Numerical applications and sub-word parallelism:
The NAS benchmarks on a Pentium 4

Daniel Etiemble
University of Toronto
10 King's college Road
Toronto, Ontario, Canada, M5S3G4
PN: 416 946 65784
Fax: 416 971 22 86
de@eecg.toronto.edu

Abstract: We examine the impact of Pentium 4 SIMD instructions on the Fortran and C versions of the NAS benchmarks, either by compiler vectorization or by assembly code in-lining. If few functions generally profit from the SIMD operations, the using complex numbers or random number generators can be efficiently accelerated.

Key words: SIMD extensions, Numerical applications

1. INTRODUCTION

With clusters of PCs or multiprocessor PCs, low cost parallel machines are now available. Most programming efforts concerns the parallelization considering the two different models (shared memory inside nodes and message-passing between nodes). But, to exploit any type of available parallelism, it is worthwhile to investigate the impact of SIMD parallelism that exists now in most of modern microprocessors on numerical applications.

The SIMD extensions to general purpose microprocessor instruction sets were introduced in 1995 to improve the performance of multimedia and DSP applications. The first extensions were able to process 64-bit integer vectors: Sun VIS [SUN95], HP MAX [LEE95], MIPS MDMX [KIL96] and Intel MMX [INT97]. The introduction of SIMD instructions that process single precision floating point vectors is more recent: AMD 3D Now [CAS98] and Intel SSE [INT99]. Motorola introduced Altivec in 1999 [APP99]. With the Pentium 4, Intel introduced SSE2 [INT-AP, INT-IR] which enables double precision floating point SIMD computation. If multimedia and DSP applications remain the major applications that can profit from SIMD extensions, the double precision extensions can also be used for numerical applications. In the rest of the paper, we will use "double" as an abbreviation for double-precision floating point number. The Pentium 4 XMM registers are 128-bit long and can pack two doubles. The SSE2 instructions provide a large set of floating point SIMD operations, either on 4 packed floats or 2 packed doubles. Equivalent scalar operations are provided to access or compute the lowest float or the lowest double of the XMM registers. As Pentium 4 microprocessors are suitable to build low-cost clusters of PCs, it is worthwhile to examine the impact of the SSE2 instructions on

the performance of numerical benchmarks or applications.

Vectorization is the terminology used by Intel for the compiler use of SIMD instructions.

Intel has published application notes for trivial examples as *saxpy* or *daxpy* functions [INT-AP]. Bik et al provide a high level overview of the parallelization and vectorization methods used in the C++/FORTRAN Intel compilers and give some performance results for dot products, LU-factorization and Linpack [BIK01a]. The same authors present other results with automatic vectorization for a Pentium 4 processor in [BIK01b]. The results concern the dot products, *saxpy* and *daxpy* and preliminary results on SPEC CPU2000. In this paper, we consider the NAS benchmarks [NAS-P], which are kernels or pseudo-applications more representative of numerical applications. They are widely used for testing parallel machines.

To evaluate the potential performance improvement of SSE2 numerical codes, we first measure the impact of compiler vectorization on the execution times of each benchmark, by enabling and disabling the vectorizer. Further optimization requires the use of assembly code for functions that are not vectorized. This leads to a significant methodology issue. Most of the numerical codes are FORTRAN codes and there is no simple method to insert assembly code within FORTRAN code. The only possibilities are by calling C functions containing inline assembly code from the FORTRAN code, or by linking an assembly object file with the FORTRAN object code. Neither solution can be used for our purpose: the overhead of function calls would forbid significant measures in the first case, and assembly code should be used at loop levels inside one function and not at a module or file level for the second case. To evaluate the impact of assembly coded optimizations, the only solution is to use a C version of the different benchmarks. This is why we have used the NAS benchmarks, for which the two versions are available, rather than the SPEC2000.

Vectorization issues are not exactly the same for C and FORTRAN programs. When using C or C++, the compiler has to worry about aliasing considerations. It cannot optimize a loop if it cannot ascertain whether the array references in a loop overlap. This problem doesn't exist for FORTRAN code. To check if aliasing issues significantly change the performance of the NAS benchmarks when using C versions, we have measured the execution times of the two versions for several optimization levels. We have compared the number of vectorized loops for each version of each benchmark and we have evaluated the impact of the vectorized loops on the overall execution times. Even if the number of vectorized loops is different for C and FORTRAN versions, the vectorization that is realized by the C and FORTRAN compilers has no significant impact on the overall performance of each version. This implies that the vectorized loops are not the time consuming loops. We will give some more details on this disappointing result. But it allows us to claim that the assembly coded optimization results that we present for the C version of the NAS benchmarks would be roughly the same for the FORTRAN version.

