

# Efficient SIMD Numerical Interpolation

Hossein Ahmadi, Maryam Moslemi Naeini, Hamid Sarbazi-azad

Sharif University of Technology, Tehran, Iran.  
IPM School of Computer Science, Tehran, Iran.  
{h\_ahmadi, moslemi}@ce.sharif.edu, azad@sharif.edu

**Abstract.** This paper reports the results of SIMD implementation of a number of interpolation algorithms on common personal computers. These methods fit a curve on some given input points for which a mathematical function form is not known. We have implemented four widely used methods using vector processing capabilities embedded in Pentium processors. By using SSE (streaming SIMD extension) we could perform all operations on four packed single-precision (32-bit) floating point values simultaneously. Therefore, the running time decreases three times or even more depending on the number of points and the interpolation method. We have implemented four interpolation methods using SSE technology then analyzed their speedup as a function of the number of points being interpolated. A comparison between characteristics of developed vector algorithms is also presented.

## 1 Introduction

Many interpolation algorithms for various applications have been introduced in the literature [6]. In general, the process of determining the value of a specific point using a set of given points and their corresponding values is named *interpolation*. In other words, interpolation means to fit a curve on a set of points. Interpolation on large number of points requires a great amount of computation power and memory space. Parallelism is one of the most practical approaches to increase the performance of different interpolation methods [3, 4, 5].

Since the arrival of modern microprocessors with single-instruction multiple-data stream (SIMD) processing extensions, little effort has been made to increase the performance of time-consuming algorithms using the SIMD computational model [1, 2, 10]. In the SIMD model, processors perform one instruction on multiple data operands instead of one operand as scalar processors do. In other words, in the SIMD model data operands are vectors. Therefore, to reach an appropriate speedup using the SIMD model we should focus on packing data elements to form data vectors.

In this paper, we show how various interpolation methods can adapt to gain speedup using SIMD computational model. The fact that how these methods implicitly allow parallelism greatly affects the amount of speedup that can be achieved by vectorizing them. The discussed interpolation methods are Lagrange, Newton-Gregory forward, Gauss forward, and B-Spline. For each of these methods, a generic

vector algorithm is designed and then implemented in assembly and C++ (using their SIMD instruction sets) for Pentium processor.

The rest of paper is organized as follows. Section 2 introduces SIMD processing and available technology in current SIMD microprocessors. In Section 3, four common interpolation methods are briefly introduced. The vector algorithms for SIMD computation of the four interpolation methods are presented in Section 4. In Section 5, experimental evaluation of the implemented algorithms is reported. Finally, conclusions and future work in this line are suggested in Section 6.

## 2 SIMD Processing

The SIMD processing model allows performing a given operation on a set of data instead of a single (scalar) data. Such a computational model is now supported by many new processors in the form of SIMD extensions. Examples of such extensions are Intel's MMX, SSE, SSE2 and SSE3 technologies [7], AMD's 3DNow! technology [8], Motorola's AltiVec technology implemented in PowerPCs, and Sun's VIS technology [12]. We can assume these units as a vector processor with different vector sizes. For example 3DNow! Technology uses 8 packed 64-bit register to perform SIMD operations. When single precision floating-point data elements are used, 3DNow! extension appears as a 2-element vector processing model, while SSE technology using 128-bit registers can perform operations on 4 single precision floating-point elements simultaneously, which means a 4-element vector processing model.

We will focus our implementations on one of the most popular technologies, Intel's SSE technology. The packed single-precision floating-point instructions in SSE technology perform SIMD operations on packed single-precision floating-point operands. Each source operand contains four single-precision floating-point (32-bit) values, and the destination operand contains results of the parallel operation performed on corresponding values of the two source operands. A SIMD operation in Pentium 4 SSE technology can be performed on 4 data pairs of 32-bit floating-point numbers.

In addition to packed operations, the scalar single-precision floating-point instructions operate on the low (least significant) double words of the two source operands. The three most significant double words of the first source operand are passed through to the destination. The scalar operations are similar to the floating-point operations in the sequential form. It is important to mention that the proposed algorithms are designed to be implemented on any processor supporting vector or SIMD operations. Consequently, all algorithms are described with a parameter  $k$ , independent of the implementation, which represents vector size of the vector processor and, in a way, indicates the level of parallelism.

