# Fractal LU-Decomposition with Partial Pivoting

P.D'Alberto G.Bilardi A.Nicolau

August 8, 2000

## 1 Introduction

In the last decades we have seen the proliferation of several high performing software packages for the solution of matrix computation problems. Because of the availability of fast low-cost desktop and workstation computer systems, different version of the same software package has been reworked for different platforms. Sometime the same package has been re-written in different dialects of the same programming language or in different languages, i.e. Fortran to C. The variety of architectures, operating systems, applications and also users can be overwelming, for a team about to produce any software package claiming high performance across platforms. In such a heterogeneous systems, portability of a software package is major concern. BLAS is *de-facto* the standard library for *Linear Algebra* (i.e. [17, 15, 40, 10]). BLAS is composed of three layers, namely 1, 2 and 3, where each layer is based upon the lower ones. BLAS level 1 describes routines for scalar by vector operations, BLAS level 2 describes routines for vectors by vectors operations and BLAS level 3 describes matrix by matrix computations, i.e. matrix multiplication. BLAS is a description of a common interface, it is a portable package for linear algebra applications and support for more complex packages as LINPACK, EISPACK, LAPACK, MATLAB (i.e. [4, 36, 41, 32]). Peak performance across different platforms was not BLAS's intent, but portability and fairly good performance were. Indeed, they proposed a rough but general register allocation for RISC architectures, and the basic idea can be still found in implementations of more performing and up-dated packages. Then, performance has been pursued in more systematically ways, i.e. improving the implementation of every routine. Due to the different characteristics of modern RISC/ILP the portability could be traded in for performance (and vice-versa) and hand coded optimizations were replaced by optimizing scripts introducing a new approach to build scientific code libraries. Scripts inquiry the system determining the main characteristics. The source code can be written automatically and on demand respecting some of the parameters so determined. The binary code is generated and installed. A big step towards a portable & performing BLAS software package was given by the discover that only a few number of routines may need re-working for an architecture: shortening the installation and focusing the attention on a few lines of code more than several procedures ([29, 30]). Matrix multiplication is (one of) *the basic procedures*. Today, there are packages that implement BLAS 3 focusing just on re-target-able matrix multiplication and achieving optimal performance [10, 40].

In this paper we investigate LU-decomposition with partial pivoting which is important to solve linear, dense and very large systems. For example, in a system $AX = B$ ($A$ and $B$ constants and $X$ free variables) matrix $A$ is decomposed in the product of one lower triangular matrix $L$ by one upper triangular matrix $U$. The system is reduced to two easier-to-solve triangular systems: $LY = B$ and $UX = Y$. LU-decomposition has a simple and elegant blocked description where matrix multiplication can be basic operation on fixed in size blocks [23]. Partial pivoting is a simple, efficient and, for most practical cases, technique to make the LU decomposition numerically stable. Indeed, it performs rows permutations during the matrix decomposition. Complete pivoting is a technique to make the LU decomposition numerically stable. It performs rows and columns permutations during the decomposition. The two approaches differ by their "stability" and by the extra computation introduced to the decomposition: for a square matrix $A$ of size $m \times m$, partial pivoting takes $O(m^2)$ steps and complete pivoting takes $O(m^3)$ steps, which is dominant for any implementation of the LU-decomposition ($O(m^3)$), [38, 27, 23]. Matrix permutations involve operations on slow components, as caches and memory, and each permutation synchronizes the computaion and it may stop the computation stalling faster components, i.e. ALUs and FPUs [26, 16].

We are taking a step backwards from the current trend pursuing static code optimiazations, at source code,

without any interaction with the architecture (that we believe in the next future a compiler will be able to do automatically). We implement Toledo's algorithm, [38] for LU-decomposition with partial pivoting using non standard layout [12, 13], fractal matrix multiplication [6] and the optimizations introduced in the fractal framework. We are going to show how the fractal framework works for LU-decomposition and how, surprisingly, the performance can overcome other implementations based on *faster* matrix multiplication routines. Because the optimality of the implementation of matrix multiply is not a sufficient condition for the optimality of applications based on such routine.

The paper must be seen as companion paper of the fractal multiplication [6], since we can see an application of fractal matrix multiplication and optimizations for recursive matrix-multiply-like algorithms. Furthermore, we can see how fractal matrix multiplication, which is thought and implemented for square matrixes, is extended to non square matrixes. The paper is organized as follows. In Section 2 we recall Toledo's algorithm. In Section 3 we report the issues of the fractal framework such as matrix layout and recursive call optimizations which will be revisited and used in this paper. In Section 4 we introduce a model to estimate misses for LU-decomposition and we report the implementation of *Lower Triangular System* solver (LTS). We recall some considerations on fractal matrix multiply and we will see how for square matrixes Theorem 2.1 in [38] can be applied, giving an upper bound to the access complexity for LU decomposition. In Section 5 we give a detailed description of our implementation. In Section 6 we propose our experimental results describing the cache behavior of our algorithm for seven different memory hierarchies (with fixed code) and performance evaluation on a subset of four machines.

# 2 LU-decomposition with Partial Pivoting

LU-decomposition with partial pivoting for a square matrix $A$ $m \times m$ is an algorithm able to find three matrixes $P$, $L$ and $U$ such that $PA = LU$. $L$ is a $m \times m$ unitary lower triangular matrix, $U$ is an $m \times m$ upper triangular matrix and $P$ is a row permutation matrix.

**Algorithm 2.1** *(***Toledo's algorithm) [38]** *Matrix $A$ can be logically composed of four sub-blocks:*

$$A = \left| \begin{array}{cc} A_0 & A_1 \\ A_2 & A_3 \end{array} \right| \tag{1}$$

*where $A_0$ is a $\lceil m/2 \rceil \times \lceil m/2 \rceil$ square matrix. For any $M > 0$:*

1. *if $m < M$, perform pivoting and scaling and determine $P^1$, $L_0$, $L_2$ and $U_0$ so that*

$$P^1 * \left| \begin{array}{c} A_0 \\ A_2 \end{array} \right| = \left| \begin{array}{c} L_0 \\ L_2 \end{array} \right| * U_0 = \left| \begin{array}{c} L_0 U_0 \\ L_2 U_0 \end{array} \right| \tag{2}$$

2. *if $m > M$, recursively determine $P^1$, $L_0$, $L_2$ and $U_0$ so that*

$$P^1 * \left| \begin{array}{c} A_0 \\ A_2 \end{array} \right| = \left| \begin{array}{c} L_0 \\ L_2 \end{array} \right| * U_0 = \left| \begin{array}{c} L_0 U_0 \\ L_2 U_0 \end{array} \right| \tag{3}$$

3. *Apply permutation $P^1$ on the other half of the matrix:*

$$\left| \begin{array}{c} A_1 \\ A_3 \end{array} \right| = P^1 * \left| \begin{array}{c} A_1 \\ A_3 \end{array} \right| \tag{4}$$

4. *Determine $U_1$ in the lower triangular system $A_1 = L_0 * U_1$.*
5. *$A_3 -= L_2 * U_1$.*
6. *Recursively determine $P^2$, $A_3$, $L_3$ and $U_3$ so that $P^2 * A_3 = L_3 * U_3$.*
7. *permute $L_2$ by $P^2$, $L_2 = P^2 * L_2$*

*8. return* $P = P^1 \cup P^2$.

We investigate a further step in the decomposition, to exploit implementative details such as the access pattern of the computation. We need to solve recursively:

$$P * \left| \begin{array}{c} A_0 \\ A_2 \end{array} \right| = \left| \begin{array}{c} L_0 \\ L_2 \end{array} \right| * U_0 = \left| \begin{array}{c} L_0 U_0 \\ L_2 U_0 \end{array} \right| \tag{5}$$

that is

$$P * \left| \begin{array}{cc} A_{00} & A_{01} \\ A_{02} & A_{03} \\ A_{20} & A_{21} \\ A_{22} & A_{23} \end{array} \right| = \left| \begin{array}{cc} L_{00} & L_{01} \\ L_{02} & L_{03} \\ L_{20} & L_{21} \\ L_{22} & L_{23} \end{array} \right| * \left| \begin{array}{cc} U_{00} & U_{01} \\ U_{02} & U_{03} \end{array} \right| \tag{6}$$

so that we recursively determine $P^1$, $L_{00}$, $L_{02}$, $L_{20}$, $L_{22}$ and $U_{00}$ from

$$P^1 * \left| \begin{array}{c} A_{00} \\ A_{02} \\ A_{20} \\ A_{22} \end{array} \right| = \left| \begin{array}{c} L_{00} \\ L_{02} \\ L_{20} \\ L_{22} \end{array} \right| * U_{00} \tag{7}$$

Then $P^1$ is applied the the rest of the "columned" matrix.

$$\left| \begin{array}{c} A_{01} \\ A_{03} \\ A_{21} \\ A_{23} \end{array} \right| = P^1 * \left| \begin{array}{c} A_{01} \\ A_{03} \\ A_{21} \\ A_{23} \end{array} \right| \tag{8}$$

The lower triangular system $A_{01} = L_{00} * U_{01}$ can be solved and $U_{01}$ determined. The blocks permuted can be normalized using $U_{01}$.

$$\left| \begin{array}{c} A_{03} \\ A_{21} \\ A_{23} \end{array} \right| = \left| \begin{array}{c} A_{03} \\ A_{21} \\ A_{23} \end{array} \right| - \left| \begin{array}{c} A_{02} \\ A_{20} \\ A_{22} \end{array} \right| * U_{01} \tag{9}$$

Each multiplication needs two blocks stored in two different columns and we recursively determine $P^2$, $L_{03}$, $L_{21}$, $L_{23}$ and $U_{03}$ from

$$P^2 * \left| \begin{array}{c} A_{03} \\ A_{21} \\ A_{23} \end{array} \right| = \left| \begin{array}{c} L_{03} \\ L_{21} \\ L_{23} \end{array} \right| * U_{03} \tag{10}$$

The overall permutation is $P = P^1 \cup P^2$. The layout of the matrix can complicate the computations because at different level in the decomposition the subblocks of the matrix may be stored in different formats. For example in Equation 7, blocks $A_{00}$ and $A_{02}$ can be stored in row-major format but $A_{20}$ and $A_{22}$ are sub blocks in $A_2$ which is stored in row-major format.

