# A comparative study of the Itanium 64 bits and IA-32 architectures for solving scientific problems

**Project of the Computer Science Department, School of Sciences, University of Buenos Aires, Argentina.**

**Directors**: Darío Robak and Hugo D.Scolnik
**Members**: Martín Degrati, Xavier Ruiz Ibáñez and Ximena Pisani

November 2004

# 1 Abstract

We study in this work how some features of the Itanium 2 processor can be successfully used for scientific applications.

The first step was to determine the best way to compile C programs in order to achieve the best performance or precision in the computations. For that purpose we wrote different scripts based on the compiler's available documentation that automatically search for the optimal combinations of compiling options. The conclusion was the Intel "icc" compiler offers the best results. Moreover, when using the PGO (Profile Guided Optimization) options provided by that compiler and the Itanium processor, the CPU times obtained were the least when compared with all other compilers.

We also studied the real advantages of using the combined multiply-add operation (FMA) in regard to other architectures lacking this option. That operation computes $f_1 = f_3 * f_4 + f_2$ in a single step. We describe in what floating-point calculations the use of this operation gives a higher precision, characterizing the rounding error effects and their implications.

In particular, we show the correlations appearing among the *f2, f3 y f4* values in those cases where this operation brings the higher benefits. Those correlations are valid in different algorithms, as for example the range reduction for computing sines or in improving the precision of a quotient, which was approximately calculated.

We show that the floating-point registers of the Itanium processor provide unique and extremely useful characteristics, no yet fully exploited by today's compilers. We explain what those advantages are and give recommendations about how they can be used.
The main benefit arises from the use of floating point registers with extended range that use 17 bits for the exponent instead of 15, without affecting performance. This feature diminishes the likelihood of facing underflow/overflow conditions. However, there is no suitable way of controlling this capability from the C, C++ or Fortran compilers. We believe that this shortcoming is a consequence of this feature not being available in other architectures. Therefore, it would be very useful for the scientific community to have compilers and libraries supporting registers with that extended exponent. Along this direction, we suggest ways of achieving that goal.

When using quadruple precision, the results were remarkable. The compared architectures do not offer hardware support for that precision, and therefore it has to be implemented by software. As expected, the precision was similar since the emulation follows the IEEE specs for these sorts of data types. On the other hand the features offered by the Itanium processor as the use of the internal parallelism, the higher number of registers, the operations of division implemented by software and fused multiply-add, led too much better CPU times. We do believe that this is a real example where the Itanium architecture offers outstanding benefits

This study as a whole shows the flexibility of the architecture that allows implementing scientific applications with higher performance and precision.

**Table of contents**


1 Abstract .......... 2
2 Introduction .......... 5
3 Compilers .......... 6
- 3.1 Tests .......... 6
- 3.2 Results .......... 8
- 3.3 Analysis of the results .......... 10

4 Multiply-add fused operation .......... 11
- 4.1 Introduction .......... 11
- 4.2 The FMA operation .......... 11
- 4.3 Benefits and uses of the fused operation .......... 12
- 4.4 Results .......... 12
  - 4.4.1 Verifying the differences between both architectures .......... 12
  - 4.4.2 When the results differ .......... 13
  - 4.4.3 Characterization of the phenomenon .......... 14
  - 4.4.4 Effects of the intermediate rounding. .......... 16
- 4.5 A summary of the results .......... 17
- 4.6 Recommendations for scientific programming .......... 18

5 Floating point registers .......... 19
- 5.1 Introduction .......... 19
- 5.2 Using registers with extended exponent .......... 19
- 5.3 Advantages and limitations of the exponent's extended range .......... 20
- 5.4 Results .......... 20
  - 5.4.1 Verifying underflow/overflow conditions .......... 20
  - 5.4.2 Using floating point registers .......... 23
- 5.5 Analysis of the results .......... 24
- 5.6 Benefits of low level programming .......... 25
- 5.7 Recommendations for scientific programming .......... 25

6 Quadruple precision .......... 27
- 6.1 Introduction .......... 27
- 6.2 Selected problems .......... 27
  - 6.2.1 LU factorization .......... 28
  - 6.2.2 The Cholesky decomposition .......... 29
  - 6.2.3 Matrix inversion .......... 30
- 6.3 Results .......... 31
  - 6.3.1 CPU time .......... 31
  - 6.3.2 Precision .......... 39
- 6.4 Analysis of the results .......... 41
- 6.5 Recommendations for scientific programming .......... 42

7 Conclusions .......... 43

# 2 Introduction

The Itanium 64 bits architecture has been developed for providing high availability, scalability and performance in solving scientific and business problems.

It is based on the EPIC (Explicitly Parallel Instruction Computing) technology that assumes as a basic principle that either the programmer or the compiler should explicitly indicate the parallelism in a sequence of instructions without forcing the processor to reconstruct it from a sequence of operations. Such technology has the capability of executing several instructions simultaneously, providing both an efficient use of the memory and flexibility and scalability.

It also allows maximizing the benefits of the parallelism at instruction level in the source code, by using predication and speculation techniques. Moreover, this approach uses methods to reduce the negative effects in performance produced by a very limited number of registers, high latency time for memory access, costs of bifurcations and nonexistent or low degree of parallelism.

Another important aspect is its compatibility with the IA-32 architecture at hardware instruction level, hence allowing compatibility with legacy systems.

From a scientific viewpoint the Itanium floating point architecture has a wide range of data types, including those of high precision, multiply-add operations, division, square roots and a large number of floating point registers.

The floating-point data types may be internally mapped into an 82-bits format with a mantissa of 64 bits and an exponent of 17 bits. This allows to get a wider numerical range as compared with other processors; an important aspect for scientific computations.

The large number of floating point registers, some static and other in rotational state, allows to efficiently implement complex calculations.

Using the multiply-add operation, which combines two operations into a single one with only one meaningful rounding error, facilitates the implementation of new software based algorithms as division and square root.

In the following, we study how these characteristics can be used for solving scientific problems. We make comparisons with the IA-32 architecture and give recommendations for a better use of the Itanium 2 processor.

# 3 Compilers

The fact that the IA-64 architecture assigns the possible exploitation of parallelism to the compiler, frees hardware resources for implementing large caches and a high number of registers. Therefore, a direct relation exists between the speed and the quality of the generated code.

Due to this reason, the first part of this study was focused in the comparison of different compilers and the options they offer.

The tested compilers were those of GNU, the best well-known option for Linux platforms, the Intel compilers, and two other developed if open source projects for the IA-64 architecture. The last two are:

- Open64 Compiler Tools
- Open Research Compiler (ORC) for the Itanium Processor Family

All tests were carried out using:

- 2 Intel® Itanium® 2 processors
- Clock frequency 1.5GHz
- 6MB L3 cache
- 6.4GB/s system bus bandwidth
- 2GB DDR memory; 8.5GB/sec bandwidth
- Operating system: Red Hat Linux 7.2

## 3.1 Tests

As expected, the first step was to found out the best compiling options. We used FreeBench, a free benchmark for all UNIX like systems as well for windows systems, composed by four programs for testing performance with integers and three programs for floating point calculations. They are briefly described in the following:

Programs for integers:

1. *Analyzer* (Language C): This program is a tool for analyzing memory access traces for data dependences.
   *Stress points*: This program is for integers only but it is mainly limited by memory system performance. The memory accesses are very scattered, and while out-of-order processors can hide some of the memory latency, they cannot hide all of it. Fast caches larger than 512kB seem to help to some extent but not entirely. The key to good performance is a well-balanced machine with good memory bandwidth, low memory latency and good latency hiding capabilities. Compilers able to insert prefetches should be able to hide some of the memory latency, and thus improve performance quite a bit.

2. *FourInARow* (Language: C): This is a program that plays a game of "four in a row" against itself. It uses a min-max method with alpha-beta pruning as a search algorithm.
   *Stress points*: This program uses only integer arithmetic, but is not limited by the memory system. Some arithmetic is done using 64 bit integers; so 64 bit machines should fare well. The memory footprint is small and the execution time is spent in small recursive loops. Most of the execution fits in on-chip caches. Wide super scalar processors with OoO capabilities should see good performance in this program. Some compilers have difficulties generating good code for this program. Thus it is also a good compiler test.

3. *Mason* (Language C): This program solves a puzzle.
   *Stress points*: This program uses only integer arithmetic, and has a very small data set. This is a pure clock frequency and ILP limited program. It seems to stress compilers too, as some compilers generate bad code and many compiler optimizations make the program slower.

4. *pCompress2* (Language C): A file compressor using a three-stage approach. Burrows Wheeler blocksorting, run length encoding and Arithmetic coding.
   *Stress points*: This program is quite memory intensive. The whole indata file is processed at once and is repeatedly scanned for compressible data. The program makes heavy use of the C library functions qsort, memcmp and memcpy. If your C library is slow, this program will also be slow.

Programs for floating point calculations:

5. *PiFFT* (Language C): This program uses a huge FFT to calculate many decimal places of PI (3.1415...). The reference dataset calculates 4 million decimal points.
   *Stress points*: Floating-point hungry and intensive memory use are the characteristics of this program.

6. *DistRay* (Language C): This is a small ray tracer using random ray distribution to achieve anti-aliasing and soft shadows. It features variable recursion depth for reflections.
   *Stress points*: This is a floating-point program, with small memory footprint. Most of the execution is spent in recursive loops, with many floating-point multiplications and additions. A good amount of ILP should be found, stressing the FPU resources heavily.

7. *Neural* (Language: C): This is a neural network doing character recognition. It tries to find a way of successfully storing (and thus being able to identify) a number of characters written out in ASCII graphics.
   *Stress points*: This program has quite a large dataset and is somewhat memory intensive. Fast memory is key to good performance.

The efficiency of the generated code is in direct correspondence of the compilation flags. The execution times vary dramatically according to the optimization options, as can be seen

in the following section. Therefore, in order to determine the compiling options, we developed a number of scripts that run during several hours testing combinations and registering the corresponding CPU times.

From the information gathered in those runs, we were able to choose the best options as the first step.

The second step was to compare the best options with each compiler, arriving at a conclusion in regard to the one to be used in our study.

## 3.2 Results

Figures 1 and 2 show the results obtained with the simplest options offered by each compiler. The y-axis takes as a measure the number of times the value exceeds the one obtained with a Sun Ultra 10 workstation.