## 3 Interpolation methods

In this paper, we consider four well-known interpolation methods: Lagrange, Newton-Gregory forward, Gauss forward and B-Spline. Lagrange interpolation is one of

the most appropriate methods to be executed on a vector processor. Moreover, the simplicity of sequential implementations makes this method be used in many interpolation applications. Lagrange interpolation, for a set of given points  $(x_m, y_m)$ ,  $0 \leq m \leq N - 1$ , in point  $x$  is carried out as

$$f(x) = \sum_{m=0}^{N-1} L_m(x) y_m \quad (1)$$

where  $L_m(x)$  is called Lagrange polynomial [6], and is given by:

$$L_m(x) = \frac{\prod_{\substack{0 \leq i \leq N-1 \\ i \neq m}} (x - x_i)}{\prod_{\substack{0 \leq i \leq N-1 \\ i \neq m}} (x_m - x_i)} \quad (2)$$

It requires a long computation time to carry out the above computation in sequential form. Hence, this interpolation method is usually implemented on a number of parallel systems with different topologies, such as the k-ary n-cube [3].

Newton-Gregory forward method is based on a difference table. The table contains elements as [6]

$$\Delta^k f_n = \Delta^{k-1} f_{n+1} - \Delta^{k-1} f_n, \quad \Delta f_n = f_{n+1} - f_n \quad (3)$$

and the computation for equally spaced input points can be realized as

$$f(x) = f_0 + \binom{s}{1} \Delta f_0 + \binom{s}{2} \Delta^2 f_0 + \dots + \binom{s}{n} \Delta^n f_0, \quad s = \frac{x - x_0}{h} \quad (4)$$

where  $h$  is the difference between  $x$  values.

Similarly, Gauss forward method uses a difference table; indeed, it operates on different path in the difference table. The Gauss forward interpolation for given points  $(x_m, y_m)$ ,  $-N/2 \leq m \leq (N+1)/2$ , can be formulated as [6]

$$f(x) = f_0 + \binom{s}{1} \Delta f_0 + \binom{s}{2} \Delta^2 f_{-1} + \binom{s+1}{3} \Delta^3 f_{-1} + \binom{s+1}{4} \Delta^4 f_{-2} + \dots + \binom{s + [(n-1)/2]}{n} \Delta^n f_{-(n/2)} \quad (5)$$

B-Spline is probably one of the most important methods in surface fitting applications. It has many applications especially in computer graphics and image processing. B-Spline interpolation is carried out as follows [8]:

$$y = f_i (x - x_i)^3 + g_i (x - x_i)^2 + h_i (x - x_i) + k_i, \quad \text{for } i = 1, 2, \dots, n-1, \quad (6)$$

By applying boundary and smoothness conditions in all points, parameters  $f$ ,  $g$ ,  $h$ , and  $k$  can be obtained as:

$$f_i = \frac{D_{i+1} - D_i}{6l_i}, \quad g_i = \frac{D_i}{2}, \quad h_i = \frac{y_{i+1} - y_i}{l_i} - \frac{l_i D_{i+1} + 2l_i D_i}{6}, \quad k_i = y_i. \quad (7)$$

while  $D$  is the result of a tri-diagonal equations system, and  $l_i = x_{i+1} - x_i$ .

$$\begin{aligned} 2(l_0 + l_1)D_1 + l_1D_2 &= m_1 \\ l_{i-1}D_{i-1} + 2(l_{i-1} + l_i)D_i + l_iD_{i+1} &= m_i, \quad 2 \leq i \leq n-2 \\ l_{n-2}D_{n-2} + 2(l_{n-2} + l_{n-1})D_{n-1} &= m_{n-1} \end{aligned} \quad (8)$$

$$m_i = 6\left(\frac{y_{i+1} - y_i}{l_i} - \frac{y_i - y_{i-1}}{l_{i-1}}\right)$$

#### 4 Implemented Algorithms

To achieve the highest speedup, one can try to use vector operations for the interpolation process as much as possible. Using vector computing independent operations on different data elements in a  $k$ -element vector, a speedup of  $k$  is expected. Some other factors on SSE technology also help us to make our algorithm gain a speedup of more than  $k$  due to fewer accesses to memory hierarchy, faster data fetching from internal cache, and better use of pipelining. Obviously, for the methods with low data dependencies between different steps, better speedup can be obtained.

