Matrix calculations with SIMD floating point instructions on x86 processors

André Muezerie[1], Raul J. Nakashima[2], Gonzalo Travieso[3], Jan Slaets[4]

[1, 2, 3, 4] Departmento de Física e Informática, Instituto de Física de São Carlos, Universidade de São Paulo
Av. Dr. Carlos Botelho, 1465 CEP 13560-250 - São Carlos - SP - Brazil
[1]{andremuz@if.sc.usp.br}, [2]{junji@if.sc.usp.br}, [3]{gonzalo@if.sc.usp.br}, [4]{jan@if.sc.usp.br}

Abstract—

This paper describes and evaluates the use of SIMD floating point instructions for scientific calculations. The performance of these instructions is compared with ordinary floating point code. Implementation concerns, the effects of loop unroll as well as matrix size variations are analyzed. Execution speeds are compared using matrix multiplication. The intrinsic incompatibility of the SIMD floating point implementations used by different manufacturers requires the use of two different instruction sets: 3DNOW! on the AMD K6 processor and the Streaming-SIMD Extensions (SSE) on the Intel Pentium III processor.

Keywords— **SIMD, 3DNOW!, SSE, vector operations, performance evaluation.**

I. INTRODUCTION

Today's microprocessors employ nearly all the performance enhancing techniques used in older mainframes and super-computers. Modern PC processors are now superscalar and their microarchitectural features, based on pipelining and instruction level parallelism, overlapping instructions on separate resources, made it possible to obtain extraordinary levels of parallelism and speeds [DIE 99].

The fast growing multimedia applications, demanding significant processing speeds, led to the implementation of new instructions to improve mathematical calculations used in video processing and speech recognition as well as DSP functionality as used in soft modems, ADSL, MP3 and Dolby Digital surround sound processing. These new instructions are implemented using Single Instruction Multiple Data (SIMD) technology in which a single instruction operates in parallel on multiple pieces of data using superscalar implementations [INT 97]. In 1998 a set of SIMD floating point instructions were made available on the AMD-K6 processors with the implementation of the 3DNow! technology [AMD 99a]. Nine months later Intel delivered the Pentium III with "Internet Streaming SIMD Extensions" denominated SSE. Up to now the use of these instructions is limited to specialized multimedia and DSP applications.

In this paper we illustrate and evaluate the use of these SIMD instructions in mathematical vector operations. To verify the speedup obtained on an AMD-K6 processor a matrix multiplication routine written in C is compared with an assembler version making use of the SIMD floating point instructions. The same algorithm is also used to evaluate loop unrolls exploring the functionality of the multiple pipelines and execution units present in the evaluated microprocessor architectures. Figures illustrate the performance improvements obtained exploring loop unrolls and SIMD instructions on a Pentium III and an AMD-K6 processor. Matrix multiplication was chosen as a benchmark because of its use of linear algebra operations important in many algorithms. The principles applied in the examples used can be easily extended to other mathematical calculations where vector operations can be identified.

II. OVERVIEW OF THE SIMD IMPLEMENTATIONS

The first implementation of vector operations in the microarchitecture of x86 type processors was obtained with Intel's MMX technology. This new technology added 57 new instructions and 4 new data types (byte, word, double word and quadword integers) to the x86 basic architecture. In 1996 Intel publicly released the details of the MMX technology turning it part of the today's industry-standard x86 instruction set processors.

Unfortunately different manufacturers added completely different architectural implementations of floating point SIMD extensions to their x86 microprocessors.

The AMD-K6 implementation.

The architecture of the K6 processor implements the 3DNow! as a simple extension of the MMX instruction set [AMD 99b][AMD 99c]. The floating-point vector operations are performed on eight logical 3DNow! registers of 64 bits each. This added features permits the simultaneous calculation of two vector operations resulting in a parallel execution of four single precision floating-point operations. The "femms" instruction enables the user to switch between the x87 instruction set and the 3Dnow! or MMX instructions.

This simple implementation requires no additional support from the operating system. However, programmer

attention is required in the case of a mixed use of 3DNow and MMX instructions since the related register sets are overlapped.

The Pentium III implementation.

The Pentium III architecture allows 4 single precision floating-point operations to be carried out with a single instruction. Architecturally the SIMD-FP feature introduces a new register file containing eight 128-bit registers, each capable of holding a vector of four single precision floating-point elements [ABE 99][MAC 99][INT 99].