After this introduction, the second section presents the methodology that we use. The third section presents the execution times of the C and FORTRAN versions of the NAS benchmarks and gives detailed results on the compiler vectorization and vectorization efficiency. The fourth section details the most time consuming functions for each NAS benchmark. The fifth section presents the assembly coded optimizations and the corresponding performance improvement. Section 6 presents the related works. After the conclusion, the appendix gives details on some semantics issues when optimizing random generator functions.

2. METHODOLOGY

2.1 Measures

All the results correspond to measures on a Dell PC. The CPU was a 1.4 GHz Pentium 4 with 512 MB RDRAM running under Windows 2000 Professional. We have used the Microsoft Visual C++ environment with the 5.0 version of the Intel C/C++ compiler, the 5.0 version of the Intel FORTRAN compiler, and the Intel VTune profiler [INT-VT]. For each compiler, we have used the "maximize speed" (basically O2+QxW) and "high level optimization" (O3+QxW). The QxW option generates specialized code for the Pentium 4. Each tested program has been measured at least 5 times and we have taken the averaged value. All the measures have been done with only one running application (Visual C++). The results are provided either as execution time (sec) or as speed-ups, defined as old_execution time/new_execution time.

2.2 The NAS benchmarks

There is a large spectrum of high-end numerical applications and choosing some benchmark is always debatable. We have used the NAS benchmarks because they are one of the widely used and acknowledged benchmarks for which both FORTRAN and C versions are available. The FORTRAN version is the NPB2.3 serial version [NAS-P]. We have used the C OpenMP version of the NAS benchmarks [NAS-C] and removed all OpenMP directives to get a serial version. Although this version is probably not the best C serial code, it can be considered as a good reference for comparisons. We used the FP kernels (CG, FT and EP) and the FP pseudo applications (BT, LU and SP) with class A.

3. INITIAL PERFORMANCE OF THE FORTRAN AND C VERSIONS

Table 1 gives the execution times of the different benchmarks according to three different criteria: language (C versus FORTRAN), optimization level (O3 versus O2) and vectorization (with vectorization referred to as V and without vectorization referred to as NV). All the other options are similar: the compiler generates "Exclusively Streaming SSE2 extensions" (QxW option).

	BT	CG	EP	FT	LU	MG	SP
FORTRAN-O2-QxW-V	586.0	6.65	147.0	34.41	388.6	11.19	501.4
FORTRAN-O2-QaxW-V	595.5	6.50	148.1	33.83	394.1	11.57	506.8
FORTRAN O2-QxW-NV	592.5	6.55			387.8	12.31	494.9
FORTRAN O3-QxW-V	UNS	6.63	147.3	34.72	368.1	11.32	UNS
FORTRAN O3-QxW-NV	586.8	6.54			369.9	12.30	UNS
C-O2-QxW-V	618.7	7.24	179.4	30.64	410.0	11.62	481.7
C-O2-QaxW-V	619.8	7.00	154.5	29.66	410.5	11.61	478.0
C-O2-QxW-NV		6.60			409.8	11.63	489.4
C-O2-QaxW-NV		6.47			410.2	11.61	486.0
C-O3-QxW-V	643.0	7.28	179.3	30.50	412.4	10.11	481.9
C-O3-QxW-NV		6.60			412.9	11.62	489.5

Table 1: Execution time (sec) of the different benchmarks according to language, compiler, and vectorization options: O2 and O3 are the standard compiler options, QxW and QaxW are the specific P4 compiler options, V and NV stand for

vectorized and non-vectorized options, UNS corresponds to “unsuccessful” results and blank entries correspond to benchmarks without any vectorized loop for which V and NV execution times are identical.

Table 2 gives the number of vectorized loops when the vectorizer is on.

	BT	CG	EP	FT	LU	MG	SP
FORTRAN -O2	6	18	0	0	3	15	41
FORTRAN -O3	6	18	0	0	3	15	41
C-O2	0	13	0	0	6	1	44
C-O3	0	13	0	0	6	1	44

Table 2: Number of vectorized loops for FORTRAN and C versions.

In Table 1, we have covered most of the configurations according to main features of the C++ and FORTRAN Intel compilers:

a) Optimization levels: O2 and O3 are the standard compiler options and O2 and O3 names are used in the FORTRAN compiler. O2 corresponds to the Maximize Speed option in the C/C++ compiler and O3 to the CUSTOMIZE option. With the C compiler, we have used the ot, oa, og and oi sub-options that corresponds to “assume no-aliasing”, “intrinsic functions”, “favor fast code” and “global optimization”.
b) Code specialization: when compiling code for a Pentium 4 processor there are two different options. Either you compile using only the P4 specific code with SSE2 extensions (QxW option) or you compile two versions of the code including both generic IA32 code and P4 specific code (QaxW option) and the final choice is done at run-time. On a P4 machine, the two options should be equivalent, with a slight disadvantage for the QaxW option due to the overhead of the run-time choice. It turns out that the present implementations of the versions lead sometimes to opposite results. This is why we measured both.
c) Vectorization option: the compiler with QxW or QaxW options automatically tries to vectorize loops. The vectorization can be disabled, globally in the FORTRAN compiler and globally or at a loop by loop level in the C compiler. This option is useful to evaluate the efficiency of the compiler vectorization.