For any of the four methods, a well optimized vector algorithm running on a general vector processor is designed and implemented using Pentium 4's SSE instruction set. In what follows, we briefly explain the implementation of Lagrange, Newton-Gregory forward, Gauss, and B-Spline interpolation techniques. Next section reports experimental results for the performance evaluation of implemented algorithms.

For Lagrange interpolation, first, the common dividend in all Lagrange factors,  $L_i(x)$ , is computed. Each factor can be derived by dividing this common dividend with  $(x - x_i)$ . Next, for each factor the divisor is calculated. After computing Lagrange factors, the final value is calculated using Eq. (2). All operations involved in these steps can effectively exploit SIMD operations in SSE; thus, a noticeable speedup is expected.

The proposed algorithm for B-Spline interpolation is based on solving tri-diagonal equations system associated with  $D$  values. The method is based on simultaneous substitution in Gauss-Jordan method solving a system of equations [9]. More detailed, in the first iteration, values of  $l$  and  $m$  are initialized in a completely parallel manner. In the second iteration, values for  $b'$  and  $d'$  are the results of elimination of  $x_{i-1}$  from  $i$ -th equation which are computed for all  $k$  values. In the last iteration back-substitution is performed to obtain  $x_i$  values.  $\log(k)$  operations are necessary to eliminate or back-substitute  $k$  elements. Therefore, we may expect a speedup of  $k /$

$\log(k)$ , which is 2 for SSE technology with  $k = 4$  ( $4/\log(4) = 2$ ). Figure 1 shows a pseudo code of the vector algorithm for B-Spline method.

```

for i ← 0 to (n/k)
  [li×k, li×k+1, ..., li×k+k-1] ← [xi×k+1, xi×k+2, ..., xi×k+k] - [xi×k, xi×k+1, ..., xi×k+k-1]
  Y ← ([yi×k+1, yi×k+2, ..., yi×k+k] - [yi×k, yi×k+1, ..., yi×k+k-1]) / [li×k, li×k+1, ..., li×k+k-1]
  [mi×k, mi×k+1, ..., mi×k+k-1] ← (Y - Y') / 6
for i ← 0 to (n/k)
  V1 ← [mi×k, mi×k+1, ..., mi×k+k-1]
  V2 ← [-li×k / d'i×(k-1), -li×k+1 / d'i×(k-1)+1, ..., -li×k+k-1 / d'i×(k-1)+k-1]
  V1 ← V1 + V1(1) × V2
  V2 ← V2 × V2(1)
  [b'i×k, b'i×k+1, ..., b'i×k+k-1] ← V1 + V1(2) × V2
  V1 ← -[li×k, li×k+1, ..., li×k+k-1] × [li×k, li×k+1, ..., li×k+k-1]
  V2 ← 2 × ([0, li×k+1, ..., li×k+k-1] + [0, li×k+2, ..., li×k+k])
  V3 ← [1, 0, 0, ..., 0]
  V4 ← [1, 1, ..., 1]
  V1' ← V1 × V3(1) + V2 × V1(1)
  V2' ← V1 × V4(1) + V2 × V2(1)
  V3' ← V3 × V3(1) + V4 × V1(1)
  V4' ← V3 × V4(1) + V4 × V2(1)
  [d'i×k, d'i×k+1, ..., d'i×k+k-1] ← (V1' × V3'(2) + V2' × V1'(2)) / (V3' × V3'(2) + V4' × V1'(2))
for i ← (n/k) to 0
  V1 ← [b'i×k / d'i×k, b'i×k+1 / d'i×k+1, ..., b'i×k+k-1 / d'i×k+k-1]
  V2 ← [-li×k-1 / d'i×(k-1), -li×k / d'i×(k-1)+1, ..., -li×k+k-2 / d'i×(k-1)+k-1]
  V1 ← V1 + V1(1) × V2
  V2 ← V2 × V2(1)
  [Di×k, Di×k+1, ..., Di×k+k-1] ← V1 + V1(2) × V2

```

**Fig. 1.** Pseudo code for B-Spline method

For Newton-Gregory forward method, we begin with the computation of the difference table. Then, each  $\Delta f_i$  value is multiplied by  $\binom{s}{i}$  and added to  $S$ , and the sum of calculated terms is computed. Adopting the same technique, we can realize the Gauss forward method as well.