Since a new dedicated register set is used to implement the FP vector operations in the Pentium III architecture, operating system support is needed to save these new SSE registers in a multiprogramming environment.

This implementation is, from the programmers' point of view, a little bit more complicated due to the need of SO support and the requirement of special initialization instructions as pointed out in the description of the test programs below.

III. The Matrix Multiplication Test program

A simple matrix multiplication test program is written in C language with the matrix elements stored row by row in consecutive memory positions as shown in list 1. This sequential matrix storage technique is adopted to optimize memory and cache access. This alignment enhances also the needed memory to vector register transfers used in the SIMD floating point implementations of the test programs.

The basic C language routine, here denominated "c1", is modified as shown in list 2 and 3 implementing loop unrolls of 2 and 4 as will be needed later by the vector implementations of the program. We will call the C programs with loop unrolls respectively "c2" and "c4". The "C" code is compiled and linked with the gcc compiler as described in the following section.

```
int prod_c_c1(float *x, float *y, float *z,
long int N)
{register long int i, j, k;
  for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
     z[i*N+k] += x[i*N+j] * y[j*N+k];
   return 0;
}
```

List 1 - The c1 basic test program.

```
int prod_c_c2(float *x, float *y, float *z,
long int N)
{register long int i, j, k;
  for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
    for (k = 0; k < N; k+=2)
     {  z[i*N+k] += x[i*N+j] * y[j*N+k];
        z[i*N+k+1] += x[i*N+j] * y[j*N+k+1];
     }
   return 0;
}
```

List 2 - The c2 test routine with a loop unroll of 2.

```
int prod_c_c4(float *x, float *y, float *z,
long int N)
{register long int i, j, k;
  for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
    for (k = 0; k < N; k+=4)
     {  z[i*N+k] += x[i*N+j] * y[j*N+k];
        z[i*N+k+1] += x[i*N+j] * y[j*N+k+1];
        z[i*N+k+2] += x[i*N+j] * y[j*N+k+2];
        z[i*N+k+3] += x[i*N+j] * y[j*N+k+3];
     }
   return 0;
}
```

List 3 - The c4 test program with a loop unroll of 4

The code with vector instructions is obtained editing the core loop of the assembler code produced by the gcc compiler. Since the gcc compiler produces an assembler code in GNU format a conversion to Intel's assembler format is performed with the optimizer [OPT 98] utility. To produce pure x86 code the optimizer program is used without applying any optimizations performing only a GNU to Intel assembler conversion. This conversion is needed because the NASM assembler [NAS 99] which is capable of assembling 3DNow! as well as SSE instructions do not accept GNU format

The vector instructions are introduced into the Intel assembler codes generated from the original C programs in substitution to the ordinary instructions that performed the matrix calculation in the core loop.

For the K6 based systems two versions of the test program are developed and are referred to as a2 for the version with a loop unroll of 2 and a4 for the implementation with a loop unroll of 4. The measuring of the elapsed time was done with the UNIX system call *gettimeofday* leading to result with a resolution of 1 microsecond. The programs were executed in single user mode, to avoid interference from other processes.

The SSE implementation called a4-sse is illustrated in list 6 and uses a loop unroll of 4. In this case a Windows NT time call was used with a resolution of 1/60 seconds to determine the elapsed time.

The assembler code shown in lists 4 to 6 contains the vector instructions introduced in the core of the loop.

```
.L13:
        mov     eax, [ebp-8]
        add     eax, ecx
        mov     edx, [ebp-12]
        add     edx, ecx
        mov     edi, [ebp-20]
        mov     esi, [ebp+8]
        movd    mm0, [esi+edi*4]
        movq    mm3, mm0
        psllq   mm0, 32
        por     mm0, mm3
        mov     edi, [ebp+12]
        movq    mm1, [edi+edx*4]
        pfmul   mm0, mm1
        mov     esi, [ebp+16]
        movq    mm2, [esi+eax*4]
        pfadd   mm0, mm2
        movq    [esi+eax*4], mm0
        add     ecx, 2
        cmp     ecx, [ebp+20]
        jl      near .L13
```