# 3 Fractal Framework

The divide and conquer technique proposed for the solution of matrix computation can exploit in special way data locality when matrixes are in non standard layout as *fractal layout* [12, 13]. Such layout is a *recursive layout* and it is a variation from Z-Morton layout. Recursive layouts are appealing because they nullify self interference, even for caches with very low associativity. A matrix $A$ of size $m \times m$ can be composed of 4 sub-matrixes $A_0$, $A_1$, $A_2$ and $A_3$. $A_0$ is a matrix of size $\lceil m/2 \rceil \times \lceil m/2 \rceil$ where $A_0 = \{a_{ij} | 0 \leq i, j \leq \lceil m/2 \rceil\}$. $A_1$ is of size $\lceil m/2 \rceil \times \lfloor m/2 \rfloor$ where $A_1 = \{a_{ij} | 0 \leq i \leq \lceil m/2 \rceil, \lceil m/2 \rceil \leq j \leq m-1\}$. $A_2$ is of size $\lfloor m/2 \rfloor \times \lceil m/2 \rceil$ where

$A_2 = \{a_{ij} | 0 \leq j \leq \lceil m/2 \rceil, \lceil m/2 \rceil \leq i \leq m-1\}$. And $A_3$ is of size $\lfloor m/2 \rfloor \times \lfloor m/2 \rfloor$ where $A_3 = \{a_{ij} | \lceil m/2 \rceil \leq i, j \leq m-1\}$. Each $A_i$ is decomposed recursively. The projection of the matrix on a unidimensional array, layout, follows the oder of the sub-blocks such as $A_0$ is recursively stored in a contiguous interval on the linear array, then $A_1$, $A_2$ and eventually $A_3$. See Figure 1.

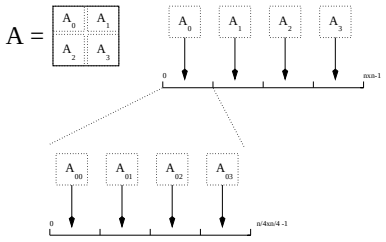



Figure 1: Matrix Layout

In [6] an auxiliary data structure has been introduced to burst up the performance of recursive algorithms for matrix multiplication: the type DAG. Noticing that, sub problems involving the same type, have the same index computation: the type dag exploits the common computation which is computed just once, off line, and can be stored around the data structure in the type dag. The type dag for a problem $< m, m, m >$ has size $O(\log m)$ and it can be initialized in $O(\log m)$ steps. Hence his compact size and complexity, the data structure is an efficient support to fast recursive matrix multiply algorithms. LU-decomposition of a matrix $A$ is the representation of matrix $A$ by the product of a lower triangular matrix $L$ and a upper triangular matrix $U$, $A = LU$. A type dag can be associated with the decomposition and it can be used not only by the fractal matrix multiplication in the LU decomposition but also in the LTS solver. In the following it will be presented implementative details of LU-decomposition.

# 4 Access Complexity of LU Decomposition

Given the size of a cache $s$ in words, the access complexity of a matrix computation $\mathcal{M}$ over matrixes of size $m$, indicated shortly as $Q_{\mathcal{M}}(s, m)$, is the number of memory accesses over the cache to the upper levels of the memory hierarchy. The access complexity describes the number of misses.

There are two basic operations that LU-decomposition recursively uses: the Lower Triangular System (LTS) solver and matrix multiply. In [6] the fractal matrix multiplication is described and an upper bound and lower bound to its access complexity is given, $Q_{MM}(s, m) = \gamma\sqrt{\frac{3}{s}}\frac{3m^3}{2}$ for $m \geq \sqrt{s/3}$. The coefficient $\gamma$ is introduced to consider the fact that caches are not ideal. The fractal matrix multiply satisfies Equation 2.2 [38] when $\gamma = 1$. We are going to describe our implementation for LTS and how it satisfies Equation 2.1 [38].

**Algorithm 4.1** *The system is $AX = B$. $A$ is a lower unitary triangular matrix of constants, $B$ is a matrix of constants and $X$ is a matrix of free variables. $X$ will be computed and stored in $B$.*

```
LTS(A,X,B) {
   if (|A|+|B| < s) solve AX=B and X is in B.
   else {
     LTS(A0,X0,B0)
     LTS(A0,X1,B1)
     B3 -= A2X1;     /* Fractal Matrix Multiply */
     B2 -= A2X0;
     LTS(A3,X2,B2)
     LTS(A3,X3,B3)
   }
}
```

**Theorem** The access complexity of LTS for square matrixes of size $m$ is $Q_{LTS}(s, m) \leq \frac{m^3}{\sqrt{s/3}} + m^2$ for $m > \sqrt{s/2}$.

$\diamondsuit$

**Proof.** We can see the recurrence equations for the access complexity when $m$ is an integer power of two: $Q_{LTS}(s,m) \leq 4^i Q_{LTS}(s, \frac{m}{2^i}) + \sum_{j=0}^{i-1} 4^j 2 Q_{MM}(s, \frac{m}{2^{j+1}})$ and we stop when $i = 1/2 \log_2(2m^2/s)$ achieving $Q_{LTS}(s,m) \leq \frac{2m^2}{s} Q_{LTS}(s, sqrts/2) + \gamma \frac{3\sqrt{3}m^3}{2\sqrt{s}}[1 - \sqrt{\frac{s}{2m^2}}]$. At leaf level there is locality exploited among LTSs and matrix multiplications so that $Q_{LTS}(s, sqrts/2) \leq 3s/4$. For $n > \sqrt{s/2}$ the solution to the equation is $3m^2/4 + \gamma \frac{3m^3}{4\sqrt{s/3}}(1 - \sqrt{\frac{s}{2m^2}})$. Ideally, $\gamma = 1$ and looking the coefficient values the theorem holds. ♢

Since the fractal LTS satisfies Equation 2.1 [38], for the implementation of LU-decomposition we have: $Q_{LU}(s,m) \leq \zeta[2m^2(\frac{m}{2\sqrt{s/3}} + \log_2 m) + 2m^2(1 + \log_2 m)]$. Where a constant $\zeta \geq 0$ can be introduced due to the non ideal memory hierarchy. We define as $\mu(m) = Q_{LU}(s,m)/(\frac{2}{3}m^3)$ the number of miss per FLOP.

# 5 Implementation

Given a matrix $A$ of size $m \times m$ we can determine the type DAG describing the computation $A = LU$. We recursively decompose the matrix so that we achieve tiles smaller than $k \times k$. We decompose recursively the problem and we solve any sub-problem of size smaller than $m \times k$. The decomposition might be fallacious, when during a column computation some matrix blocks can be stored by different layouts. We propose the implementation in Figure 2. The algorithm describes the steps required when the recursion stops and how the

```
/*****************************************************************
* A is a matrix composed of A0, A1, A2 and A3
* min, max      LU decomposition from column to column            */
 1   BOOLEAN LU(A,min,max) {
 2       width = max - min;
 3       IF (width <= END_FRACTAL_LAYOUT) {
 4          flag = MAX(A,min);
 5          IF (flag) PERMUTE_ROWS();
 6          FOR i=0 TO width-1 {
 7               IF (pivot=ZERO) return FALSE;  /* NO LU-decomposition*/
 8               flag = GELM(A,min+i);
 9               IF (flag) PERMUTE_ROWS();
10          }
11        ELSE {
12          IF ( NOT LU(A,min,max-width/2)) retun 0;
13          PERMUTE(A1,A2);
14          LTS_and_GEMM(A,min,max);
15          IF (max = ColumnOf(A)) return LU(A3,0,max/2);
16          ELSE return LU(A,max-width/2, max);
17       }
18     return 1;
19  }
```

Figure 2: LU decomposition

problem is decomposed recursively. When the recursion stops the search of the pivot and Gaussian elimination (line 3-10) "*work*" on column of unitary width and permutations are performed on rows of length as sub-block size (line 5 and 9). In the recursive decomposition of the problem both scaling and local permutation are done on sub-blocks (line 13 and 14). Due to the column-wise computation in the LU-decomposition, the recursive decomposition must be guided (line 15-16). Indeed, solving $A_3 = L_3 U_3$, the computation accesses only sub-matrix $A_3$. The computation at the leaves follows the following order:

1. pivot searching, search of the maximum element in absolute value,
2. permutation of two partial rows,
3. Gaussian elimination and scaling of the sub-blocks.