Figure 1 compares the execution time for the best versions of FORTRAN and C. BT, EP and LU execute faster with FORTRAN whereas FT, MG and SP are faster with C. CG has equivalent performance with the two languages. Except for FT, the difference is less or equal to 10%. The larger difference for FT probably originates from the FORTRAN implementation of double complex numbers.

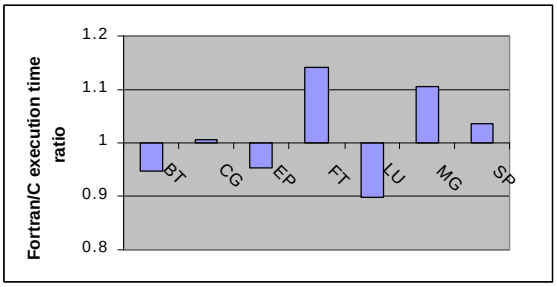


Figure 1: FORTRAN execution time/C execution time (the best version is used for each language). A ratio less than 1 means than FORTRAN executes faster.

Figure 2 shows the effect of the O3 option compared to O2. The O3 C version of BT is nearly 5% slower than O2 and the O3 FORTRAN version of LU is 5% faster than O2. Otherwise, there is no significant difference between the two options. Non-vectorized and vectorized ratios are quite similar. We have used the non-vectorized version to overcome one vectorization problem with BT (FORTRAN) for which the vectorized version is “unsuccessful” when the non-vectorized one is “successful”. For SP (FORTRAN), both vectorized and non-vectorized versions are unsuccessful.

Figure 3 examines the effect of “automatic dispatch” with QaxW option versus the “P4 only” QxW option. As previously mentioned, the results are opposite to our expectations for CG and FT (C and FORTRAN) and EP and SP (C). For EP, the difference is more than 10%. Otherwise, the differences are not significant. According to [ANS01], the two options don’t exactly use the same heuristics to determine

whether to do certain optimizations or not. One important case is the CMOV optimization. There are situations where QaxW will not perform CMOV optimization in certain routines where QxW normally would. CMOV's can be pretty expensive on a P4 due to their long latencies.

Figure 4 examines the compiler vectorization efficiency. Only the FORTRAN version of MG exhibits a spectacular effect, with a near 10% speedup with 15 vectorized loops. On the other hand, vectorizing the C version of CG induces a 10% slow-down. For all the other benchmarks where loops are vectorized, the impact of vectorization is less than 2%. Unfortunately, the impact is either positive or negative.

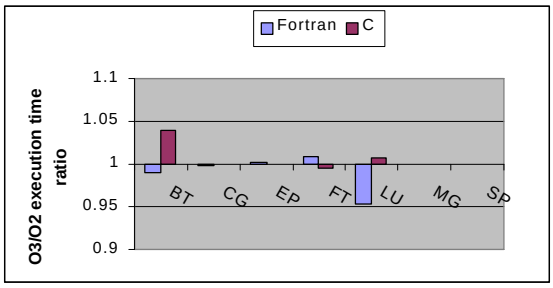


Figure 2: O3 execution time/O2 execution time with QxW and non-vectorized options. A ratio less than 1 means that O3 version executes faster. O3 version doesn't exist for the FORTRAN SP benchmark (unsuccessful result).

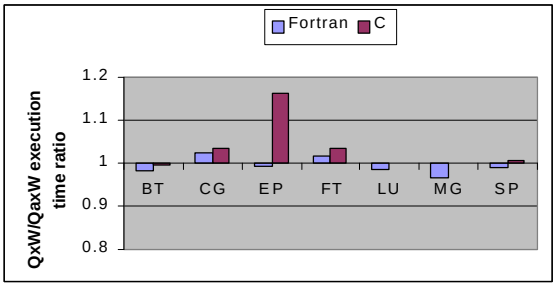


Figure 3: QxW execution time/QaxW execution time with O2 and vectorized options for FORTRAN and C. A ratio less than 1 means that QxW is more efficient.

Our examination of the FORTRAN and C versions of the NAS benchmark execution times according to the compiler options shows a different behavior for each benchmark. However, for most of the benchmarks and each language, the results of the O2 option, QxW and non-vectorized version are close to the best results. We will choose candidate benchmarks for assembly in-lining code after bottleneck detection. For these benchmarks, we will clarify the gap between O2+QxW+NV version and the best one.

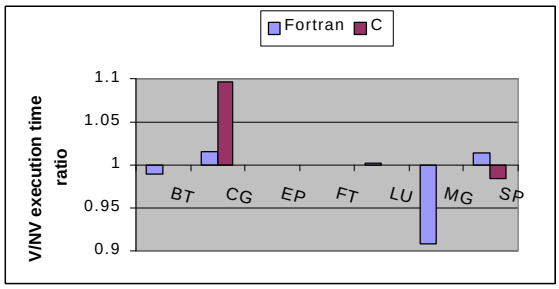


Figure 4: Vectorized execution time/non-vectorized execution time with O2 and QxW options. A ratio less than 1 means that the vectorization is efficient.