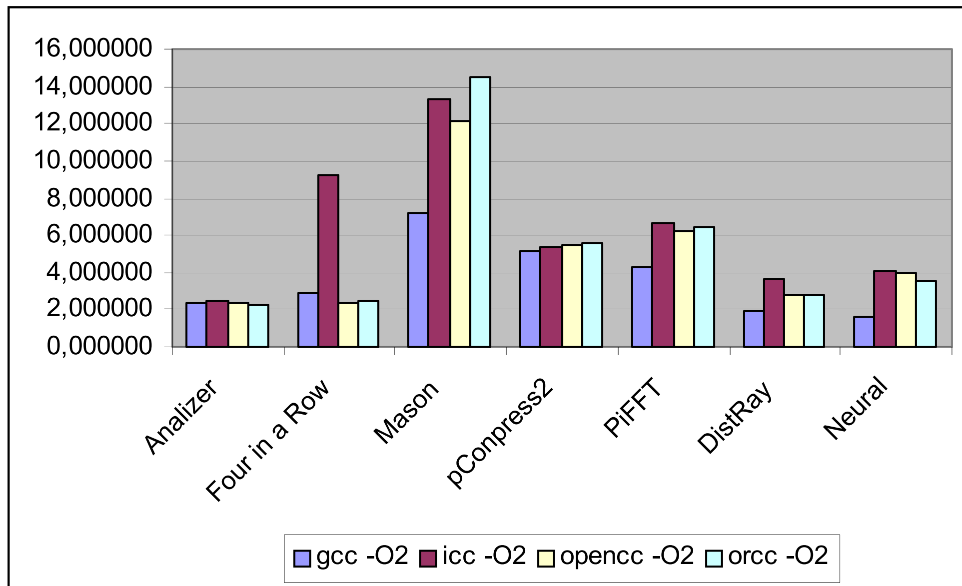



Figure 1

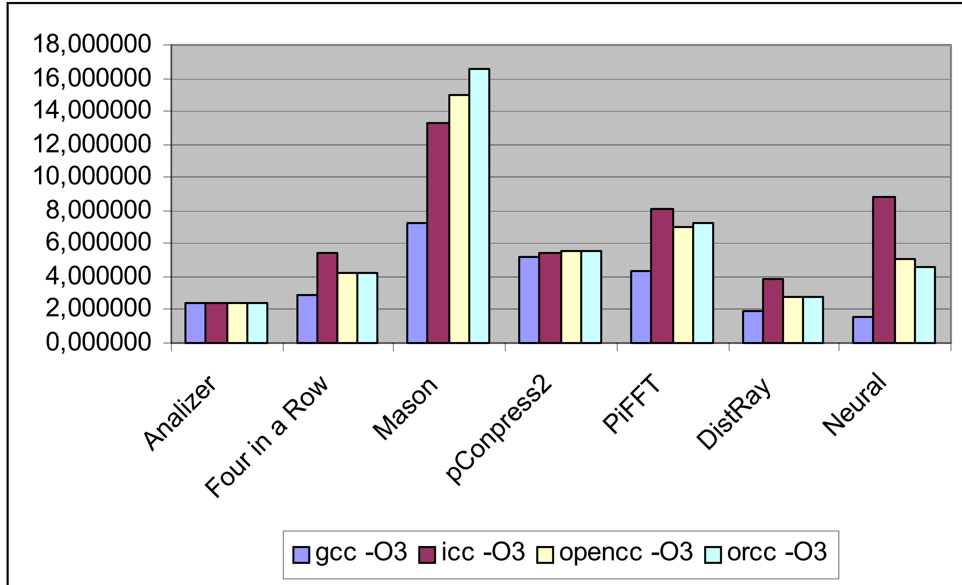



Figure 2

Figures 3 and 4 show the results obtained with the best compiling options for each compiler. It is important to point out the Intel compiler offers a much better documentation than all other, a fact that very likely influenced positively on the results obtained with that compiler.

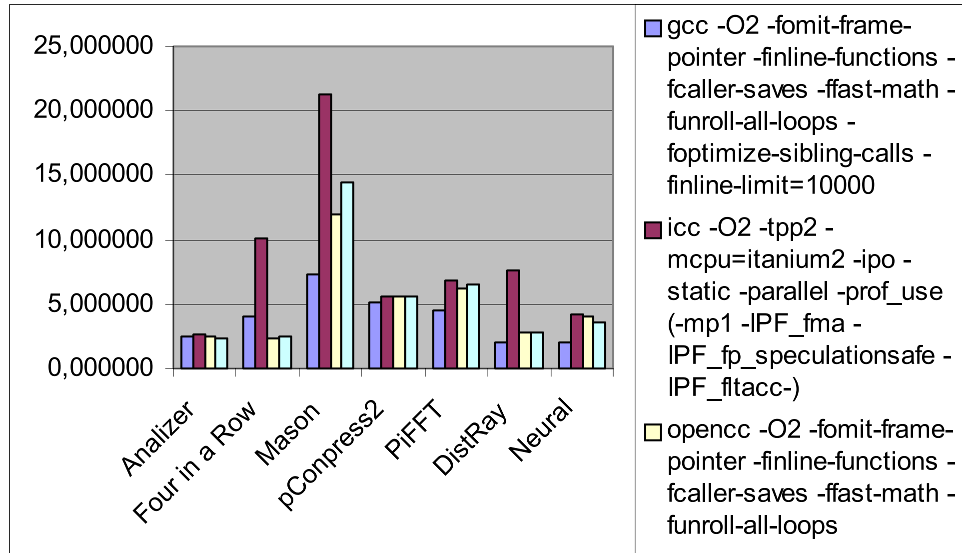



Figure 3

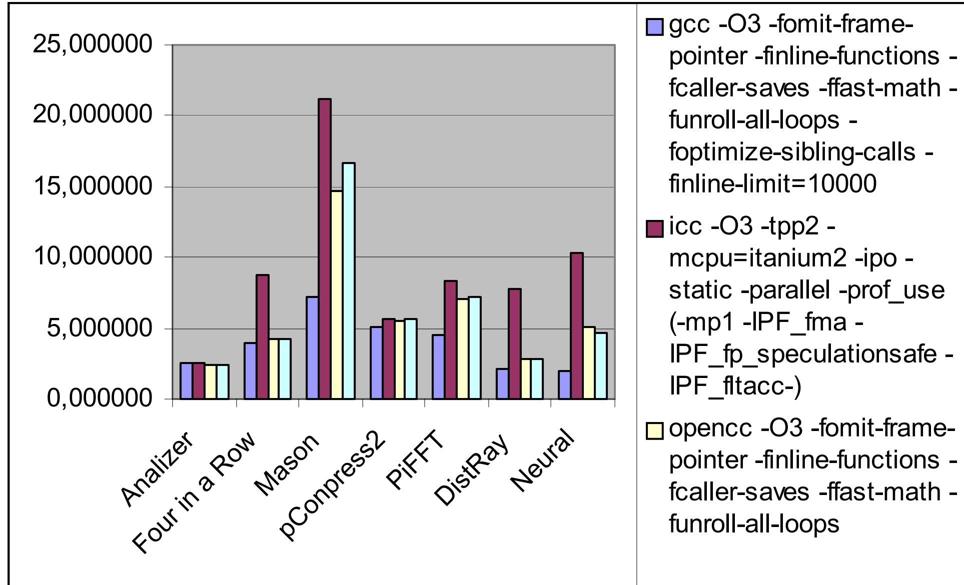



Figure 4

## 3.3 Analysis of the results

In most cases the compiler that offered the best performance results was the Intel's "icc". Moreover, when using "Profile Guided Optimization" the obtained CPU times were by far much better than those given by other compilers.

Due to those results we decided to use the Intel compilers for our study, with the following compiling options for both, programs computing with integers or floating point numbers:

| | **Compiler** | **Optimization options** |
|---|---|---|
| **Programs computing with integers** | Intel | -O2 -tpp2 -mcpu=itanium2 -ipo -static -parallel -prof_use |
| **Programs computing with floating point numbers** | Intel | -O3 -tpp2 -mcpu=itanium2 -ipo -static -parallel -prof_use -mp1 -IPF_fma -IPF_fp_speculationsafe -IPF_fltacc- |

# 4 Multiply-add fused operation

## 4.1 Introduction

In most scientific computing problems the programmer has to implement operations involving combinations of additions and multiplications. This is particularly true when solving computational linear algebra problems, in iterative algorithms, etc.

The computation of a product followed by an addition of floating point numbers when performed in a classical architecture by means of individual operations, involves two round-off errors: the error from the multiplication and the final error of the operation.

In the IA-64 architecture a fused multiply-add operation exists called FMA for floating point operands. That operation computes the product of the first two operands with high precision, followed by the addition of the third operand. The advantages are that it's faster than performing the same calculations separately and a unique significant round-off error appear.

In this section we are going to analyze the mentioned benefits in problems involving this sort of operation by comparing different architectures.

## 4.2 The FMA operation

The fused multiply-add operation computes in a single step:

$$f1 = f3 * f4 + f2$$

where $f1, f2, f3$ y $f4$ are floating point registers of the IA-64 architecture. The machine code instruction for this operation is:

*fma.pc.sf     f1 = f3, f4, f2*

where *pc* and *sf* are indicators of the rounding and precision controls respectively.

The above operation first executes the calculation without losing precision (that is, internally representing the result with the necessary number of bits). The product of two floating-point numbers can be exactly represented by using a number of bits at least equal to the double of those in the operands' mantissas. For addition, it is enough to have an extra bit.

In the IA-64 architecture, addition and multiplication are pseudo-operations, implemented by the FMA instruction and the use of the special registers *F0* and *F1*, whose values are initially set to zero and one respectively. Thus, the instructions:

*fadd.pc.sf    f1 = f3, f2*
*fmpy.pc.sf    f1 = f3, f4*

are pseudo-operations expanded respectively to:

*fma.pc.sf     f1 = f3, F1, f2*
*fma.pc.sf     f1 = f3, f4, F0*

There is also a variant of that fused operation that implements multiplication and subtraction in a similar way, called FMS. Analogously, the pseudo-operation FSUB computes the subtraction of two floating-point operands expanding itself into the FMS instruction, and using the *F1* register, which is always equal to one.

## 4.3 Benefits and uses of the fused operation

As mentioned before, the main advantages of using the fused operation are improved performance and precision. In the context of the present report, we will be focused on the benefits arising from the use of higher precision in the calculations.

Calculations of the form $A * X + B$ appear frequently in scientific applications. Usually the final result is obtained after a long sequence of cycles or iterations where each one involves at least an expression of this sort. Therefore, the final result depends on how precise those intermediate expressions are computed.

As mentioned before, the IA-64 architecture computes $A * X + B$ by means of a single machine instruction. Since each floating-point instruction involves a rounding to the nearest machine number, IA-32 computers produce a rounding error after computing $A * X$, and another one for the final evaluation of $A * X + B$. We will see later the clear superiority of the IA-64 architecture in regard to the IA-32 one.

Another benefit of the fused operation appears when the product $A * X$ has a similar magnitude of $B$ but with different signs because in such a case the subtraction of those values leads to a loss of significant digits.