This order of the computation may requires three distinct visits of the column, which is not efficient. At the beginning of the computation we determine the first pivot and we permute the rows. During the Gaussian elimination we can search the next pivot element and, when we have computed the new elements in the column, we can permute the required rows. The visits of columns are reduced in number. Computations are mostly on blocks.

The type DAG describing the matrix multiply $A = LU$ can be used to access matrix $A$ and help determining *column computations*. The column computation can be described as matrix multiply $A_{\mathbf{x}} = L_{\mathbf{x}} * U_{x_0}$ where $\mathbf{x}$ is an ordered set of indexes, $x_0 \in \mathbf{x}$ and first element in $\mathbf{x}$. The indexes for the column computation $A_{\mathbf{x}} = L_{\mathbf{x}} * U_{x_0}$ are determined in two distinct steps.

1. A logical column identifies the first elements of the computations $A_{x_0} = L_{x_0} * U_{x_0}$. Two possible choices are available at any time: the column identifies either $A_0 = L_0 U_0$ or $A_3 = L_3 U_3$. This is a binary search and each comparison result can be memorized and the whole path can be traced.

2. The type DAG is visited column wise. The trace permits to visit the type DAG in a specified direction. When a leaf is visited the search goes up the binary search tree throught the trace and determines the next sub-tree, then the next leaf.

The descending phase on the type DAG $< m, m, m >$ takes $O(\log m)$ steps but the overall search for the column (row) computation takes $O(m)$ steps. In average, the complexity per access is constant.

The computation, $A_x = L_x * U_y$, is associated with a node at any level in type DAG. Type and size of a node specify computation and data lay out of the operands. If we perform Gaussian elimination, so that matrixes $L_x$ and $U_y$ are determined on $A_x$, the search in the type DAG returns a leaf (respectively, an internal node). Indeed, a leaf in the matrix multiply can be a particular case and therefore the operands can be stored in mixed formats. If we indicate with $k$ the integer value so that every sub matrix of size smaller than $k \times k$ is stored in row-major format we can enumerate the following cases:

1. $A_x = L_x = U_x$, the matrixes coincide, and they are in fractal format, i.e. the problem is $< m, n = m, p = m >$ and $m \geq k$.

2. $A_x = L_x = U_x$, the matrixes coincide, and they are in row-major format, i.e. the problem is $< m, n = m, p = m >$ and $m < k$.

3. $A_x = L_x$ is in fractal format and $U_y$ is in row-major format, i.e. the problem is $< m, n = p, p >$ with $m = p + 1$ and $p < k$ .

4. $A_x = L_x$ is in row-major format but $U_y$ is in fractal format, i.e. the problem is $< m, n = p, p >$ with $m + 1 = p$ and $p = k$.

The procedures must use the information available so that they can access correctly the elements in the sub matrixes.

## 5.1 Gaussian Elimination Routine

When the pivot is determined and the permutation took place, the row determined is called *normailizing row*. Gaussian elimination subtracts to the following rows the normalizing row multiplied by a proper scalar value. Gaussian elimination and scaling can be logically *divided* in two steps (in Figure 3 we can see a possible access pattern on different stages in the computation).

1. on $A_x = L_x * U_x$: the normalizing row is in the block, the computation always accesses a square sub matrix of $A$.

2. on $A_y = L_y * U_x$: the normalizing row is not in the block and the computation always accesses a *columned* sub-matrix of $A$.

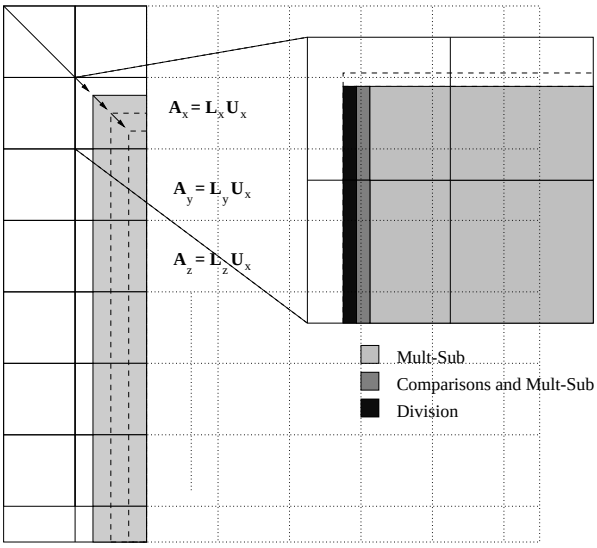



Figure 3: Visualization of the access pattern when successive column Gaussian eliminations are performed.

In every column we elaborate only part of each block. On the right side of Figure 3 we can see, through a zoom effect, the operations performed in the first block. Computations in different rows are independent. In each row the division computes the scalar value to multiply by the normalizing row. Division for double precision floating points is a very expensive operation. It is often computed by a non pipelined functional point unit with long latency which does not stall the microprocessor (i.e. SparcUltraIIi has a non blocking division functional point unit, its latency is function of the type of operands and for double is 22 cycles). We can hide division latency exploiting the parallelism among rows. The division at the first row cannot be hidden, but we can issue the division of the second row as soon as the first division is committed and while the operations in the first row are not committed yet. There are three possible cases that we can find summarized in Figure 4. The matrix is

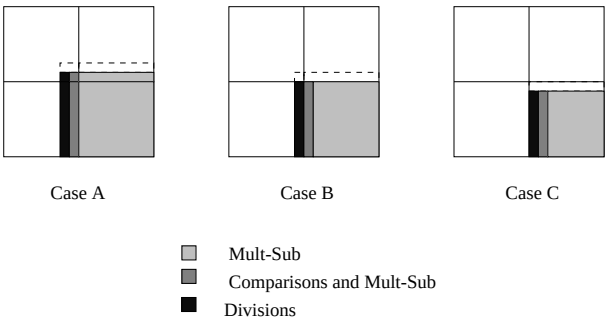



Figure 4: Case A: The first quadrant has all types of computations. Case B: first quadrant has only a pivot element. Case C: the computation is done only in the fourth quadrant

composed of four blocks in row-major format.

**Case A:** we perform the normalization of the rows in every quadrant.

**Case B:** we perform the normalization only in the fourth quadrant and divisions only in the third.

**Case C:** the computation involves only the fourth quadrant.

If the matrix is a single block in row major format, the case C describes the computation. The computation to solve other blocks, described by equation $A_y = L_y * A_x$, involves columned sub matrix of $A$ and if the sub block is in fractal format there are again three cases as we can find in Figure 5.

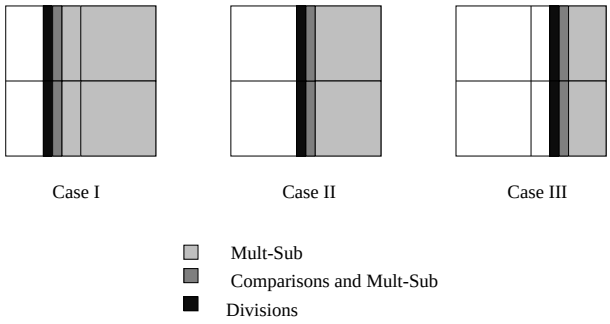



Figure 5: Case I: in all four quadrants there are multiplication-subtractions. Case II: divisions are performed in the first and third quadrant and Multiplication-subtractions in the others. Case III: the computation is limited in the second and fourth quadrants.

### 5.2 Permutation

Due to the generality of the problem, we relax the requirement to use the type DAG. We use a *sub structure* of the type DAG: *data layout DAG*[1]. It describes recursively the data lay out of a matrix stored in fractal format. A definition of the structure follows:

```
typedef struct node_structure_layout NSL;
struct node_structure_layout {
  NSL *sons[4];      /* links to the sub-blocks A0, A1, A2 and A3   */
  int m,n;           /* size of A */
  int di[3];         /* offset of A1, A2, A3 with respect to A   */
};
```

Conceptually, permutations on a fractal matrix using the data layout dag involve similar algorithms to the column computations previously described:

1. for each row to swap, we determine the location of the first row element in the matrix by a binary search. Each decision taken during the search is store and the path, in the data layout DAG, is traced;
2. once a swap is performed in a sub-block associated with a leaf, the traced path is visited and modified properly so that the *neighbor* is visited.

The descending phase on the data layout DAG $< m, m >$ takes $O(\log m)$ steps but the overall search for the row permutation takes $O(m)$ steps. We can say that row permutation on a fractal matrix is just by a constant slower than a row permutation on a row-major matrix.

### 5.3 Column Normalization

When the Gaussian elimination has been performed in a column we need to compute a normalization of the following column (i.e. Equation 9). By construction, the column computation can be split in two parts. In the first part it is computed a multiple right side system and a multiply-subtraction. In the second part, if the column comprises more than one block, it is computed only multiply-subtractions. This operation is a non square matrix multiply. The type DAG introduced for square matrix multiplication is used to perform a more general matrix multiply.