4. HOT SPOTS IN NAS BENCHMARKS

4.1 VTune profiling

The profiling has been done with the Intel VTune profiler in "released" mode, which means with O2+QxW optimized code, to outline both the most time consuming C functions and the specific C statements within these functions that use most clock cycles. The hot spots for the NAS benchmarks in class A are given in Table 3. The hot spots are evaluated as the percentage of used clock cycles for each benchmark.

The figures in Table 3 show two different situations. Some benchmarks (CG, EP and FT) have clearly one function that consumes most of the clock cycles: *conj_grad* (CG) is close to 95%, while *vranlc* (EP) and *fftz2* (FFT) represent more than 50 % of the clock cycle samples. On the other hand, the execution time for BT, LU, and SP is spread all over different functions. BT is a good example with six different functions that sums for 2/3 of the

overall samples, but no one exceeds 17% of the samples. MG exhibits an intermediate situation.

Program	FUNCTION	% OF SAMPLES
BT	mat_mul sub	16.7%
	binvcrhs	15.3%
	lhsy	13.7%
	compute_rhs	11.9%
	lhsx	11.1%
	lhsz	9.3%
CG	conj_grad	94.4 %
EP	vrancl	53.9 %
	main	46.1 %
FT	vranlc	35.9 %
	Compute_indexmap	21.6%
	fftz2	18.8 %
	cftts3	9.5 %
	cftts2	6.4%
	cftts1	2.8 %
LU	rhs	19.9%
	jacld	19.0%
	jacu	18.6%
	buts	18.6%
	blts	17.6%
MG	main	59.7%
	vranlc	39.7%
SP	compute_rhs	28.5%
	y_solve	15.2 %
	x_solve	13.2%
	z_solve	9.6%
	lhsx	9.5%
	lhsy	8.5%
	lhsz	7.3%

Table 3: Hot spot functions in NAS benchmarks (class A)

A deeper examination of the different functions reveals the functions themselves also exhibit the same behaviour: some functions (*vranlc*, *conj_grad*, *fftz2*) spend most of the time in a few C statements while others have the execution time spread all over the function. The four benchmarks that are candidate to assembly in-lining code are EP and MG (*vranlc*), CG (*conj_grad*) and FFT (*fftz2*). Two of them have C as the best version and only one has a best FORTRAN version. The O2+QaxW+NV version is the best one for CG, EP, and FT. The O3+QxW+V version is the best for MG[1]. These versions will be our reference.

BENCH MARK	BEST VERSION	O3/ O2	QXW/ QAXW	NV/V
CG	=	=	QaxW	NV
EP	FORTRAN	=	QaxW	=
FT	C	=	QaxW	=
MG	C	O3		V

Table 4: Behavior of the C candidate benchmarks to manual coding according to language and compiler options. Bolded entries outline sharp differences.

5. PERFORMANCE SPEED-UP WITH ASSEMBLY CODING

We consider now the further steps in loop vectorization that can be done by using assembly code. As previously explained, this hand coding is done with the C version of the NAS benchmarks. According to previous results, we only use assembly coding for CG, EP and FT. The other applications could be improved also, but the difficulty appears much greater. Examination of the hot spots shows little room for “SIMDing” the code.

NAS benchmarks have some attractive features for testing and debugging the modified versions with some assembly code. First, all the benchmarks measure the execution time. Second, all benchmarks test the numerical results. Correct code delivers a “successful” result whereas incorrect code delivers an “unsuccessful” one. All the “modified” versions which we have measured deliver “successful” results: the results are the same as for the original C version of the program. There is only one exception that we describe below.

Several benchmarks use a pseudo-random generator (*vranlc* function). EP (Embarrassingly

[1] *Figure 2* that compares O3 and O2 execution times with non-vectorized loops shows no advantage for the C O3 version of MG. However, MG is the only case where O3 gives a significant advantage to the vectorized version versus the O2 version. This is why we use O3+QxW+V version as the reference one for MG.

Parallel) just generates random numbers. Some benchmarks use the generator for Monte Carlo simulations, while others use it to initialize variables. The optimization of *vranlc* would benefit several benchmarks. SSE2 instructions can be used to generate simultaneously two different sequences of "random" numbers, one in the lower double and the other on the upper double of an XMM register. The other option is to generate a single sequence, with the numbers of even rank in the lower double and of odd rank in the upper double. In the first case, we generate two times more numbers in the same time compared to the original case, but the function is now different as it should be initialized with two different values and the function call is also modified. In the second case, we generate a same length sequence using half of the original time. This second approach does not change the function call. In this case, for reasons that we detail in the appendix, the resulting sequence is different from the sequence that is used in the NAS benchmarks. Since the "successful" results of the benchmarks using *vranlc* correspond to the specific original sequence used in the NAS benchmarks, changing the vranlc functions leads to "unsuccessful" results. This is the only case for which the execution times of "unsuccessful" results are presented. We took care to check that all the "modified" codes were "successful" with C version of *vranlc* before modifying the *vranlc* function.