In this work we will concentrate our attention in the additional error introduced by the intermediate round off. We will mainly interested in those cases where the evaluation of $A * X + B$ using the FMA instruction differs from the one obtained by separate multiply-add instructions. For those cases we have that:

$$Round(Round(A*X) + B) \neq Round(A*X + B)$$

## 4.4 Results

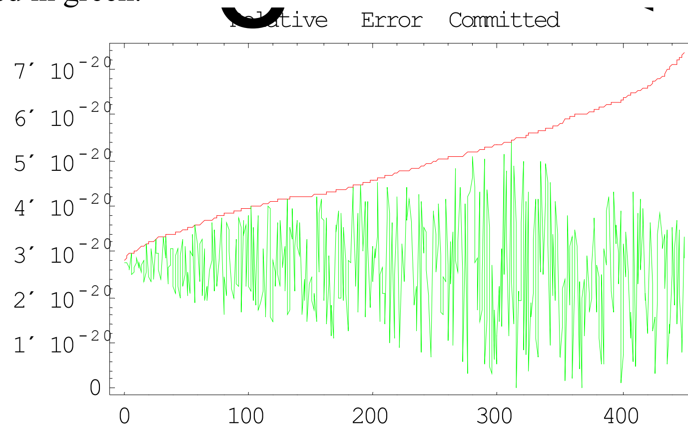
In this section we present several experiences aiming at determining the advantages of using the FAM fused operation. The comparisons are made using compiled code that uses FMA against code forced to perform the operations separately.

### 4.4.1 Verifying the differences between both architectures

As a first experience a C program was written that generates triplets of long double floating point numbers (double-extended IEEE-754) in normalized form, with each number following a uniform distribution. Considering that each triplet consists of the numbers *A, X* and *B*, the expression $A * X + B$ is computed using the above mentioned alternatives. Both

results are compared, and when a difference is observed all involved numbers are stored. Initially, 500 registers of this kind were generated.

In order to have a preliminary analysis of the results obtained with that program, for each register the "true" value of $A * X + B$ and the corresponding relative error were calculated with very high precision using special libraries. The results were sorted according to the resulting error when using the IA-32 mode. In the following figure the relative error using both alternatives is shown for 500 cases where the evaluations differ. The relative error with the IA-32 mode appears in red, while the errors for arising from the use of the FMA instruction are depicted in green.




As can be seen in the graph, in the cases where both modes differ the relative error produced by the FMA instruction is always less than the one given by the IA-32 architecture.

In the following, we will see that such a difference is due to the additional round-off error introduced by the calculation of $A*X$, inherent to the use of separate instructions. Such an error does not occur if the FMA instruction is used.

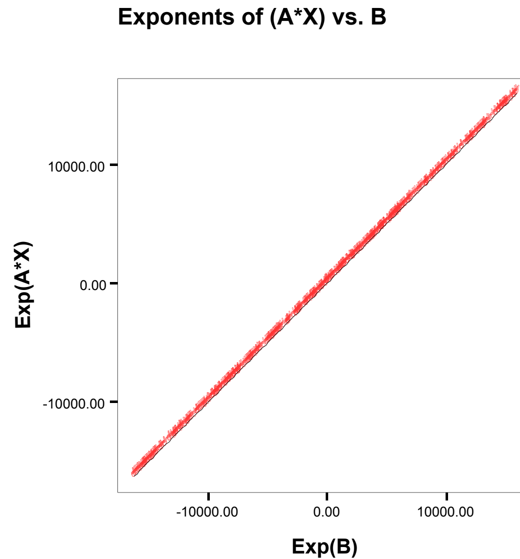
For the generated triplets of floating point numbers, 1 out of 25000 contains values such that both modes differ in the calculation of $A * X + B$. Therefore, both approaches lead to different results with a relative frequency of 0.0004.

The rounding strategy used in this experience is the one giving the nearest machine number (default option in both architectures). However, the sort of rounding used does not affect the conclusions. A similar consideration is valid for the floating-point data type used since the results are valid not only for double-extended numbers, but also for any other floating-point format.

### 4.4.2 When the results differ

An analysis of the data generated by the previous experiment shows that the additional error produced by the intermediate round-off appears when the exponent of $A*X$ has a

similar magnitude than the exponent of *B*. Extracting the exponents of *A*X* and *B*, for the registers previously obtained, a very significant linear correlation is obtained.




The data analysis also shows that the obtained line has a "width" approximately equal to the length of the mantissa measured in bits. This cannot be seen directly in the graph because of the used scale. This phenomenon occurs because it is necessary that certain bits in the mantissas of *A*X* and *B* coincide in their positions for getting that the round –off error introduced in the computation of *A*X* affects the result of the operation $A * X + B$.

Therefore, when both ways of computing $A * X + B$ lead to different results, the exponents of *A*X* and *B* should have the same order, except by a constant whose value is at most the length in bits of the corresponding mantissas.

In the next section we will analyze this relation in more detail.

### 4.4.3 Characterization of the phenomenon

Let δ be the absolute round-off error introduced in the calculation of *Round(A*X)*. When both modes differ we have that

$$Round(Round(A*X)+B) \neq Round(A*X+B)$$

Those differences should necessarily have the same order of δ, the absolute error introduced by the intermediate round off in the computation of *A*X*. Since this error is in the order of the last significant bit of the mantissa of *Round(A*X)*, the same happens with the magnitude of the difference between both evaluations.

More precisely:

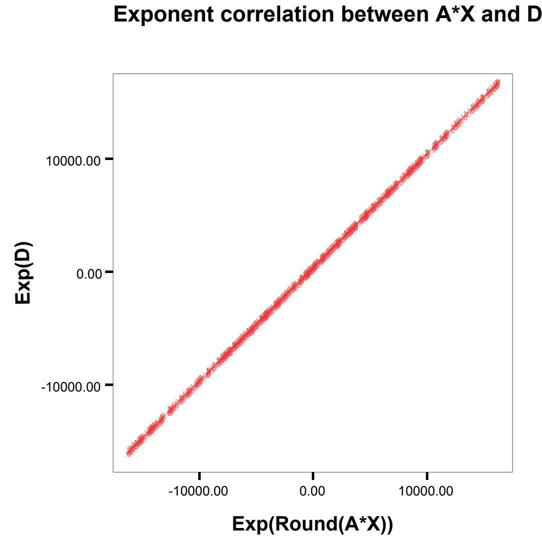
$$Exp(Round(A*X)) - Exp(D) \approx N$$

where *N* is the length in bits of the mantissa used in the floating point representation and

$$D = Round(Round(A*X)+B) - Round(A*X+B)$$

is the difference between both models.

The following is the graph of the correlation between the exponents of *Round(A*X)* and *D*, calculated from the data gathered in the experiment previously described. The linear relation between the exponents is clear:




It is necessary to point out that in the relation $Exp(Round(A*X)) – Exp(D) \approx N$ the strict equality does not hold due to the carrying on effect produced by the rounding operation.

The analysis of the data collected by the previous experiment shows those cases rarely occur and that they are not relevant for the problem we are interested in.

The same analysis shows that in those infrequent cases $Exp(B) > Exp(A*X)$ may occur. When that relation holds, the rounding error $\delta$ produced in the operation *Round(A*X)* may cause the differences in the evaluation of $A*X+B$ if it affects the mantissa of $B$. This can happen as a carrying effect in at least $|Exp(A*X) - Exp(B)|$ bits of the mantissa of *Round(A*X)*. Therefore, in most cases where rounding produces differences in the evaluation of $A*X+B$, the following inequality holds

$$Exp(A*X) \geq Exp(B)$$

The cases in which that relation does not hold are exceptional because of carrying effects or by the above mentioned phenomenon of precision loss in the subtraction of $A*X$ and $B$ with comparable magnitudes (see ***Benefits and uses of the fused operation***).

Using that inequality, we can characterize more precisely the result obtained in the previous section. When differences occur in the evaluation of $A*X+B$ due to the additional rounding

introduced by the second operation, the exponents of the floating point numbers *Round(A*X)* and *B* are of the same order except by a constant. That relation is given by

$$Exp(B) \leq Exp(A*X) \leq Exp(B) + N + 1$$

or

$$0 \leq Exp(A*X) - Exp(B) \leq N + 1$$

As an example, the graph of the linear regression presented before and the last inequalities not only show that the “width” is of the order of the mantissa *N* measured in bits, but also that the intercept of the regression line has order *N/2*. In the experiment that empirically exhibits this result the long double C data type (double-extended) was used with 64 bits mantissas.

Hence, it was to be expected that the intercept value should be approximately 32. The regression line computed for 500 observations was

$$Exp(A*X) = Exp(B) + 31.39$$

Finally, it is obvious that if an accurate linear relation exists of the form

$$Exp(Round(A*X)) - Exp(D) \approx N,$$

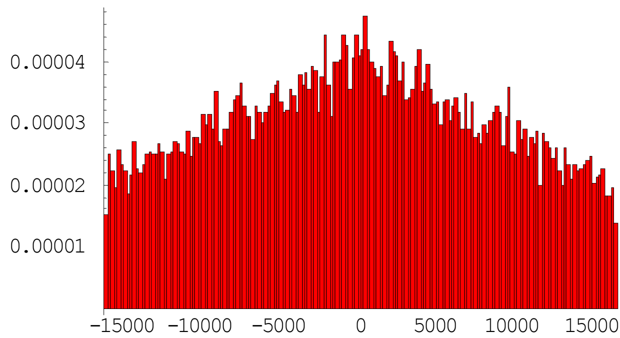
and that if the differences in the evaluation occur only when *Exp(A*X) and Exp(B)* differ by a constant bounded by the mantissas length, another linear relation between *Exp(B)* and *Exp(D)* also exists.

### 4.4.4 Effects of the intermediate rounding.

Up to now we saw that when a difference D occurs between the evaluation of $A*X+B$ by the different ways due to the intermediate rounding of the product $A*X$, it is given by the order of the least significant binary digit of the floating point number *Round(A*X)*.

A direct consequence is that the difference $D$ can be made arbitrarily large when the magnitude of $A*B$ is also large. Since that magnitude is of the order of $B$ except by a constant bounded by the mantissa’s length N, an analogous relation holds between $B$ and $D$. On the contrary, the relative error introduced by $D$ is constant, independently of the values of $A*X$ and $B$. This is so because the difference $D$ is of the order of the mantissa’s least significant bit corresponding to the floating point number representing $A*X$, and the mantissa’s lengths are constant.

It remains to be determined if the magnitudes of $A*X$ and $B$ for which significant differences appear are unusual or if such a phenomenon can occur with any possible machine number. The following histogram shows the relative frequencies of the exponents of $A*X$ for a number of cases in which differences in the evaluation of $A*X+B$ appeared:




As it can be seen, differences appear for any possible exponent of *A*X* (considering this of the double-extended type). We can consider that the distribution of those exponents follows a uniform distribution; and therefore the probabilities of occurrence are practically equal. A similar analysis holds for the distribution of the exponents of *B*, being the histogram almost identical.

## 4.5 A summary of the results

The evaluation of the expression *A*X+B* can be performed more precisely in the IA-64 architecture than in the IA-32 one, because the former has the machine instruction FMA.