## 6 Experimental Result

The presentation of the experimental results follows the following criterion: given an estimation of the upper bound to the access complexity we compute its variance from experimental results obtained by simulation, then we compare the performance on real architecture w.r.t. peak performance and, if available, ATLAS or vendor LU-decomposition. We consider only matrixes in double precision.

[1] Quad Tree

## 6.1 Cache Miss

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

Table 1: Summary of simulated configurations

| Simulated Configuration | Size (Bytes/$s$) | Line (Bytes,$\ell$) | Associativity/ Write Policy | $\zeta(1300)$ |
|---|---|---|---|---|
| SPARC 1 | | | | |
| U1 | 64KB / 8K | 16B / 2 | 1 / through | 0.249 |
| SPARC 5 | | | | |
| I1 | 16KB | 16B | 1 / | |
| D1 | 8KB / 1K | 16B / 2 | 1 / through | 0.311 |
| Ultra 5 | | | | |
| I1 | 16KB | 32B | 2 / | |
| D1 | 16KB / 2K | 32B / 4 | 1 / through | 0.285 |
| U2 | 2MB / 256K | 64B / 8 | 1 / back | 0.020 |
| R5000 IP32 | | | | |
| I1 | 32KB | 32B | 2 / back | |
| D1 | 32KB / 4K | 32B / 4 | 2 / back | 0.042 |
| U2 | 512KB / 64K | 32B / 4 | 1 / back | 0.085 |
| Pentium II | | | | |
| I1 | 16KB | 32B | 1 / | |
| D1 | 16KB / 2K | 32B / 4 | 1 / through | 0.181 |
| U2 | 512KB / 64K | 32B / 4 | 1 / back | 0.063 |
| HAL Station | | | | |
| I1 | 128KB | 128B | 4 / back | |
| D1 | 128KB / 16K | 128B / 16 | 4 / back | 0.020 |
| ALPHA 21164 | | | | |
| I1 | 8KB | 32B | 1 / | |
| D1 | 8KB / 1K | 32B / 4 | 1 / through | 0.212 |
| U2 | 96KB / 12K | 32B / 4 | 3 / back | 0.077 |

leaves is strictly smaller than problem $< 32, 32, 32 >$; (ii) the recursive layout is stopped when a sub-matrix is strictly smaller than $32 \times 32$; (iii) the matrix multiply 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 are compiled with gcc 2.95.1 (same test bed has been used in [6]).

We have also simulated the code for Sun Performance Library available as standard library for WorkShop compiler. 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 Sun Performance. In Table 1 we summarize the 7 memory hierarchies. We use the *"Shade"* notation: I= Instruction cache, D=Data cache, U=Unified cache and L=any cache at a level. In the last column in Table 1 we computed the value of $\zeta$ when matrix has size 1300. The coefficient shortly represents the discrepancy between the upper bound and the experimental results: due to the cache line effect, the cache associativity of each cache level, the hierarchy and the over estimation of the accesss complexity. We can notice the following issues due to the memory hierarchy configuration: 1) the cache line size $\ell \in \{2, 4, 8, 16\}$ for direct mapped cache has an (inverse) proportional effect; 2) cache line $\ell$ and cache associativity $\alpha$ have more than multiplicative effect $(\zeta/(\ell\alpha))$; 3) for the second level of cache the overall effect is superlinear, only exception for R5000, $(\zeta/(\alpha_1\alpha_2\ell_1\ell_2))$. From Fig. 6 to 12 we did report $\mu(m)$, that is, the number of misses per FLOP. We reported the number of miss of the data and of the code. Data miss comprises the data read and data write misses. The fractal approach, which is based on the recursive decomposition and

static link to different leaf computations, has the side effect to have a large number of code miss; at the same time, due to the finer decomposition, the fractal approach has a fewer number of data miss. In average the Fractal approach is exploiting the different memory hierarchies, offering a few number of misses.

### 6.2 Running Time

We compare the Fractal performance with the best known vendor LU decomposition, with ATLAS based LU decomposition or with peak performance (when no best algorithm is known). $C$-tiling register assignment is applied. Four architectures has been tested as we can find in Table 2. The running time can be split in two

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 / latency | distinct / 3 | distinct / 8 | single FU / 2 | single FU /4 |
| Divider / latency | distinct / 22 | distinct / N/A | distinct / N/A | distinct / 8 |
| Peak (MFLOPS) | 666 | 400 | 360 | 200 |
| Peak Fractal / matrix size | 352 / $3000 \times 3000$ | 138 /$2800 \times 2800$ | 112 / $2500 \times 2500$ | 158 / $2048 \times 2048$ |
| Peak ATLAS / matrix size | 260 / N/A | 210 / N/A | N/A | N/A |

contributions: $T(m) = T_M(m,t) + T_G(m,t)$ where matrix $A$ is a square matrix $m \times m$, $A$ is recursively composed of tiles smaller than $s \times s$ (also called blocking factor), $T_M$ is the running time executing matrix multiplication and $T_G$ is the running time for Gaussian elimination. In [6] we investigate the behavior of matrix multiply in function of $s$. The bigger $s$ the better is the matrix multiply because the reuse of values ($s$ cannot be larger than a threshold value where performance collapses due to poor utilization of L1). But the bigger $s$ the smaller is the number of times matrix multiply is called and the bigger is the running time $T_G$. An optimal parameter $s$ for matrix multiply may be non optimal for LU-decomposition. As proof of the previous observation, but not reported here, we increased the bigger tile size from $32 \times 32$ to $48 \times 48$, which is optimal for fractal matrix multiplication. The performance got worse. By profiling, we could measure that the improvements of $T_M$ were smaller than the worsenings of $T_G$. This explains why ATLAS based LU-decomposition is not optimal: they are using blocking factor greater than $32 \times 32$ for every architectures. The only exception is for PentiumII processor: because fractal matrix multiplication for pentium is two times slower than ATLAS matrix multiply, such a gap cannot be overcome. We can see that even if fratal multiply is not the fastest matrix multiplication, the optimizations, introduced for the first time in the fractal framework, can be applied for LU-decomposition and they are very beneficial to the overall preformance.

## 7 Conclusion

In this paper we presented an evaluation for cache and run time performance of a basic routine such as LU decomposition with partial pivoting, and therefore for the solution of triangular systems. By simulation we verified the cache behavior of Toledo's algorithm on several memory hierarchies and we compared running time against other LU-decomposition algorithms. From this experience we have achieved the following results: the fractal framework and optimizations introduced for fractal matrix mutiplication are also suitable for LU decomposition; in general, we have obtained the fastest LU decompostion algorithm using standard matrix multiplication routine; the cache behavior of fractal decomposition is very much portable w.r.t. different memory hierarchy; high performance is achieved it is very much portable across different architectures. The fact that LU-decomposition has not good performance for PentiumII is due to the different register allocation for the matrix multiplication. ATLAS can exploit better the stack register file structure and the floating point structure

of PentiumII. The gap cannot be overcome by the fractal approach.

# 8 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] E.Anderson, Z.Bai, C.Bischof, J.Demmel, J.Dongarra, J.DuCroz, A.Greenbaum, S.Hammarling, A.McKenney, S. Ostrouchov and D.Sorensen: LAPACK User' Guide, Release 2.0, 2nd ed., SIAM Publications, Philadelphia.

[5] U.Banerjee, R.Eigenmann, A.Nicolau and D.Padua: Automatic program parallelization. Proceedings of the IEEE vol 81, n.2 Feb. 1993.

[6] G.Bilardi, P.D'Alberto and A.Nicolau: Fractal Matrix Multiplication: a Case Study on Portability of Cache Performance; *manuscript*.

[7] 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.

[8] G.Bilardi and E. Peserico: Efficient portability across memory hierarchies. Manuscript, March 2000.

[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 Proceedings of 9th annual ACM Symposium on Theory of Computing pag.1-6, 1987.

[15] 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

[16] N.Eiron, M.Rodeh and I.Steinwarts: Matrix multiplication: a case study of algorithm engineering. Proceedings WAE'98, Saarbrücken, Germany, Aug.20-22, 1998

[17] Engineering and Scientific Subroutine Library. `http://www.rs6000.ibm.com/resource/aix_resource/sp_books/essl`,

[18] P.Flajolet, G.Gonnet, C.Puech and J.M.Robson: The analysis of multidimensional searching in Quad-Tree. Proceeding of the second Annual ACM-SIAM symposium on Discrete Algorithms, San Francisco, 1991, pag.100-109.

[19] 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.

[20] M.Frigo and S.G.Johnson: The fastest Fourier transform in the west. MIT-LCS-TR-728 Massachusetts Institute of technology, Sep. 11 1997.

[21] M. Frigo, C.E. Leiserson, H. Prokop and S. Ramachandran: Cache-oblivious algorithms. Proc. 40th Annual Symposium on Foundations of Computer Science, (1999)

[22] 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.

[23] G.H.Golub and C.F.van Loan: Matrix computations. Johns Hopkins editor 3-rd edition

[24] 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

[25] 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.

[26] J.L.Hennesy and D.A.Patterson: Computer Architecture a Quantitative Approach; Morgan Kaufman 2nd ed. 1996.

[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] M.Marcus: Matrices and MATLAB: A Tutorial, Prentice Hall, Upper Saddle River, NY 1993.