5.1 Optimization of random generators (Vranlc)

Figure 5 shows the code for *vranlc* loop body. The critical issue is the *double* to *integer* conversions. This conversion implies rounding *towards zero* instead of *rounding to nearest* that is the commonly used rounding mode. A non optimized x86 code changes the rounding mode, which needs changing of the processor control word. The corresponding instruction serializes, which leads to a significant performance degradation. The 5.0 version of the C compiler with QxW option combines x87 code with `cvttsd2` instruction, which is one of the "double to integer with truncation" conversion instructions that have been introduced with SSE2 instructions. The execution time is referred as to the C version of Table 1.

```
t1 = r23 * x;
x1 = (int)t1;
x2 = x - t23 * x1;
t1 = a1 * x2 + a2 * x1;
t2 = (int)(r23 * t1);
z = t1 - t23 * t2;
t3 = t23 * z + a2 * x2;
t4 = (int)(r46 * t3);
x = t3 - t46 * t4;
y[i] = r46 * x;
```

Figure 5: loop body of *vranlc*

A first level of optimization consists of coding the loop body with the scalar version of the SSE2 instructions. This version of *vranlc* (referred to as S1) only uses the lower "double" of the XMM registers, using them as 64-bit registers instead of using the floating point stack of the IA-32 architecture. A second level of optimization consists of using packed doubles to generate two random numbers per iteration. We only consider the version (called S2) that is compatible with the original function call (one single sequence). It needs a prologue to generate the second number from the initial value before using packed instructions and an epilogue to generate the last random number when the sequence length is even.

Table 5 presents the execution times for generating an array of 8192 random numbers and the corresponding execution time per iteration. The C version and the modified versions have been compiled with the O2 + QaxW options. The SSE2 scalar version is 1.77 times more efficient than the C version. Two factors explain this result: the latencies of the SSE2 instructions are slightly less than the latencies of the corresponding x87 instructions and communication between the FP stack and the XMM register banks probably introduce some overhead. The SSE2 packed version (S2) is 1.75 times more efficient than the scalar version (S1). The overhead for computing the

first and the last numbers prevents a perfect speed-up.

	EXECUTION TIME	TIME PER ITERATION	Speed-up
C	1.55 s	190 ns	
S1	0.88 s	107 ns	1.77
S2	0.50 s	61 ns	3.11

Table 5: Execution time of the *vranlc* function (8192 random numbers)

5.2 Optimization of EP

As most EP execution time is spent in the *vranlc* function, we only evaluate EP execution times for the different versions of *vranlc*. Compared to the C version of *vranlc* (154.5 s), the speed-up is 1.20 with the S1 version (128.9 s) and 1.54 with the S2 version (100.4 s).

5.3 Optimization of MG

For MG, we also only consider the impact of *vranlc* function. The best C, S1, and S2 version execution times (O3+QxW+V) are respectively 10.12 s, 10.11 s, and 10.11 s. The speedup is negligible.

5.4 Optimization of CG

The hot spot for CG corresponds to the code presented in Figure 6.

```
for (j = 1; j <= lastrow-firstrow+1; j++) {
        sum = 0.0;
      for (k = rowstr[j]; k < rowstr[j+1]; k++) {
       sum = sum + a[k]*p[colidx[k]];}
        w[j] = sum;}
```

Figure 6: Critical code for conjugate gradient

The inner loop looks like a dot product, but the elements of the second array are accessed indirectly, which prevents the direct vectorization of the loop. We tested two levels of optimizations. The first level uses SSE2 instructions to compute the dot product, except that the lower double and the higher double of `p[colidx[k]]` and `p[colidx[ k+1]]` are loaded sequentially. This version is called S2. The second level of optimization unrolled the S2 version by a factor of 2. It is called U2S2.

Another issue with code presented in Figure 6 is the loop index boundaries. The first and last elements of the inner loop are unknown at compile time. For efficient memory accesses, the SSE2 move instructions between XMM registers and 16-byte memory words should be aligned, which means specific conditions on the addresses of the first and last doubles that use XMM registers. For both versions, trivial C prologue and epilogue have been used to insure that the iterations that use assembly code with SSE2 instructions have correctly aligned addresses.

For CG class A, the different versions of *vranlc* have no significant impact on the performance. We only present the results when using the original C version of *vranlc*. The execution times for the C, S2 and U2S2 versions are respectively 6.47 s, 6.71 s and 6.58 s. Assembly coding slows down, with a “speed-up” of 0.96 (S2) and 0.98 (U2S2).