List 4 - The core of the a2 program with loop unroll of 2.

```
.L13:
        mov     eax, [ebp-8]
        add     eax, ecx
        mov     edx, [ebp-12]
        add     edx, ecx
        mov     edi, [ebp+8]
        movd    mm0, [edi+ebx*4]
        movq    mm3, mm0
        psllq   mm0, 32
        por     mm0, mm3
        movq    mm4, mm0
        mov     edi, [ebp+12]
        movq    mm1, [edi+edx*4]
        pfmul   mm0, mm1
        movq    mm2, [esi+eax*4]
        pfadd   mm0, mm2
        movq    [esi+eax*4], mm0
        mov     edi, [ebp+12]
        movq    mm5, [edi+edx*4+8]
        pfmul   mm5, mm4
        movq    mm6, [esi+eax*4+8]
        pfadd   mm5, mm6
        movq    [esi+eax*4+8], mm5
        add     ecx, 4
        cmp     ecx, [ebp+20]
        jl      near .L13
```

List 5 - The core of the a4 program with loop unroll of 4.

```
.L13:
        mov     eax, [ebp-8]
        add     eax, ecx
        mov     edx, [ebp-12]
        add     edx, ecx
        mov     edi, [ebp+8]
        movss   xmm0, [edi+ebx*4]
        shufps  xmm0, xmm0, 0
        mov     edi, [ebp+12]
        movaps  xmm1, [edi+edx*4]
        mulps   xmm0, xmm1
        movaps  xmm2, [esi+eax*4]
        addps   xmm0, xmm2
        movaps  [esi+eax*4], xmm0
        add     ecx, 4
        cmp     ecx, [ebp+20]
        jl      near .L13
```

List 6 - The core of the a4-sse program with loop unroll of 4.

IV. The Test Environment

The single precision floating point vector instructions were evaluated on three systems: two AMD K6 and one Pentium III microprocessor with the following hardware and software configuration:

The K6-III system with external video controller

The AMD K6-III system runs at 450 MHz, is equipped with 128 Mbytes of RAM memory and an 8 Gbytes EIDE Hard Disk. It is based on an Asus P5SB, which uses the SiS® 530 AGP chipset [ASU 99], motherboard with onboard video controller disabled.

The Linux Slackware 7.0 (kernel 2.2.13) operating system is used with nasm version 0.98 and egcs 2.91.66 19990314 (egcs-1.1.2 release)

The K6-III system with onboard video controller

Similar with the previous one, but using the onboard video controller.

The PIII system

The Pentium III processor operating at 400MHz equipped with 128 MB RAM and a 2 Gbytes EIDE Hard Disk mounted on an ASUS P3v133 (VIA® Apollo Pro133 Series Chipset) Motherboard is denominated in this paper as the Pentium III system.

Here the Windows NT 4 with service pack 5 is used as operating system since the Linux Slackware 7.0 didn't offer support for the SSE extensions. To create an UNIX-like API on top of the Win32 API, the Cygwin tool [CYG 00] is used. This enables us to use the same version of nasm and egcs.

V. RESULTS

In this section we compare execution times of the different test program obtained on the K6 and Pentium III systems. For each matrix size the multiplication is performed three times. The average execution time is calculated and used to evaluate the performance.

Figure 1 shows the data obtained on the Pentium III processor executing the simple C program (c1) as well as the C versions with loop unrolls of 2 (c2) and 4 (c4). As expected the best result is obtained using the SSE instructions (a4-sse). This figure illustrates also that no execution gain was obtained with a loop unroll of 2. An intermediate result is obtained with a loop unroll of 4.

A better idea of the relative speedup due to the use of the SSE instructions can be obtained comparing the c4 algorithm with the a4-sse implementation as shown in figure 2. We attribute the initial dispersion of the graphic to the low resolution of the used time function (1/60 seconds). As can be observed, a typical speedup of 1.5 can be obtained. For matrices with dimensions above 350 a lower speedup (about 1.4) is obtained due to the high rate of cache misses and the greater importance of memory performance for the faster SSE implementation.

Similar behaviors are seen in figures 3, 4 and 5 where the performances of the K6 systems are evaluated.

The execution of the c2 routine shows that a speedup, although small, is obtained with this routine on the K6 processor.

The speedup of the more efficient a4 routine is in the K6 about 1.6 for large matrices, better then the 1.4 of the Pentium III. For smaller matrices, the gain is even larger, about 2.2, showing the importance of efficient memory systems for these high-performance instructions.