[33] S.S.Muchnick: Advanced compiler design implementation. Morgan Kaufman

[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] B.T.Smith, J.M.Boyle, Y.Ikebe, V.C.Klema and C.B.Moler: Matrix Eigensystem Routine: EISPACK Guide, 2nd ed., Lecture Notes in Computer Science, 1970, Volume 6, Springer-Verlag, New York.

[37] V.Strassen: Gaussian elimination is not optimal. Numerische Mathematik 14(3):354-356, 1969.

[38] 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

[39] 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).

[40] R.C.Whaley and J.J.Dongarra: Automatically Tuned Linear Algebra Software. http://www.netlib.org/atlas/index.html

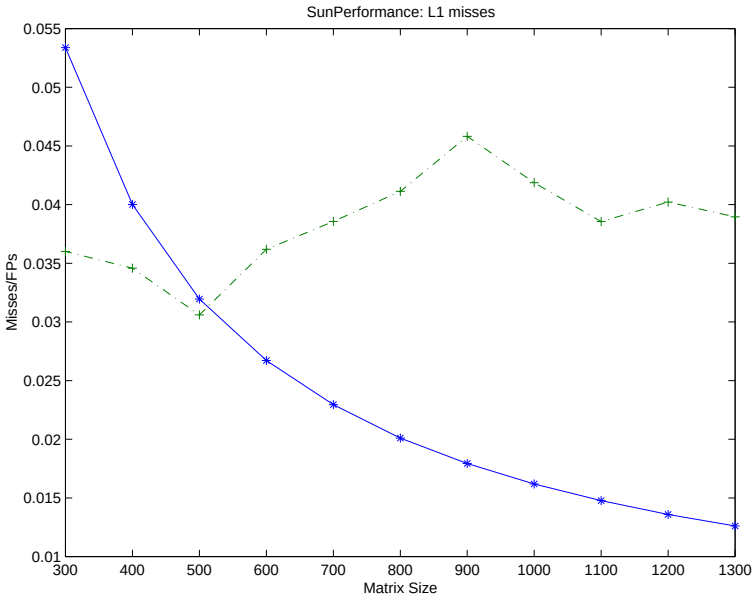
[41] J.H.Wilkinson and C.Reinsch: Handbook for Automatic Computation, Vol. 2, Linear Algebra, Springer-Verlag, New York.

[42] D.S.Wise: Undulant-block elimination and integer-preserving matrix inversion. Technical Report 418 Computer Science Department Indiana University August 1995

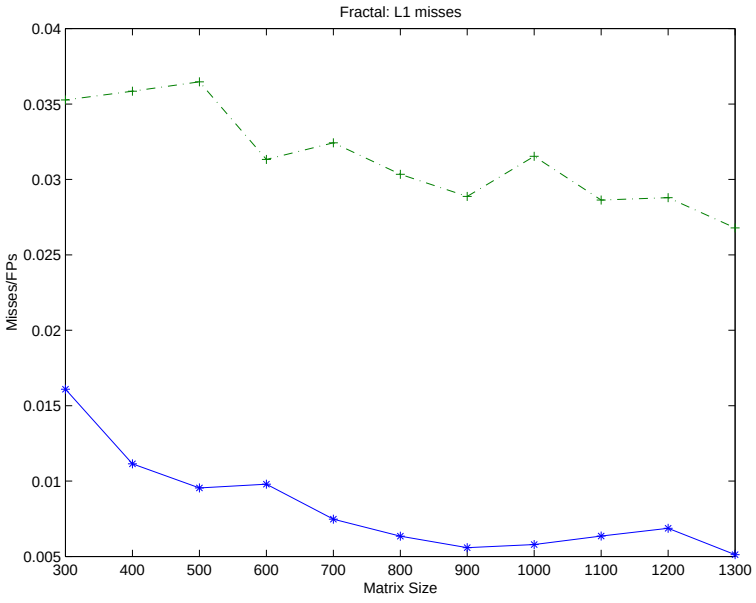
[43] M.Wolfe: More iteration space tiling. Proceedings of Supercomputing, Nov.1989, pg. 655-665.

[44] 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.

[45] M.Wolfe: High performance compilers for parallel computing. Addison-Wesley Pub.Co.1995

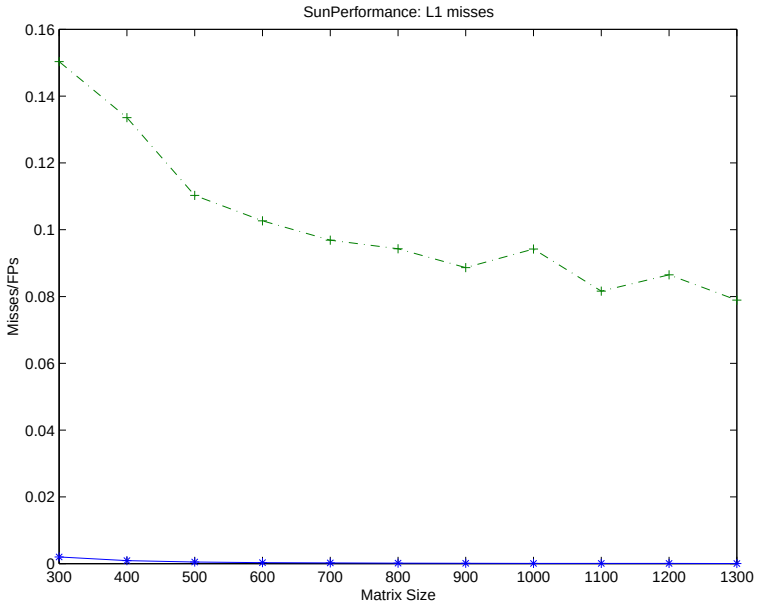



**SunPerformance**

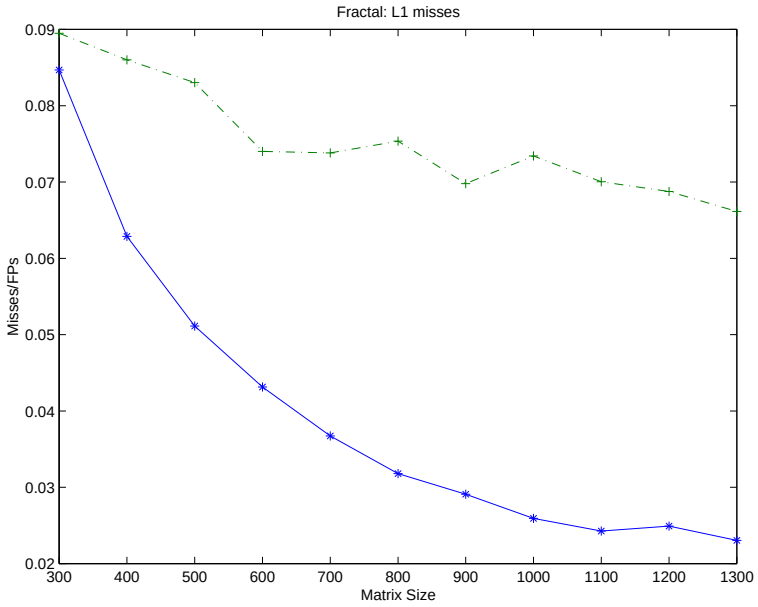



**Fractal**

Figure 6: **SPARC 1, * code misses, + data misses**. The algorithms in SunPerformance and Fractal have been simulated with matrixes of sizes between $300 \times 300$ and $1300 \times 1300$. Fractal has better number of misses and surprisingly a very small number of code misses.




**SunPerformance**




**Fractal**

Figure 7: **SPARC 5, * code misses, + data misses**. SunPerformance's application has a very good code locality but poor data locality, it never goes under 0.16. Fractal approach has a very good data locality, it is always under 0.16, and contains the number of code misses.