Both methods, Newton-Gregory and Gauss, use a sequential summation procedure to carry out  $f(x)$ . Therefore, the parallelism is exposed only in computing the differ-

ence table. Because of great data dependency between successive steps in these methods, very high speedup cannot be expected.

## 5 Performance Analysis

As the Intel's SSE technology is the most common SIMD extension used in today's personal computers, we implemented the presented algorithms on Intel's Pentium 4 processor using single precision floating point. Thus, an approximate performance gain of 4 is expected. The speedup is computed with two different reference implementations: the SIMD C++ and assembly codes, and their equivalent sequential (non-SIMD) codes. We analyze the performance of implemented methods by speedup curves as a function of the number of points being interpolated.

The system used to perform our performance analysis had the following configuration:

*CPU: Intel Pentium 4 Mobile Processor at 1.8 GHz*

*Main memory: 512 MB of DDR (266 MHz) RAM*

*Cache memory: 512KB L2 Cache*

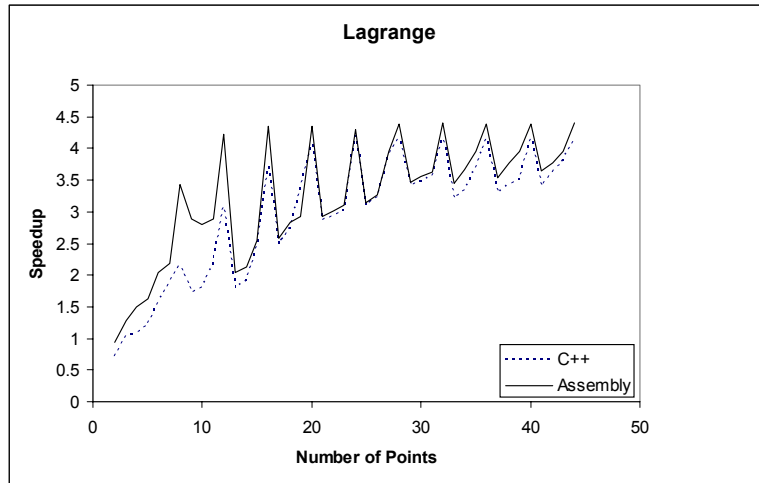
*Operating system: Microsoft Windows XP Professional Edition*

*Compiler: Microsoft Visual Studio .NET 2003*

To compute the speedup for a given number of interpolated points, execution time of the SIMD code is divided by the execution time of equivalent sequential code.

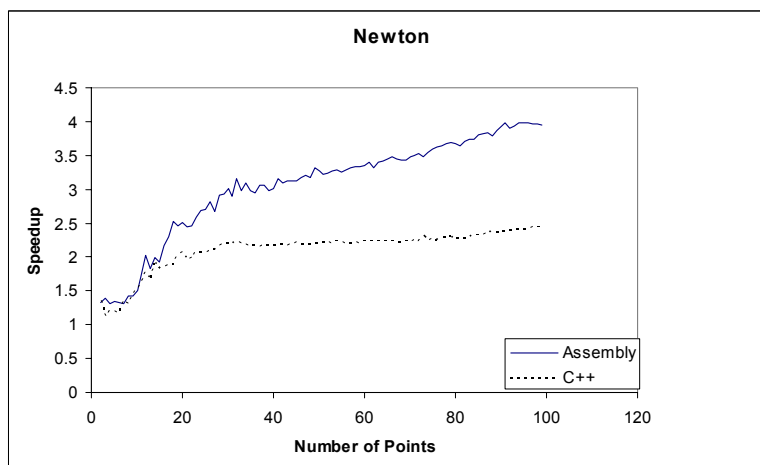
Figure 2 shows the speedup for Lagrange interpolation method. It is predictable that speedup grows as the number of input points increases because the effect of sequential parts of the program in the total running time decreases. The sequential parts of the code consist of loop control codes, initialization codes, and some serial arithmetic operations which have a high data dependency in different steps.