That instruction does not introduce the additional round-off error of *A*X*, that occurs in the IA-32 architecture because of the separate use of the multiplication and addition instructions.

The differences in the evaluation are of the order of the mantissa's last significant bit of the floating point number *A*X*. Thus, in the IA-32 architecture the absolute value can be made arbitrarily large in calculations involving *A*X+B*, according to the value of *A*X*. Since this last value can take arbitrarily large values with equal probabilities, the corresponding absolute errors will also be large. This serious shortcoming does not appear in the IA-64 processors.

This sort of error introduces a constant relative error inversely dependent of the mantissas' length. Therefore, this typical relative error of the IA-32 architecture will be greater for the single data type than for the double one. Of course it will be even less for the double-extended precision. Well on the contrary, in the IA-64 architecture this sort of error ***does not appear*** when using the *FMA* instruction, independently of the employed data types.

Another aspect to be taken into account is that even when the relative errors are constant and relatively small for computations performed with high precision, the nested evaluation of the above mentioned expressions would produce a geometric magnification of them. Therefore, the final result may be dramatically affected when solving linear algebra problems or when using certain iterative methods where expressions of the form *A*X+B* appear frequently.

## 4.6 Recommendations for scientific programming

The obvious advice when using the IA-64 architecture is to employ the compiler's option for allowing the use of the *FMA* instruction whenever possible.

When programming in assembly language, this instruction can be used in several algorithms to improve efficiency and precision. Examples of its use appear in the computation of the division and square roots of floating point numbers. We recommend taking this instruction into account when designing or implementing algorithms and looking forward for opportunities to make the best use of it.

# 5 Floating point registers

## 5.1 Introduction

When it is necessary to use the standard IEEE-754 floating-point data types, the range and precision of all variables defined in a program are completely defined. On the other hand, when the precision combined with good performance is extremely important, the floating-point unit of the IA-64 architecture has some features that allow being more flexible in regard to the limitations of the standard, mainly in the range of the different floating-point data types.

In this section we will study how you can get benefits by using the 17 bits exponents of the floating-point registers, even in programs that only use the standard IEEE-754 data types. We will see how and under what circumstances the 82 bits floating point registers, in addition to other architecture advantages allow getting more precise results than those obtained with the IA-32 architecture for a given program.

## 5.2 Using registers with extended exponent

The floating point registers of the IA-64 architecture offer two additional bits whose purpose is:

- To allow explicit signaling of particular cases. There are three different situations that occur in the IA-64 architecture that do not exist in the IA-32 one: the use of Parallel FP (two single IEEE-754 in parallel), the use of integers explicitly represented in the floating point registers, and the occurrence of NaTVal that indicates a failure in the speculative operational mode.
- To add the two bits to the exponent. Besides the standard and extended IEEE-754 data types of the IA-32 (stack single and double, with an exponent of 15 bits), the IA-64 architecture support three new extended data types: register single, register double and register (having the same precision than the single, double and double-extended of the IEEE-754 standard, but with an exponent of 17 bits). The special situations like NaNs, non-normalized floating point numbers and all their variations are signalized in the usual way of the IEEE-754 standards, using the last value of the exponent's range (that is, the value 0x1FFFF, when the 17 bits are equal to one).

We shall concentrate in this Section on how to exploit the two additional bits of the FP registers as an extension of the exponent, the way of accessing the 17 bits of the exponent for the three types of the register format by means of the WRE (Widest Range Exponent) bit of the floating point unit's control register (FPSR, Floating-Point Status Register).

In our experiments all programs had been written in C. When using the Intel's compiler the WRE bit of the FPSR register can be accessed by means of the _mm_setfpsr and _mm_getfpsr macros. The following C instruction activates that bit:

```
_mm_setfpsr(_mm_getfpsr() | 896);
```

## 5.3 Advantages and limitations of the exponent's extended range

The benefits of extending the exponent's range seem to be obvious: for instance, the likelihood of facing underflow/overflow conditions diminishes when compared to the IA-32 architecture. This is of course beneficial for the precision with which a given computation is carried out.

Unfortunately, to extend the exponent's range has limitations and therefore the former assertion is partially true only under certain circumstances. In the following, we analyze when this feature can be efficiently used and under what conditions.

The IA-64 architecture has a floating-point unit with important advantages as for instance the large number of registers and the possibility of rotating them in a window. If in addition to that we consider the FP registers with 17 bits exponents, a compiler able to exploit these features could generate a much better object code for scientific calculations from the precision viewpoint.

## 5.4 Results

In this section we present a number of experiments whose aim is to verify two characteristics of the IA-64 architecture. The first group of experiments tries to determine the impact of using 17 bits exponents in the different floating-point data type. With that purpose we compare the two architectures by checking when underflow/overflow conditions occur in certain predefined problems. The second group's goal is determine the conditions under which it is possible to get benefits from the use of extended exponents. The idea is to exhibit the limitations and under what conditions the new feature can be used.

### 5.4.1 Verifying underflow/overflow conditions

The first experiment consisted in comparing the occurrence of underflow in both architectures. For that purpose we measure the number of iterations needed for reaching the underflow condition in a program that starting from 1, divides by 2 using long double variables in C. The used computers were an Itanium 2 of 1.5 Ghz and a Pentium 4 of 2.8 Ghz.

The main idea is that at each iteration, divisions by 2 decrement the exponent by 1 without modifying the mantissa until reaching the underflow condition. After that, further divisions by 2 lead to non-normalized numbers, and therefore the exponent remains at its least possible value and the only possible changes affect the mantissa.

When using long double variables in the IA-32 architecture, 16383 iterations are required plus 63 iterations with non-normalized numbers to reach the floating-point number zero (63

bits mantissa). Those results correspond to a 15 bits exponent from 0 to –16382 and to a 64 bits mantissa including integer part bit, which coincides with a double-extended data type of IEEE-754 (equivalent to the long double type in C).

In the IA-64 architecture when the flag WRE is not activated the results are of course identical. However, when the flag is activated 65535 iterations are required for reaching underflow plus 63 iterations with non-normalized numbers to reach the floating-point number zero. Those values correspond to a 17 bits exponent and a 64 bits mantissa when considering the integer part bit.

A similar experience was made for studying the appearance of the overflow condition using multiplications by 2 instead of divisions. The results were as expected.

The next experience was focused on analyzing the impact of the extended range for a simple computation consistent in dividing the product of the first N natural numbers by itself. The used strategy was to perform N successive divisions of the first N natural numbers, starting with 1, and then to multiply the result sequentially by the same N natural numbers. The following are the results for different values of N when using the long double data type:

| *N* | *IA-32* | *IA-64 (WRE=0)* | *IA-64 (WRE=1)* |
|---|---|---|---|
| 1757 | 1 | 1 | 1 |
| 1758 | 1 | 1 | 1 |
| 1759 | 1.00002 | 1.00002 | 1 |
| 1760 | 1.06308 | 1.06308 | 1 |
| 1761 | 0 | 0 | 1 |
| 1762 | 0 | 0 | 1 |
| ... | 0 | 0 | 1 |
| 5912 | 0 | 0 | 1 |
| 5913 | 0 | 0 | 1 |
| 5914 | 0 | 0 | 1 |
| 5915 | 0 | 0 | 1.00346 |
| 5916 | 0 | 0 | 0 |
| 5917 | 0 | 0 | 0 |

The underflow condition is evident from N=1759 in the IA-32 architecture and the results are completely wrong in the next two iterations. In the IA-64 architecture, when the WRE bit is not activated, the results are identical to those obtained with the IA-32 architecture. In both cases the exponent has 15 bits. On the contrary, when activating the WRE bit, the extended exponent delays the underflow condition until the iteration corresponding to N=5915.

Up to this moment, all experiments were limited to the use of the long double data type in C. The next couple of experiments show the use of 17 bits exponents also holds for the float and double data types in C. That is, the mantissas of the used data types have 24 bits and 53 bits respectively, but the exponent has 17 bits. In the following, we give the results for the same problem:

| *N* | *IA-32* | *IA-64 (WRE=0)* | *IA-64 (WRE=1)* |
| --- | --- | --- | --- |
| 33 | 1 | 1 | 1 |
| 34 | 1 | 1 | 1 |
| 35 | 1.00001 | 1.00001 | 1 |
| 36 | 0.999803 | 0.999803 | 1 |
| 37 | 1.00293 | 1.00293 | 1 |
| 38 | 0.732911 | 0.732911 | 1 |
| 39 | 0 | 0 | 1 |
| 40 | 0 | 0 | 1 |
| ... | 0 | 0 | 1 |
| 5906 | 0 | 0 | 1 |
| 5907 | 0 | 0 | 1 |
| 5908 | 0 | 0 | 0.999999 |
| 5909 | 0 | 0 | 0.999998 |
| 5910 | 0 | 0 | 0.999998 |
| 5911 | 0 | 0 | 1 |
| 5912 | 0 | 0 | 0.990032 |
| 5913 | 0 | 0 | 0 |
| 5914 | 0 | 0 | 0 |

Repeating the same experiment with double type variables:

| *N* | *IA-32* | *IA-64 (WRE=0)* | *IA-64 (WRE=1)* |
| --- | --- | --- | --- |
| 173 | 1 | 1 | 1 |
| 174 | 1 | 1 | 1 |
| 175 | 0.999999 | 0.999999 | 1 |
| 176 | 1.00026 | 1.00026 | 1 |
| 177 | 1.03839 | 1.03839 | 1 |
| 178 | 0 | 0 | 1 |
| 179 | 0 | 0 | 1 |
| ... | 0 | 0 | 1 |
| 5912 | 0 | 0 | 1 |
| 5913 | 0 | 0 | 1 |
| 5914 | 0 | 0 | 1.0002 |
| 5915 | 0 | 0 | 0 |
| 5916 | 0 | 0 | 0 |

Therefore, when the WRE bit is on, the underflow condition occurs for values of N of the same order for each one of the tested data types. This is so because the underflow condition depends only of the exponent, and is practically independent of the mantissas' lengths. The slight observed differences arise because the results are distorted when reaching non-normalized numbers until stopping in zero. In other words, different mantissas require a slightly different number of iterations.

### 5.4.2 Using floating point registers

When analyzing the previous experiments a logical question is how the float, double and long data types of the C language can use 17 bits exponents, when the IEEE-754 standard defines them with shorter exponents. The answer lies in the way the compiler generates code that uses the floating point registers, and that is the goal of the next experiments.

As a first step we analyze the assembler code generated for the first experiment above reported. The program essentially consists of a cycle in which each iteration performs successive divisions by 2, starting with a variable set to 1. That analysis shows that the variable that is successively divided by 2 is stored in a floating point register, and the cycle where the divisions are performed operates on that FP register. Once the cycle finish, the result is stored in a long double variable in memory to be shown by means of the standard output.