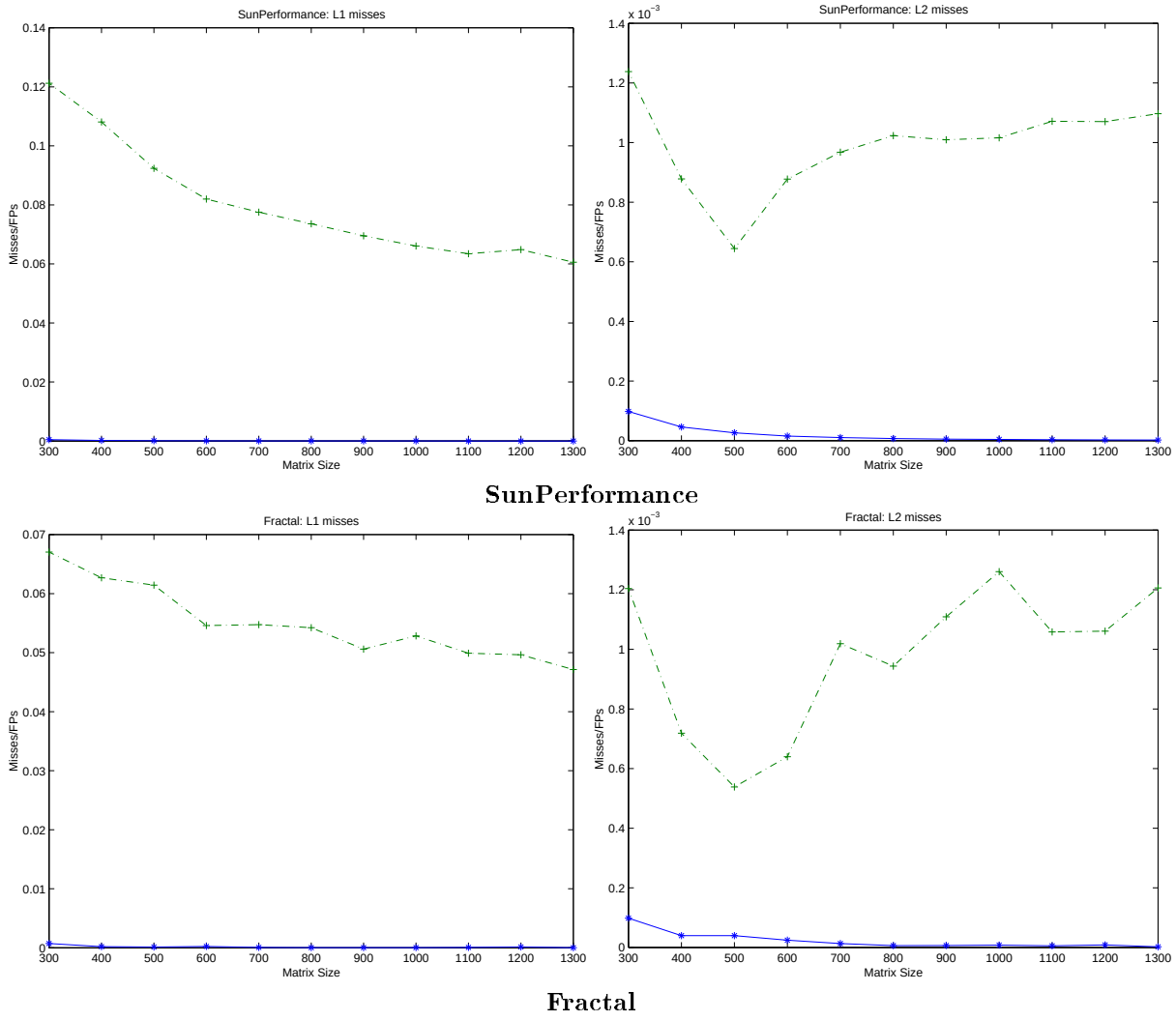
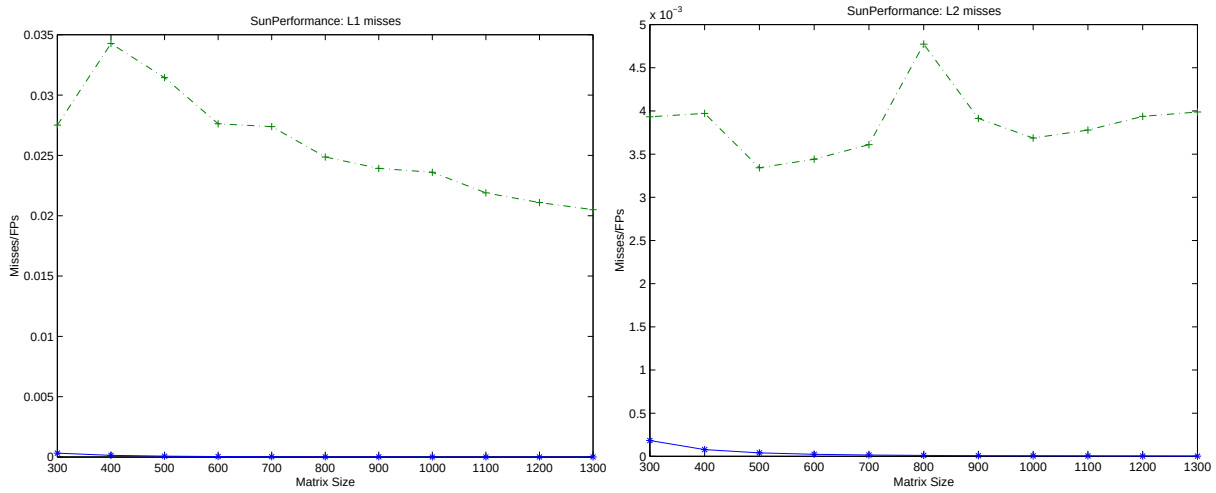
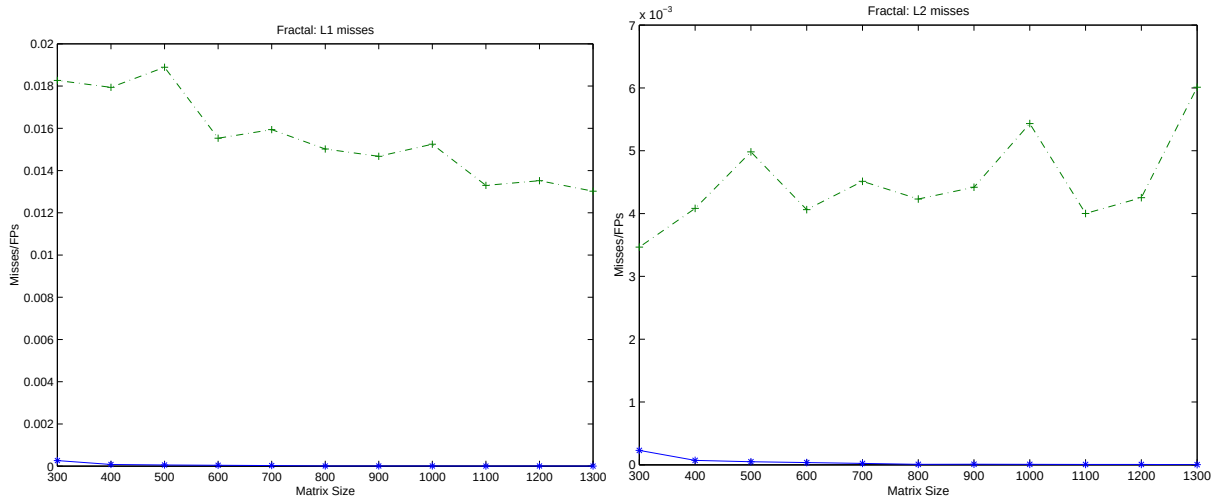



Figure 8: **Ultra 5, * code misses, + data misses**. The algorithms SUnPerformance and Fractal have been simulated with matrixes of sizes between $300 \times 300$ and $1300 \times 1300$. On the left side we can find the characterization of L1 and on the right the characterization of L2. Both algorithms are designed with particular attention to this architecture. Fractal has smaller number of miss at the first level of cache (almost a half of the number of miss of SunPerformance), and comparable behavior at the second level of cache.