There are also fluctuations in the graph with peaks on multiple of 4, the vector length in SSE extension. This effect is common in register-register vector computing architecture [11]. When the number of interpolated points is not a multiple of 4, operations on the last data block has less speedup than previous data blocks; therefore, the total speedup will be reduced. By increasing the number of interpolated points, operations on the last data block get less portion of total execution time and fluctuations will diminish. Performing Lagrange interpolation on more than 50 single-precision floating point numbers will cause the result to loose the precision. That is because of large number of factors in Eq. (2). Hence, the graph of Lagrange interpolation is drawn only up to 50 points.

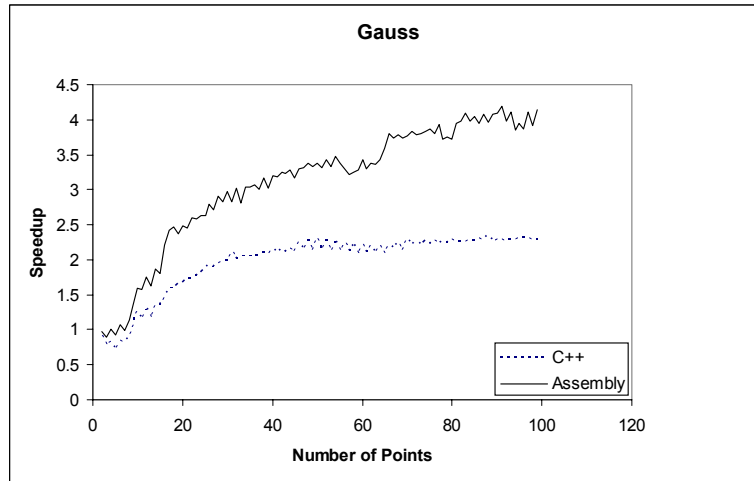


**Fig. 2.** Speedup of Lagrange method

Note that in Newton and Gauss methods, as shown in Figure 3 and Figure 4, the ripple in the speedup curves is negligible in contrast with Lagrange method. The reason is the varying size of the difference table during computation. The table size can be multiple of 4 in the first step but not in following three steps. Consequently, the initial table size does not result in visible extra instruction execution and a speedup drop.

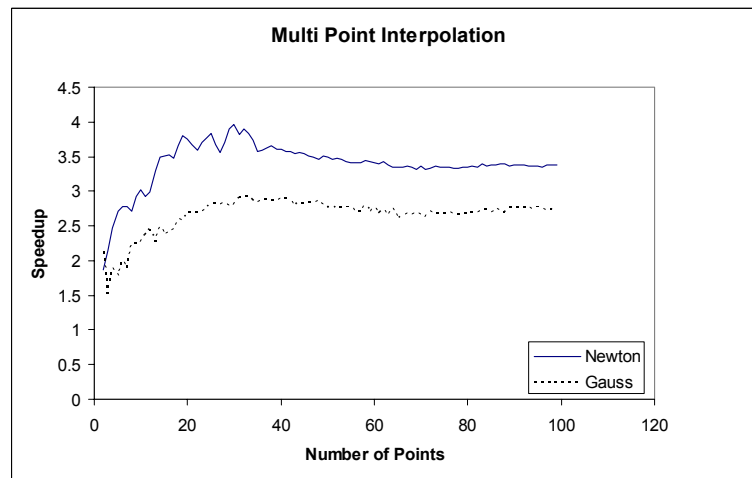


**Fig. 3.** Speedup of Newton-Gregory forward method



**Fig. 4.** Speedup of Gauss forward method

In many applications it is required to carry out the value of interpolated curve in several points. In this case, a parallel algorithm is designed based on Newton-Gregory forward and Gauss forward methods to interpolate at  $k$  points simultaneously. In this implementation a higher amount of speedup is gained which is presented in Figure 5.



**Fig. 5.** Speedup of Gauss and Newton-Gregory forward methods on interpolating multi points

For B-Spline, as it was discussed in previous section, a speedup of about 2 is obtained. Similar to Lagrange method, fluctuations are completely visible in B-Spline method because of its noticeable extra process for the points. The interesting behavior shown in the figure is the better performance achieved by the C++ code with respect

to the assembly code. This is due to the low-performance code generated by the compiler for sequential C++ code.