5.5 Optimization of FT

The main feature of the FT benchmark is that it uses complex numbers that are declared as a structure with a `double` real part and a `double` imaginary part. As this structure is laid out in memory as two consecutive 64-bit words, it is suitable for SSE2 memory accesses (two doubles to and from XMM registers) and SSE2 computations.

We have optimized several functions of the FT benchmark which constitute hot spots. Code of the *fftz2* function is shown in Figure 7. The corresponding loops are typical of FFT computation. Figure 8 shows one limited part of *cffts1* code: it basically transfers data between two matrices. Only the two inner loops have been implemented with assembly code. The remaining part of the code that has been optimized is similar. The code for *cffts2* and *cffts3* is similar.

```
for (k = 0; k < lk; k++) {
      for (j = 0; j < ny; j++) {
       double x11real, x11imag;
       double x21real, x21imag;
       x11real = x[i11+k][j].real;
       x11imag = x[i11+k][j].imag;
       x21real = x[i12+k][j].real;
       x21imag = x[i12+k][j].imag;
       y[i21+k][j].real = x11real + x21real;
       y[i21+k][j].imag = x11imag + x21imag;
       y[i22+k][j].real = u1.real * (x11real –x21real)
          - u1.imag * (x11imag - x21imag);
       y[i22+k][j].imag = u1.real * (x11imag –
            x21imag)
          + u1.imag * (x11real - x21real);}}
```

Figure 7: fftz2 function to optimize

```
for (k = 0; k < d[2]; k++) {
    for (jj = 0; jj <= d[1] - fftblock; jj+=fftblock) {
    for (j = 0; j < fftblock; j++) {
        for (i = 0; i < d[0]; i++) {
            y0[i][j].real = x[k][j+jj][i].real;
            y0[i][j].imag = x[k][j+jj][i].imag;}} }}
```

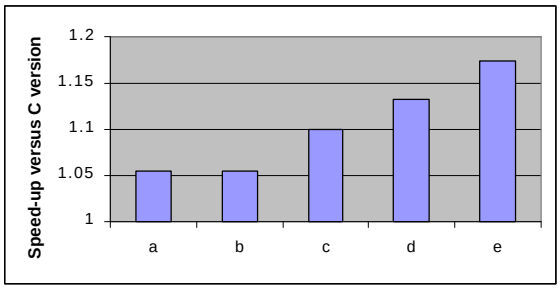
Figure 8: part of cffts1 code to optimize

	C VERSION	29.65 s
a	fftz2	28.09 s
b	cfftz + cffts1, cffts2, cffts3	28.09 s
c	fftz2 + cfftz + cffts1, cffts2, cffts3	26.94 s
d	fftz2 + cfftz + cffts1, cffts2, cffts3 + vranlcS1	26.17 s
e	fftz2 + cfftz + cffts1, cffts2, cffts3 + vranlcS2	25.26 s

Table 6: Execution time of the different C versions of FT (O2+QaxW+NV) according to assembly coded functions

We have measured several versions of the program. The original version corresponds to C code. The second version uses SSE2 instructions for the fftz2 function. The third version uses SSE2 instructions for cffts1, cffts2, cffts 3 and cfftz functions. The fourth version combines the second and third versions. The fifth version corresponds to version four together with the replacement of the C version of vranlc by the S1 version. Finally the last version is version four with the S2 vranlc function. Table 6 gives the execution times and Figure 9 shows the impact of the successive optimizations. The overall speed-up is 1.17.

Figure 9: Speed-up from the C version according to the

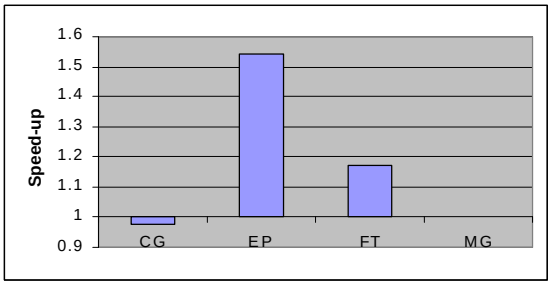


different assembly coded functions indicated in *Table 6*

5.6 Overall results

We summarize the overall results in Figure 10. As already mentioned, there is no speed-up for BT, LU, MG, and SP. Only EP, which is a very specific application, exhibits a significant speed-up when using SSE2 instructions. It mainly comes from the random generator, for which SSE2 instructions double the number of generated numbers and better perform floating point to integer conversions with truncation. The only benchmark that actually profits from the SIMD operations on two doubles is FT, for which real and imaginary parts fit in the 16-byte SIMD operation width. For FT, the overall speed-up is 1.17 when considering the new implementation of the random generator, but is limited to 1.09 when only considering the SIMD approach.

Figure 10: Speed-up for the best version with assembly



code versus the best C version for the 4 considered NAS benchmarks.