In this way, the intermediate calculations are completely performed in the floating-point registers, and the final result is stored in memory using a standard variable. This storage is done by means of the floating-point instruction STORE, which performs the conversion (cast) between the register (17 bits exponent) and the memory variable (long double, 15 bits exponent) data types. The same optimization strategy holds for the single, double and long double data types.

The remaining part of this Section studies under what circumstances the compiler can generate optimized code for using floating point registers in order to get a benefit from the use of extended exponents.

The first experiment of this series consists in rewriting the program of the successive divisions in such a way that the compiler cannot replace the variables by floating point registers.

This would allow us to show that in certain cases the compiler's optimization can be tricked in such a way that it cannot take advantage of the extended exponents.

First, two different versions of the program were written. One allocates variables in static arrays and the other in dynamic arrays using the malloc function.

For the version using long double variables allocated in static arrays the results are the same as those obtained before. That is, when the WRE flag is activated 65535 iterations are required for reaching underflow, plus 63 iterations with non-normalized numbers to reach the floating-point number zero. When checking the generated assembler code it can be seen that the optimization phase took place, and therefore it cannot be bypassed by allocating variables to static arrays. The code is practically identical to the original.

The situation changes completely when the arrays are dynamically created. With the WRE flag on underflow occurs after 16383 iterations and the execution stops. This not only shows that registers with 17 bits for the exponent are not being used, but also that all intermediate variables are stored in long double variables which do not support non-normalized values as the processor's register do.

This explains why for both architectures computations continue with non-normalized numbers after the appearance of the underflow condition because the compiler always seeks for the maximum use of registers and they do support those numbers

The last experiment consists of evaluating if the use of functions imposes constraints on the exponents' magnitudes. For that purpose a version of the above mentioned program was written where the divisions by 2 are computed by means of a function whose parameter is the value to be divided. The results are exactly those obtained before with the WRE flag activated. To pass the parameters to a function by value or by reference forces the variable to be allocated in memory and not in a FP register, losing in such a way the benefits of using 17 bits exponents.

Similar results were obtained with the other data types. The use of variables in dynamic arrays has the same limiting effect, independently of the selected data type.

The analysis of the assembler code generated by the compiler shows the transference of data between registers and memory is done by means of the LOAD/STORE instructions.

They perform a conversion (cast) between the register type data and the IEEE-754 standard, and are responsible for the loss of the two extra bits.

The explanation of why the compiler generates code with those instructions is simple: **the C language does not have data types corresponding to the registers of the IA-64 architecture.** Its data types are float, double and long double, equivalent to the single, double and double extended of the IEEE-754 standard, and therefore casting is a must.

## 5.5 Analysis of the results

The first group of experiments clearly shows the use of 17 bits exponents has a direct influence in the occurrence of underflow/overflow conditions. They also show that another important benefit is the way in which intermediate calculations are carried out. Both aspects are very important for scientific computing.

From the results of the second group of experiments is very clear that the role of the chosen compiler is essential for taking advantage of he IA-64 architecture. It is normal that a compiler in the optimization phase attempts to allocate as many variables as possible to floating point registers looking for an improved efficiency of the generated code. If the architecture offers a large number of FP registers, the compiler can generate a much more efficient code. This is a key factor for explaining the superiority of the IA-64 architecture over the IA-32 one.

Contrariwise, when the compiler is unable to allocate certain variables in registers, those advantages are at least partially lost. The last experiments show that when the variables can be dynamically allocated in memory, or when a variable is forced to be in memory for being passed as a parameter to a function, possibilities are lost. This last case is as usual as having the program invoking functions from a numerical library.

Therefore, the benefits of using registers with extended exponents are limited to variables in a program since the use of functions leads to face the above mentioned constraints. This

means that both the choice of the compiler and how the code is written are crucial elements for maximizing the advantages of the IA-64 architecture.

## 5.6 Benefits of low level programming

Taking into account the compiler generates code capable of exploiting the benefits given by extended exponent registers, the question arises if it is possible to explicitly write programs for numerical computing where the used data types have their characteristics. The answer is positive because in the IA-64 architecture the machine instructions SPILL and FILL allow to load and unload registers from and towards a memory portion of 128 bits. Therefore, it is theoretically possible to develop a program in assembler able to exploit the architecture's features without the limitations derived from the use of a high level language as C.

Such a program will use the data types corresponding to the FP register that do not exist in C. When the programmer is forced to store data in memory, s/he could use the SPILL/FILL instructions instead LOAD/STORE that perform conversions between data types.

Needless to say, such a program may have some shortcomings like not being able to be ported to computers with a different architecture.

There are some numerical libraries tuned for Itanium systems like http://www.hp.com/techservers/tools/libraries.html#hp-ux_ia-64, http://www.vni.com/company/press/itaniumLinuxRelease.html, http://www.intel.com/cd/ids/developer/asmo-na/eng/catalog/processor/itanium/19577.htm

**but they do not seem to exploit the use of extended exponents.**

We would also have the possibility of having a new data type in a high level language that would allow defining classes (e.g C++). That new data type could be able to exploit the benefits of the extended exponents.

## 5.7 Recommendations for scientific programming

This last Section has some general recommendations for writing numerical programs in the IA-64 architecture, mostly in regard to the use of extended exponents. These recommendations are independent of those contained in the Intel's manual and do not exclude them.

First, it is necessary to have a suitable compiler in order to be able to access the important characteristics of the *FP* unit. The Intel's C/C++ icc compiler proved to be able to generate code that highly uses the *FP* registers, a key factor for having the benefits of the extended exponents.

In regard to the programming aspects, the code should activate the WRE flag by means of the macros previously described.

It is important to design the code in a way that allows the compiler to allocate them to *FP* registers, minimizing synchronization with the main memory. Each such synchronization casts the *FP* register to a IEEE floating point number.

It is also important to avoid excessive atomization of the code in different functions because in such a case the benefits of the extended exponents vanish.

Contrarily to what is considered as a good methodological approach to programming, a source code with nested cycles and low abstraction level if functions can be better optimized for this architecture than a “more correct” program in the sense of being “stylish”. This can be overcome by using macros, inline functions, and C++ templates, in an intensive way.

The previous recommendations exist solely because the studied compilers lack the ability to handle the extended exponent registers properly. The best solution to this problem is to develop or extend a compiler that defines a new data type compatible with the Itanium registers. This way, programmers don’t need to be aware of the issues of processor register/memory synchronization.

# 6 Quadruple precision

## 6.1 Introduction

The quad precision data type is a floating-point data type of the IEEE style that uses 128 bits (1 for the sign, 15 for the exponent, and 112 for the mantissa), supported in the Itanium architecture by emulation.

In this Section we will present several benchmarks and will analyze the results from the point of view of the performance, both in CPU times and precision, comparing the architectures IA-32 and IA-64.

## 6.2 Selected problems

The test problems that have been chosen for the benchmarks come from the computational linear algebra field and use ill-conditioned matrices.

We recall that the condition number of a matrix is defined by:

$$k(A) = \| A \| \times \| A^{-1} \|$$

where $\|A\|$ stands for any compatible norm, but from hereafter we will use $\|A\|_2$ denoting it simply by $\|A\|$.

The singular values of a matrix are defined as the square roots of the eigenvalues of $A^t A$ and are ordered according to $\sigma_1 \geq ...\sigma_n \geq 0$. When using $\|A\|_2$, if the matrix is non-singular we have that

$$k(A) = \| A \| \times \| A^{-1} \| = \frac{\sigma_1}{\sigma_n}$$

More generally, and for eventually rectangular matrices, we define

$$k(A) = \| A \| \times \| A^{+} \| = \frac{\sigma_1}{\sigma_k},$$

where $A^{+}$ denotes the pseudoinverse of $A$ and $\sigma_k$ is the least positive singular value. It is also elementary to prove that $k(A) \geq 1$.

When solving linear systems $Ax = b$, perturbations of the entries of the matrix and the right-hand side vector lead to an error in the solution such that

$$\frac{\| \delta_x \|}{\| x \|} \leq k(A) \left[ \frac{\| \delta A \|}{\| A \|} + \frac{\| \delta_b \|}{\| b \|} \right]$$

Due to this reason if $k(A)$ is "small" we will say that the matrix is well-conditioned because it does not "amplify" the errors, and if it is "large" we will say that it is ill-conditioned.

The real effect depends upon the machine tolerance $u$ of the computer defined as the least positive number such that $1+u \neq 1$ in floating point.

The rank of the matrix is *k* if $\sigma_k$ is the least positive singular value. A wise approach for deciding the value of *k* is when $\sigma_{k+1}^2 + ... + \sigma_n^2 < u^2$.

Therefore, when the matrix "tend to have deficient rank", then $k(A) \to \infty$.

It is worthwhile to point out that the determinant of a matrix is not a suitable indicator of its numerical condition because if $Ax = b$, then for any scalar $\alpha$ we have that from $\alpha\ Ax = \alpha\ b$ we have that $det(\alpha A) = \alpha^n det(A)$. Then, the determinant can take any value and cannot be used for deciding if the system is ill-conditioned. On the contrary, $k(A)$ is invariant by transformations of this kind.

What is valid is that if the matrix is singular, then $det(A) = 0$, a fact will be later used.

### 6.2.1 LU factorization

An elementary result states that a matrix can be written as the product of a lower triangular matrix times an upper triangular one and that this decomposition is usually performed by gaussian elimination. Then, we have

$$A = LU$$

From this factorization, the linear system *Ax=b* can be easily solved using the fact that *LUx=b* and later solving two triangular systems:

$$Ly = b$$

$$Ux = y$$

For our tests, the vector b will be defined in such a way that the solution is the vector $x^t = (1,,,1)$. Thus,

$$A \times \begin{pmatrix} 1 \\ 1 \\ ... \\ 1 \end{pmatrix} = b$$

Finally, an attempt to improve the computed solution will be done using iterative refinement.

The first benchmark is based on the classical *nxn* Hilbert matrix defined as

$$H_{i,j} = \frac{1}{i+j-1}$$

It is very well-known that when n =10 the extreme eigenvalues are:

*Maximum eigenvalue: 1.7519196702*
*Minimum eigenvalue: 1.0931538193.10-3*

Since the matrix is symmetric, the condition number if the quotient of those eigenvalues. Hence, *k(H) = 1602.98* and more generally, its condition number grows exponentially when $n \to \infty$.

### 6.2.2 The Cholesky decomposition

The Cholesky factorization can be considered as a variation of the gaussian elimination for definite positive symmetric matrices. Its outstanding feature is that the Cholesky algorithm is numerically stable without applying pivoting.

The result has the form

$$A = LL^t$$

where $L$ is a lower triangular matrix.

For this test we will use the non-singular matrix defined by:

$$A_{i,j} \begin{cases} 1 & \text{if } j \geq i \\ c_j & \text{if } j < i \end{cases}$$

and the Cholesky factorization will be applied to the matrix:

$$M = A^t A$$

which is obviously positive definite since $A$ is non-singular. Finally a system is defined as before.