A comparison of figures 3 and 4 that show the execution times as a function of matrix size and an analysis of the speedups of the versions using the SIMD instructions with respect to the C versions in figures 5 and 6 shows that the effective reduction of memory bandwidth due to the onboard memory controller is very important, specially for the optimized versions using SIMD instructions. The speedup achieved for matrix sizes larger than about 400×400 on the system with onboard controller (about 1.4 for c4/a4) is significantly smaller than on the system with external controller (1.6 for c4/a4). The effect on absolute execution time is even larger: for a matrix size of 1000×1000 the system with onboard video takes 38% more time.

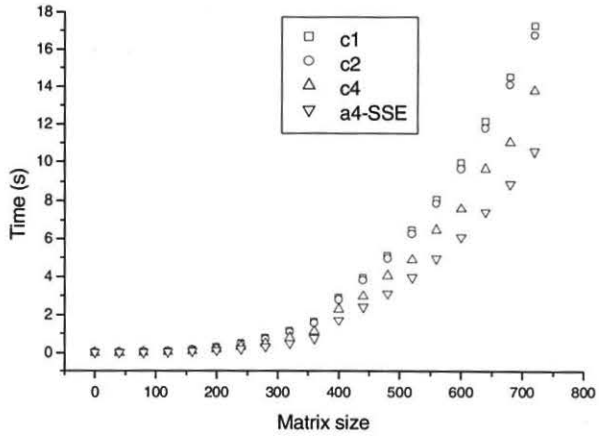

Fig. 1 - Execution times of the four routines on the Pentium III system.

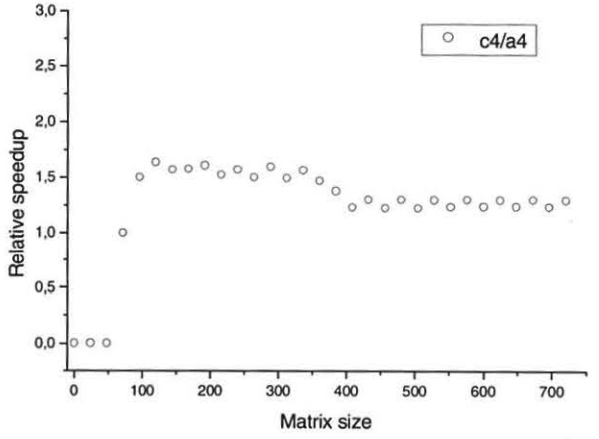

Fig. 2 – Relative speedup of the SSE routine on the Pentium III.

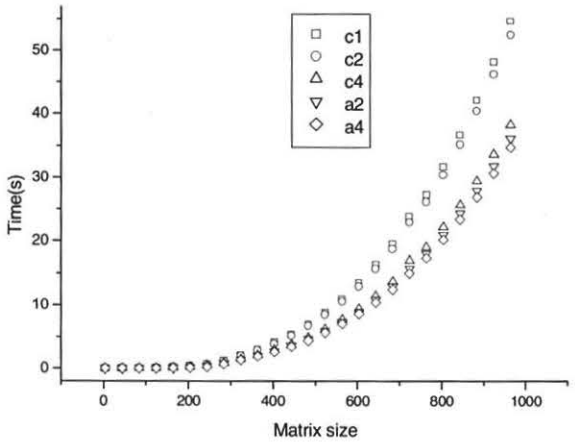

Fig. 3 - Execution times of the five routines on the K6 III system with onboard video card enabled.

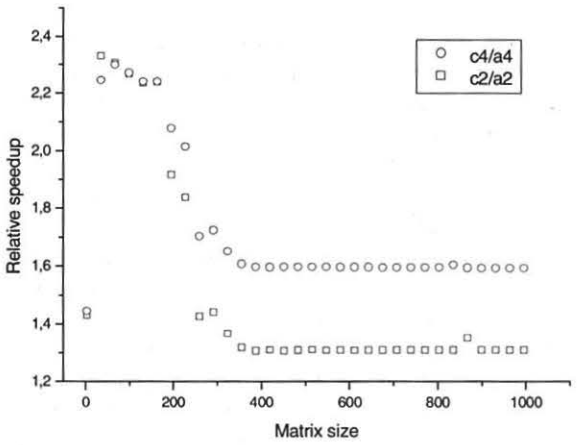

Fig 5 - Relative speedups of the SIMD versions with respect to the C versions on a K6III processor with an external video card.