5.7 Some more remarks

Some parts of code allow SIMD operations. Figure 11 shows a part of EP code that can use SSE2 instructions. Vectorization cannot be applied to the whole loop. However, the following sets of statements can operate in SIMD mode: (X1, X2, T1), (T3, T4, L) and (SX, SY). We have compared the overall execution time of the program with the ASM SSE2 version of Figure 11 with the execution time of the original C code: the speed-up is 0.98!

```
for (i = 0; i < NK; i++) {
        x1 = 2.0 * x[2*i] - 1.0;
        x2 = 2.0 * x[2*i+1] - 1.0;
        t1 = pow2(x1) + pow2(x2);
        if (t1 <= 1.0) {
      t2 = sqrt(-2.0 * log(t1) / t1);     /non simd
      t3 = (x1 * t2);                /* Xi */
      t4 = (x2 * t2);                /* Yi */
      l = max(fabs(t3), fabs(t4));
      qq[l] += 1.0;              /* counts */
      sx = sx + t3;              /* sum of Xi */
      sy = sy + t4;              /* sum of Yi */
        }
   }
```

Figure 11: Extract from EP code

A systematic use of SIMD operations can be counter-effective for floating point computations. There is one fundamental reason. There are no significant differences in the Pentium 4 latencies between the SSE2 double precision operations and the corresponding IA-32 x87 ones: 4/5 for ADD, 6/7 for MUL with the same throughput: 2. When the code only consists of chains of instructions with true data dependencies (as in the code shown in Figure 11), SSE2 instructions cannot exhibit a perfect speed-up of 2. The two x87 instructions needed for each packed SSE2 instruction are pipelined: we have two chains of x87 instructions with the same data-dependencies that the SSE2 instructions have. Compared to the SSE2 chain, the second x87chain is delayed by the sum of latency differences between the two types of instructions plus the pipeline delay between each chain. This is far less than the data dependency delay of the SSE2 instructions. “Reduction” operations are another issue. In the present implementation of SSE2 extensions, there are no instructions to operate on different parts of an XMM register. These operations, corresponding to the reduction in parallel programming, need several data dependent SSE2 instructions and the corresponding overhead could be greater than with x87 instructions. Finally, unaligned packed memory accesses are costly compared to aligned ones. Unfortunately, it is often impossible to guarantee aligned accesses in actual programs.

6. RELATED WORKS

This section discusses some related works in SIMD instruction set evaluation. Most of the published results consider multimedia benchmarks using integers or single precision floating point data. Most of them use simulated architectures. In [YAN98], Yang and al studied the impact of paired SIMD single precision floating point instructions and 4-way SIMD floating point instructions on 3D geometry transformations. In [RAN99], Ranganathan and al studied the performance of several integer image processing kernels by simulation. In [NGU99], Nguyen and al considered a small number of micro kernels with AltiVec extensions. In [BEC00], Bechennec and al presented preliminary results about AltiVec performance on floating point multimedia kernels. The results have been extended in [SEB01], which focuses on the impact of memory hierarchy (latency and bandwidth) on 9 integer and floating point micro kernel using AltiVec. As mentioned, all the results presented in these papers are based on simulations.

In [BHA98], Bhagarva and al measured the execution times of their benchmarks on a Pentium processor by using the same methodology as we have used in this paper: Visual C++ compiler, VTune profiler, etc. They have considered DSP kernels and JPEG. When the research was done with MMX technology, only the integer SIMD extension was available. In this paper, we consider the impact of SIMD

double precision floating point instructions of Intel SSE2 extensions on the performance of numerical benchmarks. Bik and al [BIK01a and BIK01b] presented Intel results on some kernels (dot products, saxpy/daxpy, LU factorization), Linpack and SPEC CPU2000. Our results are far less spectacular.

7. CONCLUSION

We have shown that the Intel C and FORTRAN compilers at present do not efficiently use the SSE2 instructions for the floating point code of the NAS benchmarks. On most benchmarks, they only "vectorize" loops that do not significantly impact the overall execution time. By considering the time consuming functions in every NAS benchmark, we have observed that very few parts of code are candidates for an efficient use of SIMD operations. Code using complex numbers can be slightly accelerated by SIMD operations. For the FT benchmark class A, we measured a 10% speed-up. All the code that uses Monte Carlo methods can benefit from improved random generators. On EP, which generates random numbers, we have obtained a 60% speed-up by generating two times faster the sequence of random numbers. The speed up obviously depends on the benchmarks. Even with significant improvement in "SIMD technology" for compilers, it would appear that the speed-up that can be expected from SIMD floating point operations on numerical applications is limited. Beyond the classical vectorization issues, the main reasons are 1) the large latencies of floating point operations that reduce the impact of packed instructions versus scalar ones in cases of data dependent instructions and 2) some weaknesses in the currently available SSE2 extensions (cost of unaligned accesses, lack of intra-register operations).