The determinant of this matrix is:

$$Det(A_n) = \prod_{i=1}^{n-1} (1 - c_i),$$

and therefore

$$Det(M) = Det(A^t \times A) = Det(A^t) \times Det(A) = Det^2(A)$$

This means that by modifying the constants $c_i$ in order to become closer to 1, we can control the value of the determinant.

### 6.2.3 Matrix inversion

Given a non-singular matrix A its inverse $A^{-1}$ is defined by:

$$AA^{-1} = I$$

A possible way of computing the inverse is to solve the *n* systems:

$$Ae_j = I \qquad para\ 1 \le j \le n,$$

where $e_j$ is the vector whose entries are all equal to 0 except the *j-th* that is equal to 1.

The benchmark consists in computing the inverse, comparing the results with the exact solution. Therefore we need matrices such that analytical expressions for the inverse are known.

The matrix with coefficients $c_i$ described for the previous benchmark fulfills this condition. As we saw before

$$Det(A_n) = \prod_{i=1}^{n-1}(1 - c_i)$$

This value is different from 0 when all coefficients are different from 1. The inverse is given by $A^{-1} = \left(b_{ij}\right)$ where

$$b_{ij} = \begin{cases} \dfrac{1}{(1-c_i)} & i = j,\ i \neq n \\[2ex] \dfrac{-1}{(1-c_i)} & i \neq n, j = i+1 \\[2ex] \dfrac{(c_{j-1} - c_j)}{\left[(1-c_j)(1-c_{j-1})\right]} & i = n, j \neq 1, n \\[2ex] \dfrac{-c_1}{(1-c_i)} & i = n, j = 1 \\[2ex] \dfrac{1}{(1-c_{n-1i})} & i = j = n \\[2ex] 0 & \text{otherwise} \end{cases}$$

## 6.3 Results

The tests were implemented using the quadruple precision data type by means of _Quad available in the Intel's C compiler. The computers used for the tests were an Itanium 2 of 1.5 Ghz and a Pentium 4 of 2.8 Ghz

### 6.3.1 CPU time

The CPU times required by the different benchmarks correspond to problems where the matrices' dimensions were successively incremented.

It is necessary to point out the following details regarding the measurements:

- During execution the processors were dedicated exclusively to the tests.
- Whenever possible all benchmarks were compiled and measured identically.
- For low dimension matrices the CPU times were almost nil and therefore in order to achieve more accuracy, the test was performed *m* times within cycles, reporting the total time by *m*.
- The code section timed produces no output to console or files to avoid delays.

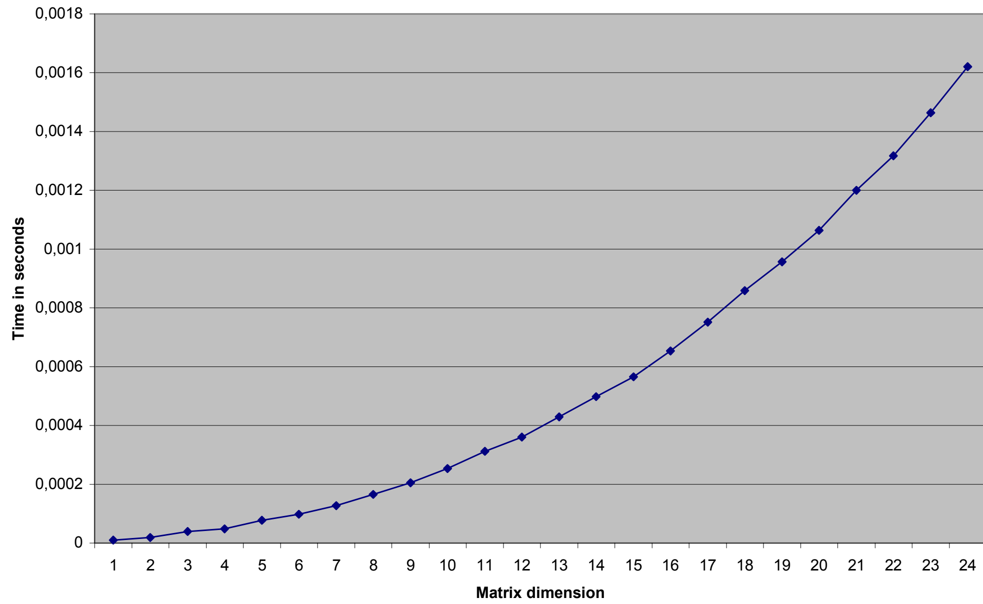
The following pseudo-code shows the model adopted for these measurements:

```
#include <stdio.h>
#include <time.h>

const int REPEAT = 100;

int main(int argc, char *argv[]) {
	clock_t time0, time1;
	double duration;

	time0 = clock();
	for (iter=0; iter < REPEAT; iter++)
		myFunction();
	time1 = clock();

	duration=(double)(time1-time0);
	duration = duration/(CLOCKS_PER_SEC * REPEAT);

	printf("%30.25f\n", duration);
}
```

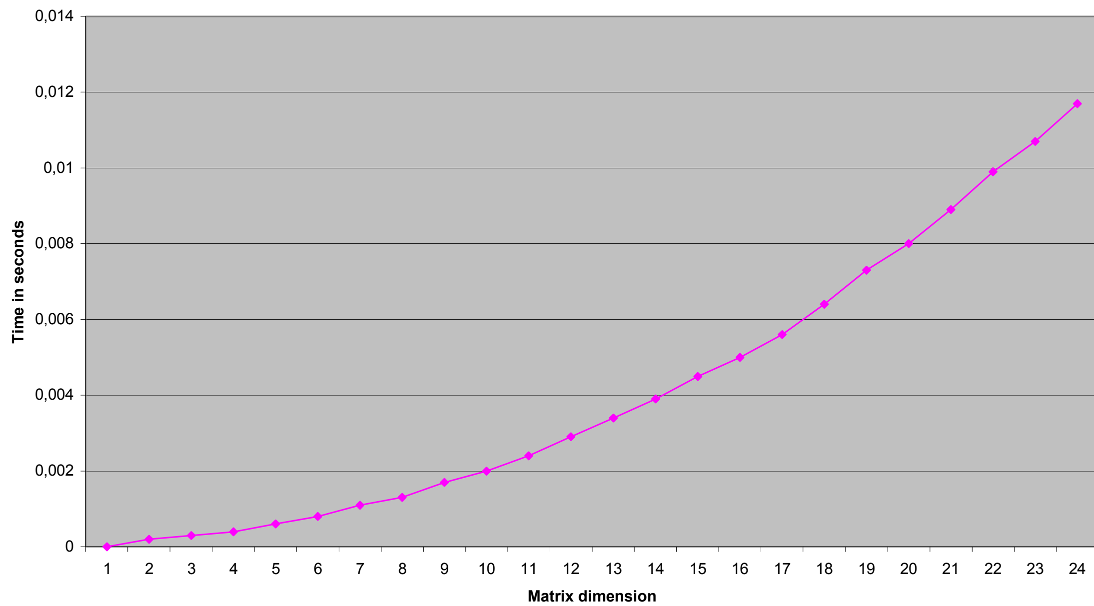
#### 6.3.1.1 LU factorization:

CPU times in seconds required for solving linear systems by LU factorization with the Hilbert matrix of different dimension in both processors:

| Matrix dimension | Time (Itanium 2) | Time (Pentium 4) |
|---|---|---|
| 2 | 0,00000976 | 0 |
| 3 | 0,00001952 | 0,0002 |
| 4 | 0,00003904 | 0,0003 |
| 5 | 0,0000488 | 0,0004 |
| 6 | 0,00007808 | 0,0006 |
| 7 | 0,0000976 | 0,0008 |
| 8 | 0,00012688 | 0,0011 |
| 9 | 0,00016592 | 0,0013 |
| 10 | 0,00020496 | 0,0017 |
| 11 | 0,00025376 | 0,002 |
| 12 | 0,00031232 | 0,0024 |
| 13 | 0,00036112 | 0,0029 |
| 14 | 0,00042944 | 0,0034 |
| 15 | 0,00049776 | 0,0039 |
| 16 | 0,00056608 | 0,0045 |
| 17 | 0,00065392 | 0,005 |
| 18 | 0,00075152 | 0,0056 |
| 19 | 0,00085888 | 0,0064 |
| 20 | 0,00095648 | 0,0073 |
| 21 | 0,00106384 | 0,008 |
| 22 | 0,00120048 | 0,0089 |
| 23 | 0,0013176 | 0,0099 |
| 24 | 0,001464 | 0,0107 |
| 25 | 0,00162016 | 0,0117 |

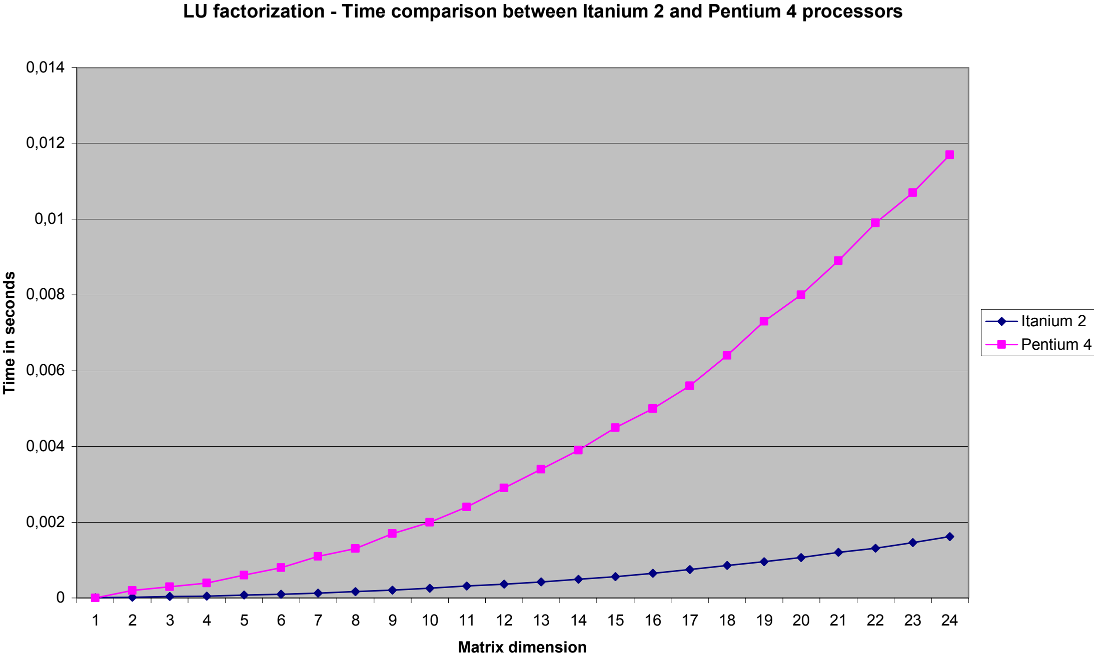


LU factorization - Itanium 2
Time in seconds
0,0018
0,0016
0,0014
0,0012
0,001
0,0008
0,0006
0,0004
0,0002
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Matrix dimension