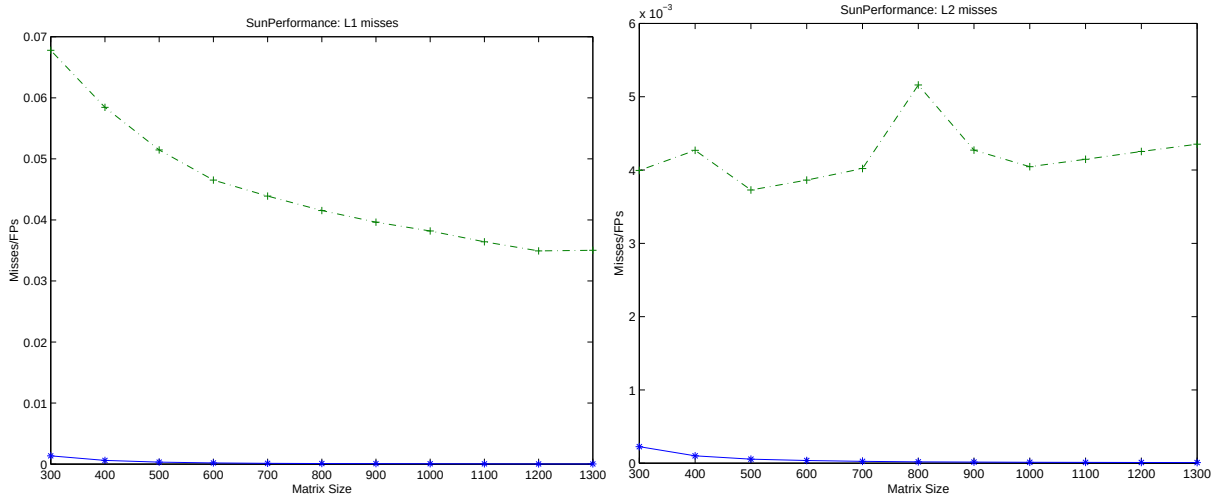



**SunPerformance**

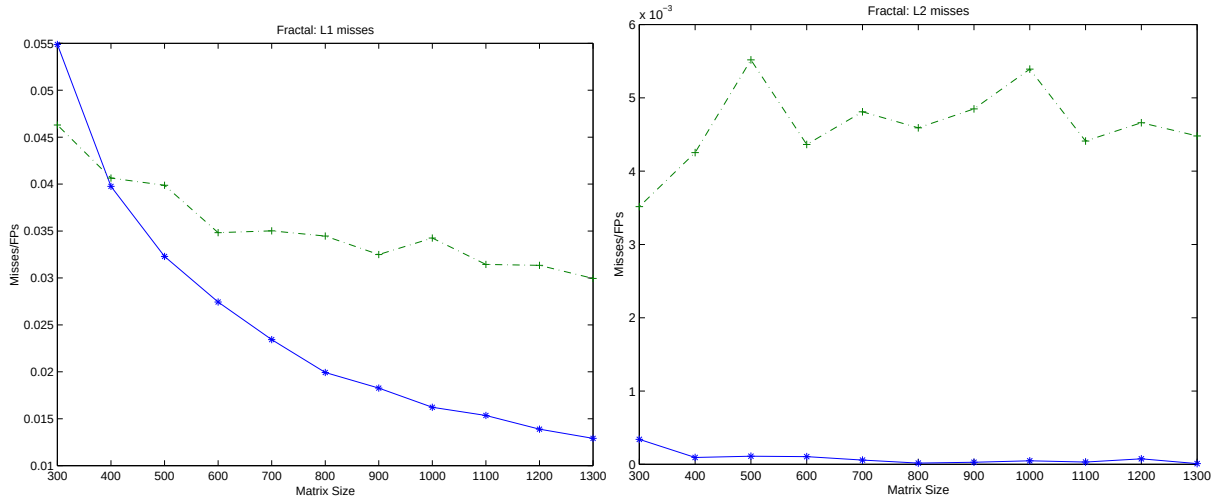



**Fractal**

Figure 9: **R5000, * code misses, + data misses**. The algorithms SunPerformance and Fractal have been simulated with matrixes of sizes between $300 \times 300$ and $1300 \times 1300$. On the left side we can find the characterization of L1 and on the right the characterization of L2. Fractal algorithm has in general a better cache performance than SunPerformance's application. At the first level Fractal has almost a half of the number of miss than SunPerformance's application has got, and the cache behavior at the second level is about the same for both applications.

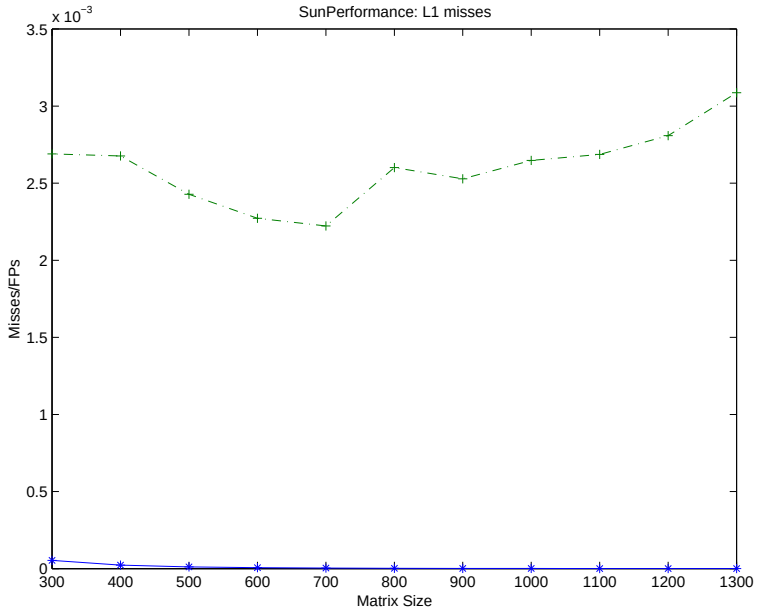



**SunPerformance**

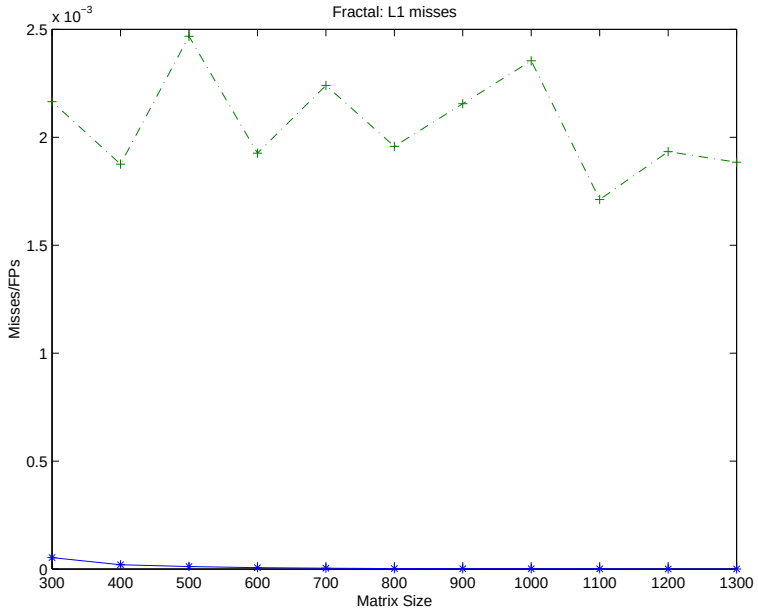



**Fractal**

Figure 10: **Pentium II, * code misses, + data misses**. The algorithms SUnPerformance and Fractal have been simulated with matrixes of sizes between $300 \times 300$ and $1300 \times 1300$. On the left side we can find the characterization of L1 and on the right the characterization of L2. At the first level of cache fractal performs better but at the second level of cache SunPerformance's application performs in average better.

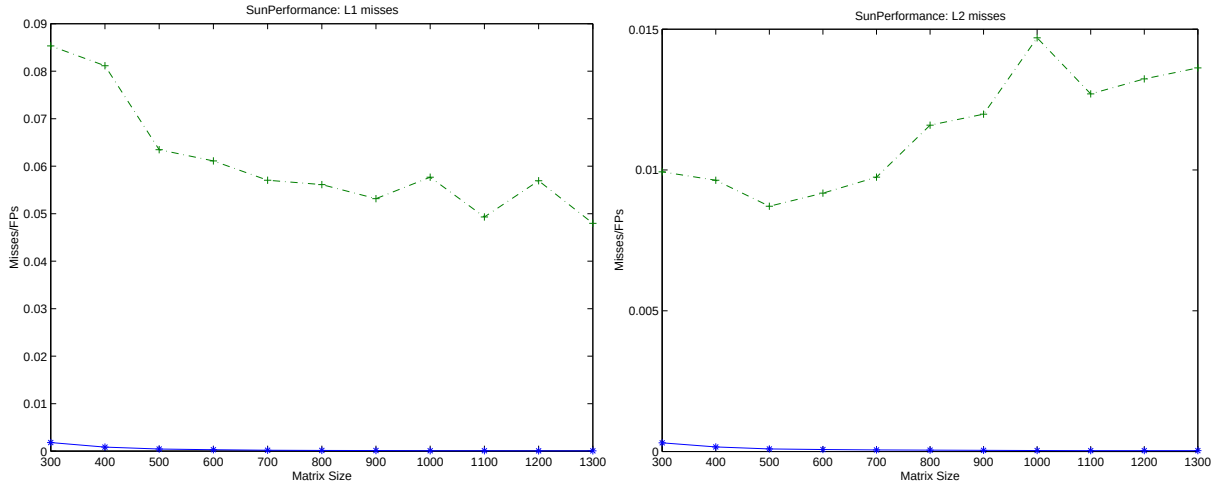



**SunPerformance**

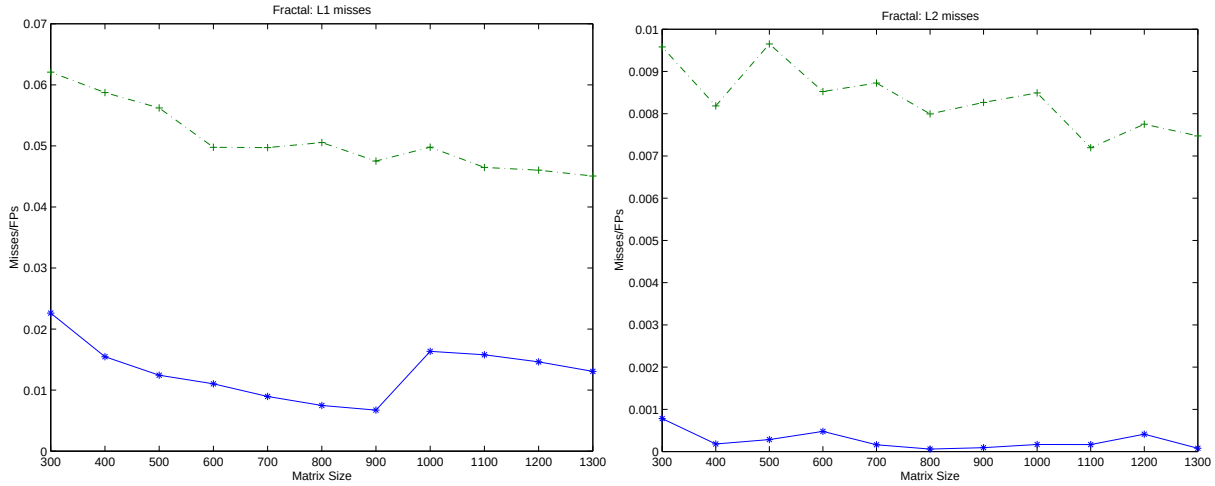



**Fractal**

Figure 11: **SPARC64, * code misses, + data misses**. The algorithms SunPerformance and Fractal have been simulated with matrixes of sizes between $300 \times 300$ and $1300 \times 1300$. In this architecture the code misses are just compulsory misses and Fractal is always better than SunPerformance.