8. APPENDIX: VRANLC FUNCTIONS

The *vranlc* routine generates N uniform pseudo-random double precision numbers in the range (0,1) by using the linear congruential generator $x[k+1] = a^* x[k] \pmod{2^{46}}$ where a and x[k] ranges are $(0, 2^{46})$.

According to the previous formula, $x[k+2] = a^{2*} x[k] \pmod{2^{46}}$ can be used to generate the sequence of every other number from an initial value. Starting from x[0], after generating x[1] with the initial formula, the second formula allows us to simultaneously generate at each iteration x[2p] and x[2p+1] for p between 0 and N/2-1. The choice of a and of x[0] is important to get a good sequence of random numbers. The *vranlc* function of the NAS benchmarks uses the largest odd 32-bit integer value. Obviously, we cannot use the same value as $\boldsymbol{a^2}$ cannot fit into a 32-bit integer. Thus the same sequence cannot be obtained, which explains why NAS benchmarks with a "modified" sequence deliver an "unsuccessful" result. However, a careful choice of a value of $\boldsymbol{a}$ less than the square root of the largest odd 32-bit integer value should lead to a good sequence of random numbers. Choosing this value of $\boldsymbol{a}$ is out of the scope of this paper. We just want to show this method which can significantly speedup the generation of random numbers does not change the function interface.

9. REFERENCES

[APP99] AltiVec Home Page, http://developper.apple.com/hardware/altivec, May 1999

[ANS01] A. Ansari, Intel Microcomputer Software Laboratories, Private communication.

[BEC00] J-L Bechennec, J. Sebot and N. Drach-Temam, "A Performance Evaluation of Multimedia Kernels using AltiVec Streaming SIMD Extensions", in Third Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW00) at HPCA00, January 2000

[BHA98] R. Bhargava, L John, B. Evans and R. Radhakrishan, "Evaluating MMX Technology using

DSP and Multimedia Applications", in Micro 31, 1998

[BIK01b] A.J.C. Bik, M. Birkar, P.M. Grey and X.Tian, "Experiments with Automatic Vectorization for the Pentium(R) 4 Processor", 9th Workshop on Compilers for Parallel Computers, June 27-29, Edinburgh, Scotland, UK, 2001

[BIK01a] A.J.C. Bik, M. Birkar, P.M. Grey and X.Tian, " Efficient Exploitation of Parallelism on Pentium(R) III and Pentium(R) 4 Processor-Based Systems." Intel Technology Journal, February, 2001

[CAS98] Brian Case, "3dnow boosts non-Intel 3d performance", Microprocessor Report, 12(7), June 1998

[INT97] Intel, "MMX technology architecture overview". Intel Technology Journal, September 1997

[INT99] Intel, "Streaming SIMD extensions", Intel Technology Journal, January 1999

[INT-AP] Intel, Using Pentium 4 Processor SSE2 instructions for SQXPY/DAXPY, AP-935.

[INT-BA] Intel, IA-32 Intel Architecture Software Developer's Manual, Volume 1, Basic Architecture, http://developer.intel.com/design/pentium4/manuals/245470.htm

[INT-IR] Intel, IA-32 Intel Architecture Software Developer's Manual, Volume 2, Instruction Set Reference, http://developer.intel.com/design/pentium4/manuals/245471.htm

[INT-VT] Intel, Intel® VTune™ Performance Analyzer 5, http://support.intel.com/support/performancetools/vtune/v5/relnotes.htm

[KIL96] E. Killian, "Mips extensions for digital media with 3". Microprocessor Forum, October 1996

[LEE95] R. Lee ,J. Huck, "64 bits and multimedia extensions in the PA-RISC 2.0 Architecture", http://www.hp.com/ahp/framed/technology/micropro/architecture/docs/pa2go3.html, 1996

[NAS-C] OpenMP C version of NPB2.3, http://pdplab.trc.rwcp.or.jp/Omni/benchmarks/NPB/

[NAS-P] NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB

[NGU99] H. Nguyen and L.K. John, "Exploiting SIMD Parallelism in DSP and Multimedia algorithms using AltiVec technology", in International Conference on Supercomputing, ISC99, 1999.

[RAN99] P. Ranganathan, S. Adve and N. Jouppi, "Performance of Image Processing with General Purpose Microprocessors and Media ISA Extensions", in International Symposium on Computer Architecture, ISCA26, May 1999.

[SEB01] J. Sebot and N. Drach-Temam, "Memory Bandwidth: The true Bottleneck of SIMD Multimedia Performance on a Superscalar Processor", Europar 2001, August 2001.

[SUN95] Sun Microelectronics, "The VIS Instruction Set", http://www.sun.com/microelectronics/vis/, 1995.

[YAN98] C. L. Yang, B. Sano and A. R. Lebeck, "Exploiting Instruction Level Parallelism in Geometry Processing for three dimensional graphics applications", in Micro 31, November 1998