LU factorization - Pentium 4
Time in seconds
0,014
0,012
0,01
0,008
0,006
0,004
0,002
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Matrix dimension

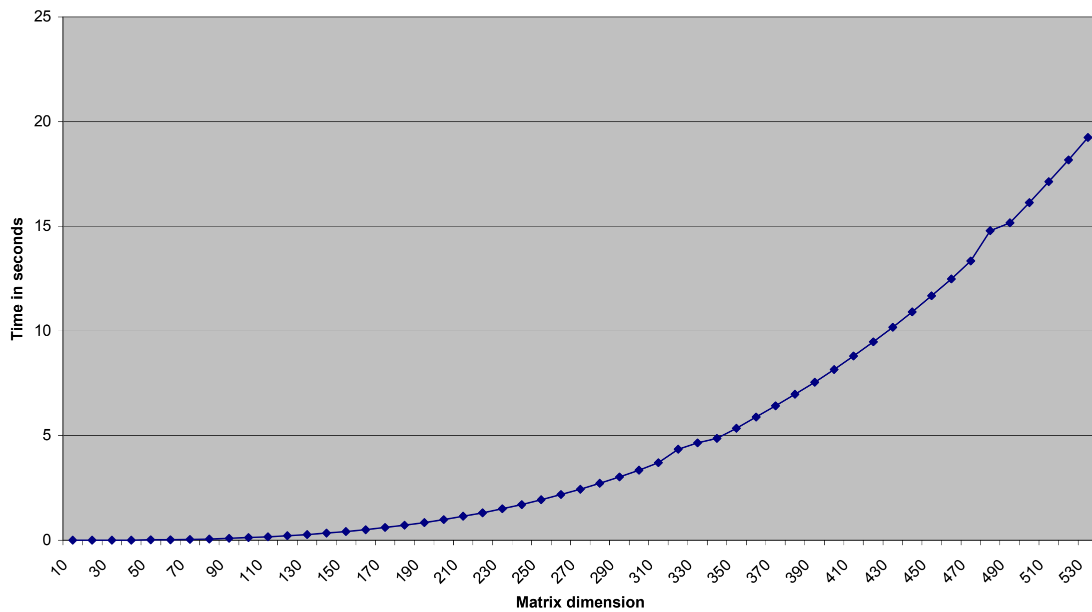
Comparison between both processors:

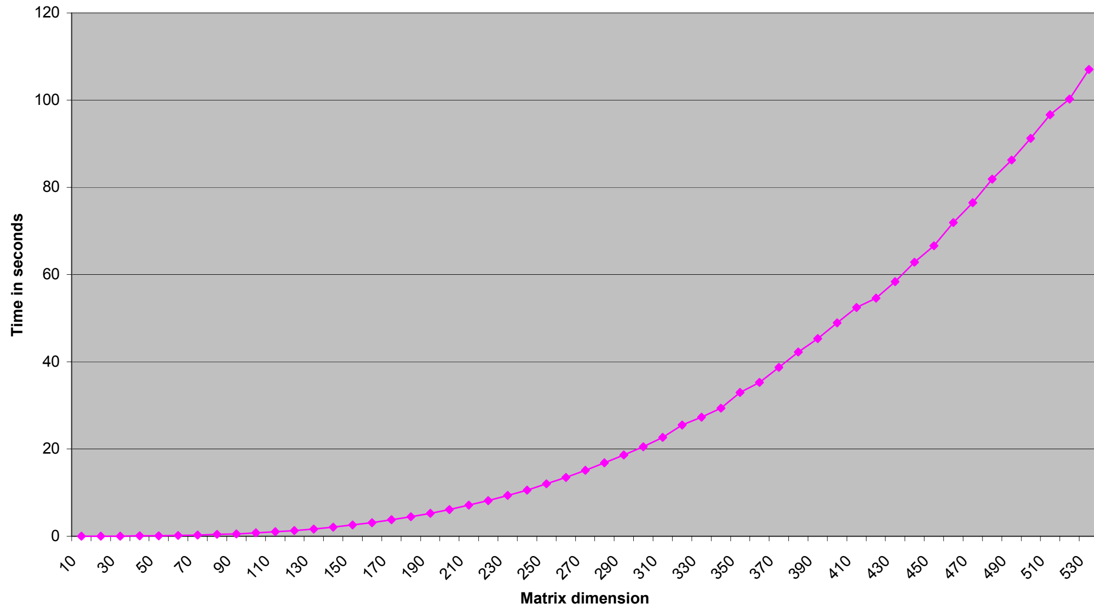


### 6.3.1.2 Cholesky decomposition

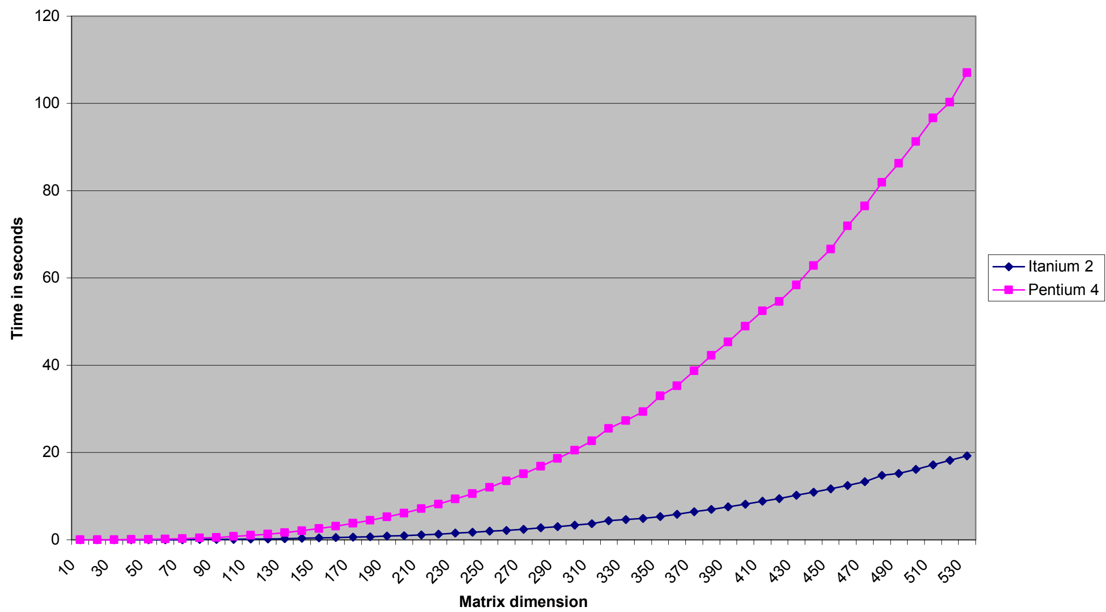
CPU times in seconds required by both processors for solving the linear system using the matrix with the $c_i$ constants, the Cholesky decomposition and different dimensions:

| Dimension | Time (Itanium 2) | Time (Pentium 4) |
|---|---|---|
| 50 | 0,015128 | 0,099 |
| 100 | 0,1227808 | 0,756 |
| 150 | 0,414312 | 2,576 |
| 200 | 0,9860528 | 6,095 |
| 250 | 1,9320896 | 12,047 |
| 300 | 3,3446544 | 20,555 |
| 350 | 5,3527744 | 33 |
| 400 | 8,1523328 | 48,96 |
| 450 | 11,6828176 | 66,65 |
| 500 | 16,1250816 | 91,25 |





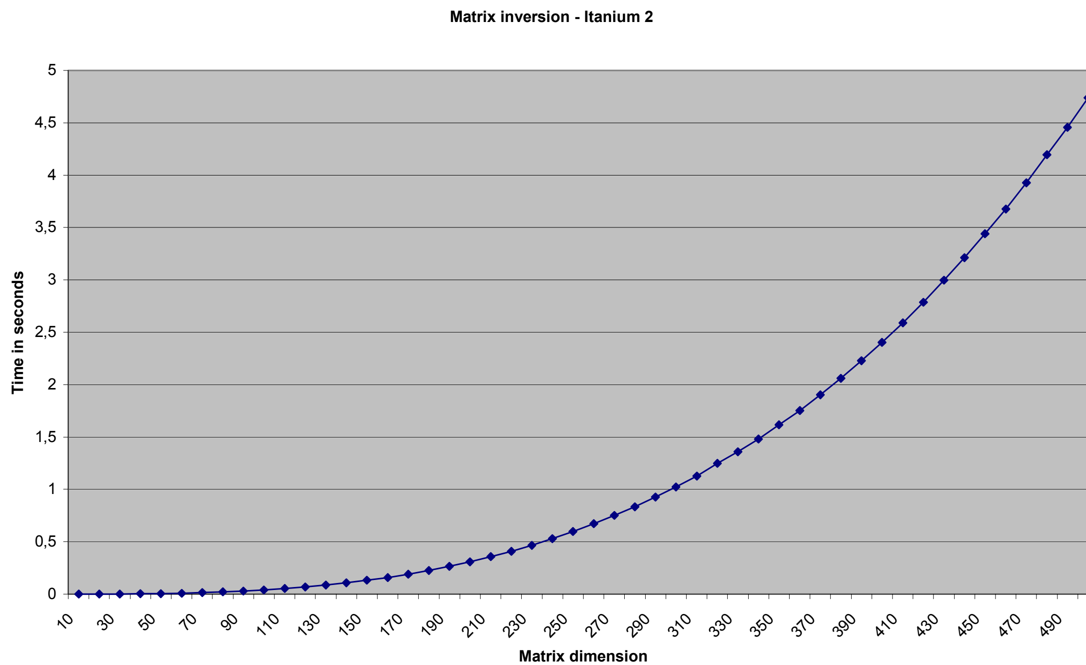

Comparison between both processors:

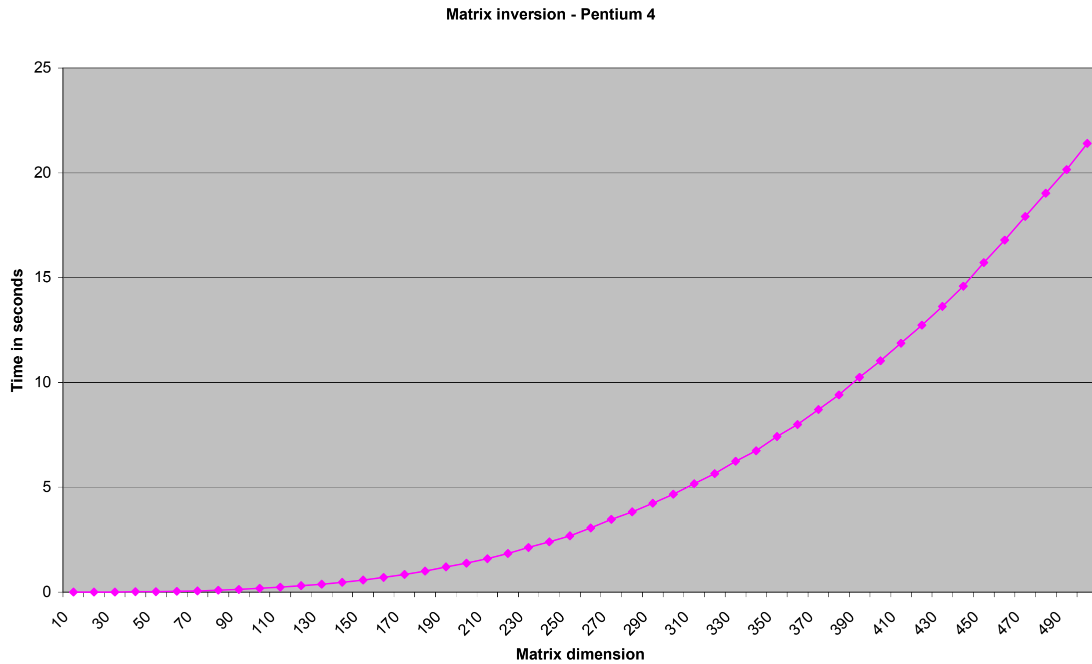


#### 6.3.1.3 Matrix inversion

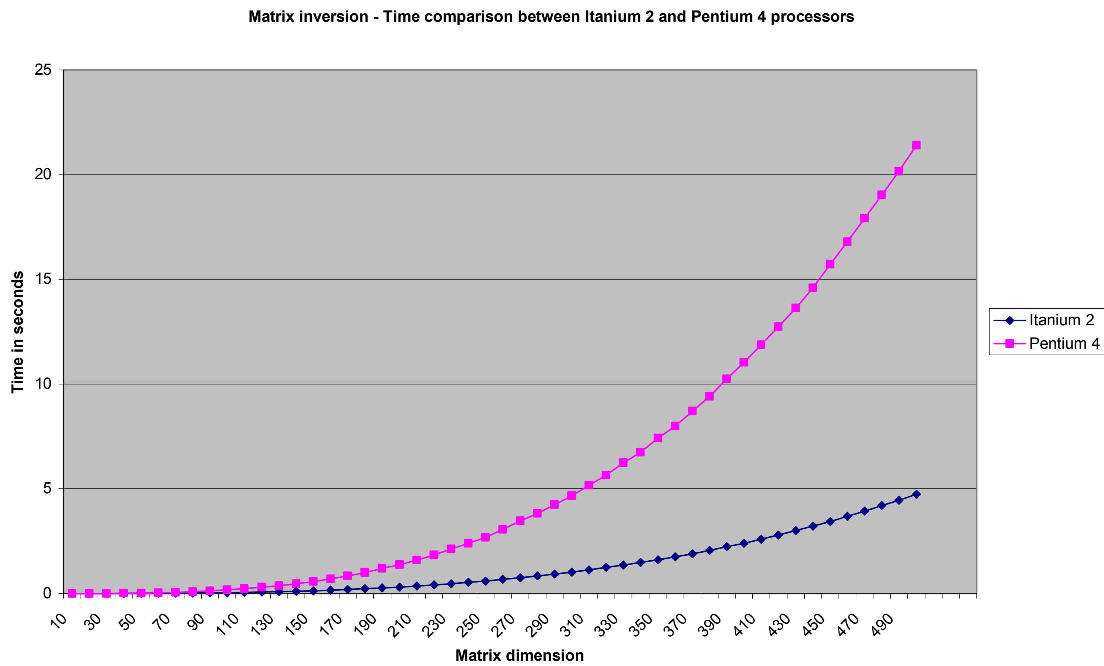
CPU times in seconds required by both processors for inverting the matrix with the $c_i$ constants and different dimensions.

| Dimension | Time (Itanium 2) | Time (Pentium 4) |
|---|---|---|
| 50 | 0,00488 | 0,02 |
| 100 | 0,03904 | 0,18 |
| 150 | 0,13176 | 0,57 |
| 200 | 0,30744 | 1,38 |
| 250 | 0,597312 | 2,69 |
| 300 | 1,023824 | 4,66 |
| 350 | 1,61528 | 7,42 |
| 400 | 2,403888 | 11,04 |
| 450 | 3,439424 | 15,72 |
| 500 | 4,740432 | 21,4 |






Comparison between both processors:




### 6.3.2 Precision

The precision measurements were also obtained using different dimensions.

The values correspond first to the infinite norm of the vector of differences between the exact and the computed values.

$\underset{1 \le i \le n}{Max} |v_i - v_i *|,$ where $v$ and $v*$ are the vectors of computed and exact values

and also to the $l_1$*-norm*

$\sum_{i=1}^{n} |v_i - v_i *|,$ where $v$ and $v*$ are as before

#### 6.3.2.1 LU factorization

**Infinite norm**

| | 10 | 15 | 20 | 24 |
|---|---|---|---|---|
| Itanium 2 | 2,971E-22 | 5,6805370049E-15 | 3,72241477721501E-07 | 9,5870305554838E-02 |
| Pentium 4 | 2,971E-22 | 5,6805370049E-15 | 3,72241477721501E-07 | 9,5870305554838E-02 |

***$l_1$-norm***

| | 10 | 15 | 20 | 24 |
|---|---|---|---|---|
| Itanium 2 | 1,0301E-21 | 2,30009592474E-14 | 1,79967885602667E-06 | 4,99060843597718E-01 |
| Pentium 4 | 1,0301E-21 | 2,30009592474E-14 | 1,79967885602667E-06 | 4,99060843597718E-01 |

#### 6.3.2.2 Cholesky factorization

**Infinite norm**

| | 100 | 500 | 1000 |
|---|---|---|---|
| Itanium 2 | 5,434714256517010E-02 | 1,771707099214840E-02 | 2,446785889680890E-01 |
| Pentium 4 | 5,434714256517010E-02 | 1,771707099214850E-02 | 2,446785889680920E-01 |

***$l_1$-norm***

| | 100 | 500 | 1000 |
|---|---|---|---|
| Itanium 2 | 1,086942606741395922E-01 | 3,5434134012200E-02 | 4,8935706784499E-01 |
| Pentium 4 | 1,086942606741395644E-01 | 3,5434134012218E-02 | 4,8935706784490E-01 |

#### 6.3.2.3 Matrix inversion

The results for this benchmark were extremely accurate when all the coefficients were taken as

$c_i = 1 - 9.E\text{-}17$ for all i.
Both processors gave underflow for values closer to 1.

**Infinite norm**

| | 100 | 500 | 1000 |
|---|---|---|---|
| Itanium 2 | 0 | 0 | 0 |
| Pentium 4 | 0 | 0 | 0 |

***$l_1$-norm***

| | 100 | 500 | 1000 |
|---|---|---|---|
| Itanium 2 | 0 | 0 | 0 |
| Pentium 4 | 0 | 0 | 0 |

## 6.4 Analysis of the results

The precisions attained with both processors using the quadruple data type are practically identical.

However, the CPU times obtained with the Itanium 2 were much lower than those of the Pentium 4.

The fact that the IA-64 offers quadruple floating point support by means of its instruction set and parallelism capabilities constitutes the key factor for solving problems with very good precision using CPU times that are much lower than the ones required by the IA-32 architecture.

Among the characteristics that are part of that support are the 64bits registers, the greater number of registers both integer and floating point, division by software and the fused operation.

As a concrete example we can mention that the mantissa $n_0 ... n_{112}$ of a quadruple precision number $N$ can be written as:

$$N = N_h + N_l = n_0 ... n_{48} 2^{64} + n_{49} ... n_{112} = n_h 2^{64} + n_l ,$$

where $n_0 ... n_{48}$ y $n_{49} ... n_{112}$ are binary integers. This allows that several operations can be implemented using arithmetic instructions for 64 bits integers both for the exponents and mantissas of the operands. Those integer operations are provided by hardware (addition, subtraction, multiplication) or by software (division) and help to avoid the use of floating point numbers.

## 6.5 Recommendations for scientific programming

The use of this sort of data types is very convenient for solving problems accurately using very reasonable CPU times.

There are examples of the use of this data type in high level languages as REAL*16 in Fortran or_Quad available in the Intel's C compiler. The later was the one used for these benchmarks although this kind of floating point data type in C is less developed.

Something to be taken into account is that arithmetic in other precisions resembles the one in quadruple precision, and therefore the benefits provided by the architecture could be extended to other sort of values.

# 7 Conclusions

We have made tests with different compilers, but in most cases the one that offered the best results was the Intel's "icc". Moreover, when using the PGO (Profile Guided Optimization) options the resulting CPU times were by far the best.

Once the compiler had been chosen, different analyses were made to show the advantages of the IA-64 architecture.

We have made a comparative analysis of the FMA fused operation provided by IA-64 in regard to what is possible to get using the IA-32. From such analysis we observed that the FMA fused operation did not introduce the additional rounding error of the multiplication that arises in the IA-32 computers because of the use of separate operations. This brings sensible benefits both from the point of view of the precision and the needed CPU times.

The analysis made comparing the standard IEEE-754 floating point and the possibility of using extended exponents in the IA-64 machines gave as expected the result that the underflow/overflow conditions were more difficult to attain in the latter case. However, the compiler is a key factor for getting the benefits that the new architecture offers, as the large number of available FP registers. In that case the compiler can generate much more efficient code and this is a clear advantage of the IA-64 architecture in regard to the IA-32 one.

When using quadruple precision the results were remarkable. Both architectures must perform the calculations by software and, as expected, the resulting precisions were absolutely alike. However, the benefits of the Itanium processor led to much better CPU times because it uses internal parallelism, greater number of registers, division by software, and the fused operation. We do believe this is an outstanding and real example of the benefits offered by the Itanium architecture.

***Bibliography***


1. Scientific Computing on Itanium-based Systems, M.Cornea, J.Harrison, P.T.P.Tang, Intel Press, 2002.
2. Numerical methods for least squares problems, A.Björck, SIAM, 1996.
3. Matrix Computations, G.H. Golub and C.F. Van Loan, Johns Hopkins Univ. Press, 2nd ed., 1989.
4. Applied Numerical Linear Algebra, J.Demmel, SIAM, 1997.
5. Intel Itanium Architecture Software Developer`s Manual. Volume 1: Application Architecture. Revision 2.1. Intel Corporation, 2002.
6. Intel Itanium Architecture Software Developer`s Manual. Volume 2: System Architecture. Revision 2.1. Intel Corporation, 2002.
7. Intel Itanium Architecture Software Developer`s Manual. Volume 3: Instruction Set Reference. Revision 2.1. Intel Corporation, 2002.