SunPerformance




Fractal

Figure 12: **Alpha 21164 architecture, * code misses, + data misses**. The algorithms SunPerformance and Fractal have been simulated with matrixes of sizes between $300 \times 300$ and $1300 \times 1300$. On the left side we can find the characterization of L1 and on the right the characterization of L2. The Fractal has a better data locality at every level of the memory hierarchy but it has a poor code locality at the first level of the memory hierarchy.

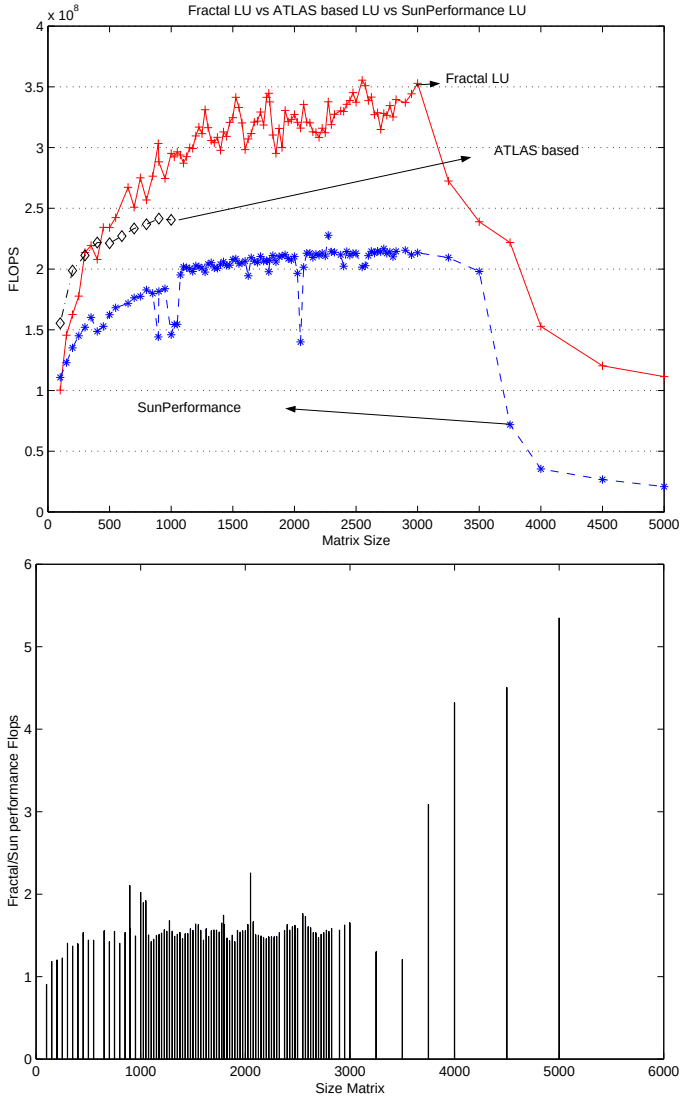



Figure 13: **Ultra 5**. (*) SunPerformance , (+) Fractal and ($\diamond$) some samples of ATLAS based LU decomposition . ATLAS based LU decomposition results are based from the experimental results proposed by Dongarra and Whaley for the Ultra Sparc 2-200. We measured the peak performance of ATLAS matrix multiply on Ultra 5 with matrixes greater than $200 \times 200$. We computed the ratio of the measured peak value over the peak performance on Sparc Ultra 2-200. Since the value is smaller than $\rho = 333/200$, we interpolated the performance of LU Decomposition on Ultra 5 multiplying by $\rho$ the experimental results from Sparc Ultra 2-200. In the first figure we can see the FLOPS: the best performance is 360MFLOPS which is very close to the average performance of the Fractal matrix multiply. Since matrix multiply is the basic operation in the decomposition, better performance cannot be achieved. In the second figure we can find the speedup of LU decomposition of Fractal respect SunPerformance. The average speed up is of 1.5, with peak of 2.0. For big matrixes when the disk must be used we can see how the fractal approach is taking full advantage of its space locality. Fractal LU-Decomposition is faster than ATLAS based LU decomposition. ATLAS-based approach decompose the matrix mostly in tiles of size $36 \times 36$, optimizing the matrix multiplication performance. Bigger the tile, less multiplications the LU-decomposition performs. Performance is less bound to matrix multiplication. Fractal decomposes the matrix on tiles up to $32 \times 32$, a finer decomposition. The overall performance is better.

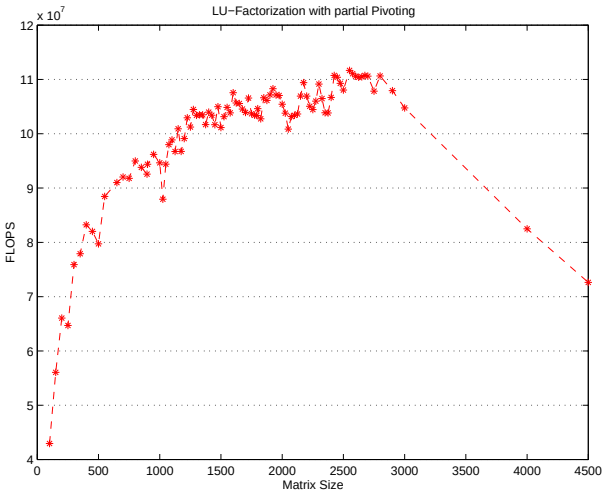



Figure 14: **R5000**. Only the Fractal LU-decomposition is reported. As we can see, the peak performance is 110MFLOPS which is very close to the peak performance of fractal matrix multiply. Even if there is space for improvements, we cannot do it better unless we improve performance of matrix multiply. For this particular architecture fractal matrix multiply is the best one.

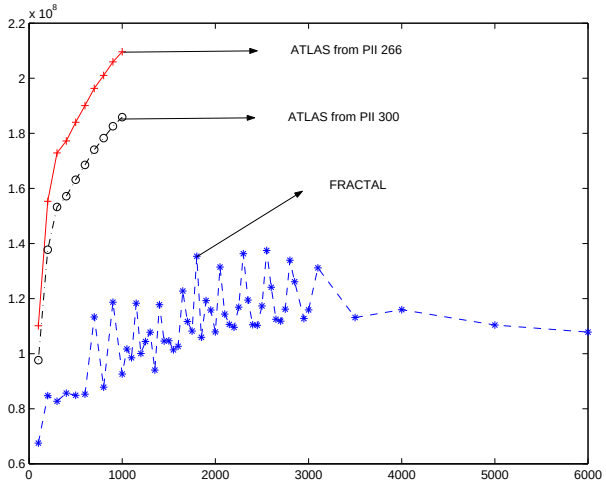



Figure 15: **PentiumII.** Square matrix elements are in double precision. ATLAS based LU decomposition experimental results are inferred from the experimental results proposed by Dongarra and Whaley for the architecture Pentium II-300MHz. We determined the peak performance of matrix multiply on Pentium II-450MHz and we determined the ratio over the peak performance for Pentium II 300MHz. Since the ratio is smaller than $\rho = 450/300$, we inferred the LU-decomposition experimental results for Pentium II-450 multiplying by $\rho$ the experimental results from PentiumII. Since in the paper is not specified if the architecture where LU-decomposition has been tested is either PII-266 or PII-300, we reported both interpretations. The ATLAS LU decomposition is two times faster than fractal LU-decomposition. This is because ATLAS matrix multiply is twice faster than Fractal matrix multiply.

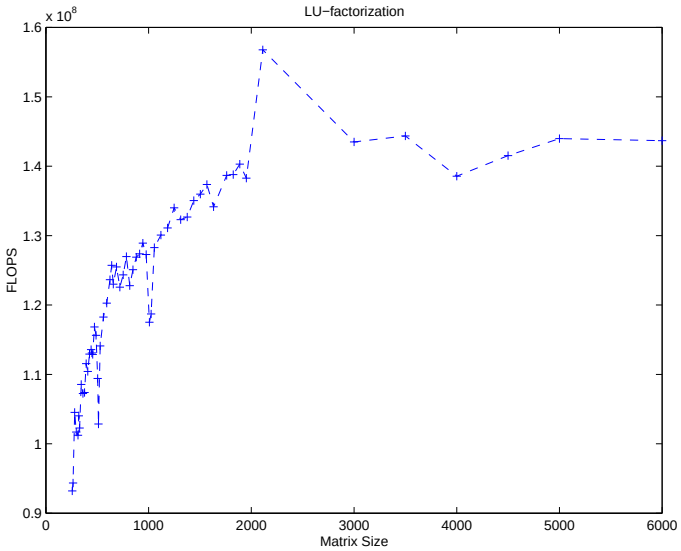



Figure 16: **SPARC64.** This architecture is under utilized by the compiler which performs a really poor register allocation but the performance for LU-decomposition is outstanding. Peak performance is almost as matrix multiplication performance. Even for very large matrixes, when the matrix does not fit the main memory and the disk is used as temporary swap, the performance is 3/4 the peak performance and in general is faster than any architecture described previously.