11-th ACM SIGPLAN, June 1999.
14. D. Coppersmith and S. Winograd: Matrix multiplication via arithmetic progression. In Poceedings of 9th annual ACM Symposium on Theory of Computing pag.1-6, 1987.
15. P. D'Alberto, G. Bilardi and A. Nicolau: Fractal LU-decomposition with partial pivoting. Manuscript.
16. M.J. Dayde and I.S. Duff: A blocked implementation of level 3 BLAS for RISC processors. `TR_PA_96_06`, available on line `http://www.cerfacs.fr/algor/reports/TR_PA_96_06.ps.gz` Apr. 6 1996
17. N. Eiron, M. Rodeh and I. Steinwarts: Matrix multiplication: a case study of algorithm engineering. Proceedings WAE'98, Saarbrücken, Germany, Aug.20-22, 1998
18. Engineering and Scientific Subroutine Library. `http://www.rs6000.ibm.com/resource/aix_resource/sp_books/essl/`
19. P. Flajolet, G. Gonnet, C. Puech and J.M. Robson: The analysis of multidimentional searching in Quad-Tree. Proceeding of the second Annual ACM-SIAM symposium on Discrete Algorithms, San Francisco, 1991, pag.100-109.
20. J.D. Frens and D.S. Wise: Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming, SIGPLAN Not. 32, 7 (July 1997), 206–216.
21. M. Frigo and S.G. Johnson: The fastest Fourier transform in the west. MIT-LCS-TR-728 Massachusetts Institute of technology, Sep. 11 1997.
22. M. Frigo, C.E. Leiserson, H. Prokop and S. Ramachandran: Cache-oblivious algorithms. Proc. 40th Annual Symposium on Foundations of Computer Science, (1999)
23. E.D. Granston, W. Jalby and O. Teman: To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. Proceedings of Supercomputing Nov 1993, pg.410-419.
24. G.H. Golub and C.F. van Loan: Matrix computations. Johns Hopkins editor 3-rd edition
25. F.G. Gustavson: Recursion leads to automatic variable blocking for dense linear algebra algorithms. Journal of Research and Development Volume 41, Number 6, November 1997
26. F. Gustavson, A. Henriksson, I. Jonsson, P. Ling, and B. Kagstrom: Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In B. Kagstrom et al (eds), Applied Parallel Computing. Large Scale Scientific and Industrial Problems, PARA'98 Proceedings. Lecture Notes in Computing Science, No. 1541, p. 195-206, Springer Verlag, 1998.

27. N.J. Higham: Accuracy and stability of numerical algorithms ed. SIAM 1996
28. 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.
29. 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
30. 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.
31. 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.
32. S.S. Muchnick: Advanced compiler design implementation. Morgan Kaufman
33. P. D'Alberto: Performance Evaluation of Data Locality Exploitation. Techincal Report UBLCS-2000-9. Department of Computer Science, University of Bologna.
34. 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.
35. John E. Savage: Space-Time tradeoff in memory hierarchies. Technical report Oct 19, 1993.
36. V.Strassen: Gaussian elimination is not optimal. Numerische Mathematik 14(3):354-356, 1969.
37. 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
38. 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).
39. R.C.Whaley and J.J.Dongarra: Automatically Tuned Linear Algebra Software. http://www.netlib.org/atlas/index.html
40. D.S. Wise: Undulant-block elimination and integer-preserving matrix inversion. Technical Report 418 Computer Science Department Indiana University August 1995
41. M. Wolfe: More iteration space tiling. Proceedings of Supercomputing, Nov.1989, pg. 655-665.
42. 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.
43. M. Wolfe: High performance compilers for parallel computing. Addison-Wesley Pub.Co.1995