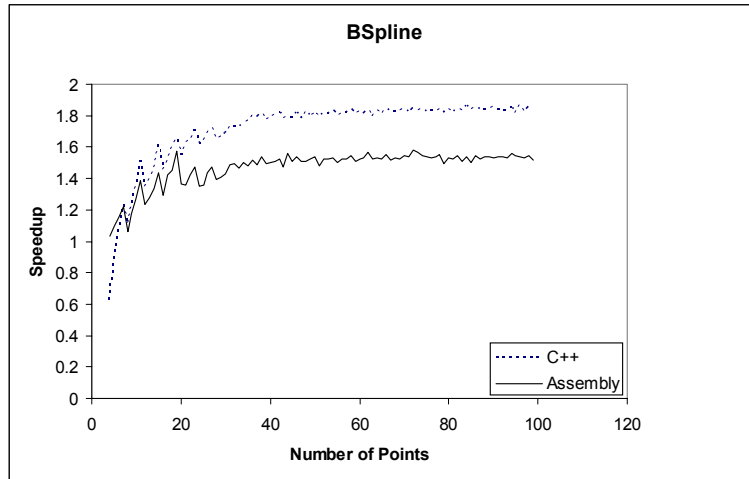


Fig. 6. Speedup of B-Spline method

Finally, a comparison between the gained speedup of different methods is presented in Table 1. As it is shown in the table, Lagrange and B-Spline interpolation algorithms achieved the highest and the lowest speedup respectively.

Table 1. Speedup of four interpolation algorithms for 40 input points.

|                         | Lagrange | Newton forward | Gauss forward | B-Spline |
|-------------------------|----------|----------------|---------------|----------|
| Assembly Implementation | 4.44     | 3.17           | 3.24          | 1.56     |
| C++ Implementation      | 4.16     | 2.21           | 2.23          | 1.81     |

## 6 Conclusion

We designed and implemented SIMD codes for four interpolation methods, namely Lagrange, Newton-Gregory forward, Gauss forward, and B-Spline, using Intel's SSE extension in Pentium 4 processors. Our performance analysis showed a noticeable speedup achieved for each case, ranging from 1.5 to 4.5.

Results showed that Lagrange method can achieve a high performance when executed in SIMD mode. In the case of interpolating multiple points, Newton-Gregory forward and Gauss forward methods also exhibit a good speedup. The completely dependent operations in B-Spline method make it very difficult to run in a parallel way.

The use of other SIMD extensions and comparing their effectiveness, for implementing interpolation techniques can be considered as future work in this line. Also, studying other interpolation techniques and implementing them using SIMD operations is also another potential future work.

## References

1. A. Strey, On the suitability of SIMD extensions for neural network simulation, *Microprocessors and Microsystems*, Volume 27, Issue 7, pp. 341-35, 2003.
2. O. Acicmez, Fast Hashing on Pentium SIMD Architecture, *M.S. Thesis, School of Electrical Engineering and Computer Science*, Oregon State University, May 11, 2004.
3. H. Sarbazi-Azad, L. M. Mackenzie, and M. Ould-Khaoua, Employing k-ary n-cubes for parallel Lagrange interpolation, *Parallel Algorithms and Applications*, Vol. 16, pp. 283-299, 2001.
4. B. Goertzel, Lagrange interpolation on a tree of processors with ring connections, *Journal Parallel and Distributed Computing*, Vol. 22, pp. 321-323, 1994.
5. H. Sarbazi-Azad, L. M. Mackenzie, and M. Ould-Khaoua, A parallel algorithm-architecture for Lagrange interpolation, *Proc. Parallel and Distributed Processing Techniques and Applications*, Las Vegas, pp. 1455-1458, 1998.
6. C. F. Gerald and P.O. Wheatley, *Applied Numerical Analysis*, 4<sup>th</sup> ed., Addison-Wesley, New York, 1989.
7. Intel Corporation, Intel IA-32 Architecture Developer's Manual, 2004.
8. AMD Corporation, 3DNow! Developer's Manual, 2003.
9. Chung, Lin, and Chen, Cost Optimal Parallel B-Spline Interpolations, *ACM SIGARCH Computer Architecture News*, Vol. 18, Issue 3, pp. 121 – 131, 1990.
10. S.Thakkar, T.Huff, Internet Streaming SIMD extensions, *Computer*, Vol.32, no.12, pp. 26 - 34, 1999.
11. K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.
12. Sun Microsystems, VIS Instruction Set User's Manual, November 2003, <http://www.sun.com/processors/vis/>