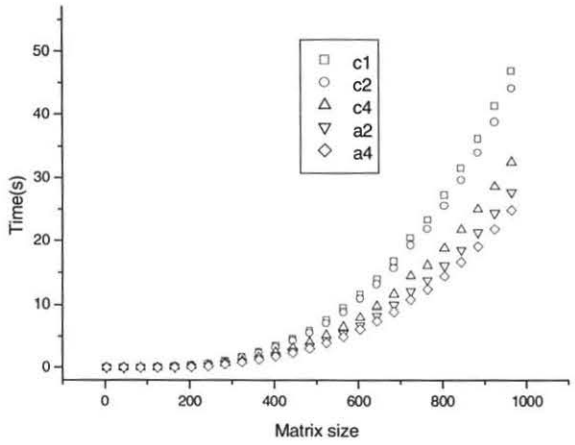

Fig. 4 - Execution times of the five routines on the K6 III system with external video card.

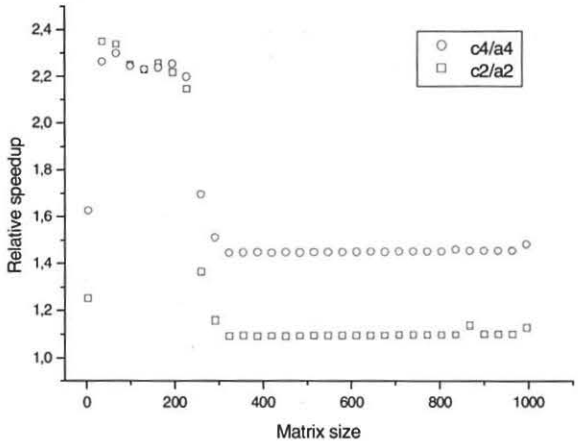

Fig 6 - Relative speedups of the SIMD versions with respect to the C versions on a K6III processor with onboard video enabled.

VI. CONCLUSIONS

We prove that with a little effort the use of single precision floating point vector operations can speed up significantly computational intensive matrix calculations. This paper reveals also that loop unroll techniques are very important to take profit of the new highly parallel multiscalar microcomputer architectures. Although the automatic use of these instructions by compilers does not appear to be straightforward, they are good candidates for manually optimized implementations of general linear algebra routines like BLAS. Recently Intel (with SSE2) has included double precision arithmetic vector support in the Pentium 4, turning the use of these instructions in scientific calculations even more interesting.

VII. ACKNOWLEDGMENTS

The authors would like to thank the financial support received from FAPESP under grant 98/14681-8 as well as the scholarship 99/05484-7 obtained from FAPESP by the first author.

VIII. REFERENCES

[ABE 99] ABEL, James et al. *Application Tuning for Streaming SIMD Extensions.* Intel Technology Journal Q2, 1999.

[AMD 99a] AMD White Paper. *Enhanced 3DNow!™ Technology for the AMD Athlon Processor.* AMD-52598A Advanced Micro Devices, Inc. October 4,199.

[AMD 99b] AMD Application Note. *3DNow!™ Instruction Porting Guide.* AMD Publication #2261, August 1999.

[AMD 99c] AMD Manual, *Extensions to the 3DNow!™ and MMX Instruction Sets.* AMD-22466B Advanced Micro Devices, Inc. August,1999.

[ASU 99] *Asus motherboard documentation.* http://asus.com/Products/Motherboard/

[BLA 01] BLAS - Basic Linear Algebra Subprograms http://www.netlib.org/blas/

[CYG 00] *The Cygwin toolpack.* http://sourceware.cygnus.com/cygwin/

[DIE 99] Diefendorff, Keith. *Pentium III = Pentium II + SSE Internet SSE Architecture Boosts Multimedia Performance.* Microprocessor Report. v.13, n.3, March 1999. p.6 – 11.

[INT 97] Intel, *Intel Architecture MMX™ Technology in Business Applications.* Intel Order Number 243367-002 June 1997.

[INT 99] Application Note. *Software Development Strategies For Streaming SIMD Extensions.* Intel AP-814 Order Number 243648-002 January 1999.

[MAC 99] Mackay, David; Chin, Steven. *Streaming SIMD Extensions and General Vector Operations.* ISV Performance Lab, Intel Corporation 1999, http://cedar.intel.com/software/idap/media/pdf/SIMD_VectorWP.pdf

[NAS 99] *The NASM conversion utility.* http://www.kernel.org/pub/software/devel/nasm/

[OPT 98] *The optimizer utility.* http://www.imada.ou.dk/~jews/optimizer/