An Empirical Study of Some x86 SIMD Integer Extensions

Isabelle Hurbain and Georges-André Silber

Centre de recherche en informatique, Mines de Paris, Fontainebleau, France,
`Isabelle.Hurbain@ensmp.fr`

Abstract. We benchmark 32 SIMD instructions of the x86 ISA that operate on integers to determine the performance gain that can be obtained for each of them. We show that these instructions can dramatically improve the speed of programs and give for each of them a reference speedup. Using these results, we check whether the state-of-the-art C compilers GCC 4.1 and ICC 9.0 are capable of automatically detect simple cases where those instructions can be used.

1 Introduction

Like several other modern processors, Intel's Pentium 4 processor provides SIMD extensions called *Streaming SIMD Extensions* (SSE), defining a set of 8 named 128-bit wide vector registers, called XMM registers, and a set of component-wise vector operations on these registers. One typical usage of these extensions is to add two vectors of 16 bytes to obtain another vector of 16 bytes. Several papers [2, 4, 3] report a significant speedup when using SSE instructions instead of general-purpose code. However, the performance of SSE instructions compared to the performance of the same operation coded with "regular" operations is not well known. Furthermore, the automatic exploitation of those instructions by compilers is difficult to verify without looking at the assembly code they produce.

In this paper, we systematically benchmark the 32 SIMD instructions that operate on integers to determine the performance gain that can be obtained for each of these extensions. We show that these extensions can dramatically improve the speed of programs and give for each of them a reference speedup. Using these results, we check whether the state-of-the-art C compilers GCC 4.1 and ICC 9.0 are capable of automatically detect simple cases where those instructions can be used. This check is done by looking at the performance of the generated code compared to a hand-coded assembly reference code and by looking at the assembly code generated by the compilers.

2 Overview of SSE instructions

What we are calling SSE in this paper is the Intel SIMD vector architecture that was deployed over time as a series of four vector extensions to the x86 Instruction

Set Architecture (ISA). The first was MMX, followed by SSE, SSE2, and SSE3. Each builds on the extension that went before it. There are three major classes of data on the SSE vector unit: integer, single precision floating point and double precision floating point vectors, each of which may be serviced by separate parts of the processor. In this paper, we only describe and study integer operations.

MMX, the first of the vector extensions provides a series of packed integer operators that utilize eight 64-bit registers. The operations defined by MMX are, generally speaking, also available in a 128-bit format in SSE2. MMX is sometimes used as a source of additional register storage area. However, since the vector arithmetic and logical unit (ALU) is shared with SSE2, there is likely no throughput advantage to using the two in parallel. SSE and SSE2 adds a series of packed and scalar single precision floating point operations, and some conversions between single precision and integer. SSE2 is enabled by default on GCC 4. In addition, SSE2 replicates most of the integer operations in MMX, except modified appropriately to fit the 128-bit XMM register size. In addition, SSE2 adds a large number of data type conversion instructions.

There is a C Programming Interface for SSE: the SIMD vector register is described in C as a special 128 bit data type (`__m128i`) and a series of function-like intrinsics are used to do SIMD style operations on those variables. In this paper, we restrict ourselves to the study of 32 integer instructions. The x86 ISA defines four sizes of integer data types: bytes, words (16-bits), double words (32-bits) and quad words (64-bits). Those four types are represented by the letters `b`, `w`, `d`, and `q` in the name of the operations: `paddq` is a parallel addition on 2 quad words, i.e. two 64-bits integers. Each operation has 3 parameters: two input vectors and one output vector. The integer instructions we studied can be divided in four families: the simple arithmetic component-wise operations, the arithmetic component-wise operations with saturation, the component-wise comparison operations, and the "miscellaneous" operations.

The simple arithmetic component-wise operations are the addition (`paddb`, `paddw`, `paddd`, `paddq`), substraction (`psubb`, `psubw`, `psubd`, `psubq`), multiplication (`pmulhw`), and unsigned multiplication operations (`pmulhuw`). The operations are available for all the cases of packing in the 128-bit registers. The overflows are managed by a wrap-around of the values.

The arithmetic component-wise operations with saturation comprise signed operations (`paddsb`, `paddsw`, `psubsb`, `psubsw`) and unsigned operations (`paddusb`, `paddusw`, `psubusb`, `psubusw`). The overflows are managed by a saturation of the value.

The component-wise comparison operations take two vectors and produce a third vector composed of values representing the results of the comparisons (`pcmpeqb`, `pcmpeqw`, `pcmpeqd`, `pcmpgtb`, `pcmpgtw`, `pcmpgtd`, `pmaxub`, `pmaxsw`, `pminub`, `pminsw`).

The "miscellaneous" operations are the component-wise average of two vectors (`pavgb`, `pavgw`), the component-wise absolute values of the difference of 16 bytes followed by the sum of those differences in 2 unsigned word integers (`psadbw`), and the component-wise multiply and add operations (`pmaddwd`).

3 Benchmarking SIMD integer instructions

The first step of our work was to design a set of basic SSE benchmarks. Their principle is simple: for each of the 32 SSE integer instructions, we wrote a sequence of "regular" C instructions doing the same operation and a C program that uses the intrinsic implementing the SSE operation. For instance, considering the operation `paddb` that adds two vectors of 16 bytes, we wrote the following C code

```
void test_loop_c (char a[16], char b[16], char c[16])
{
  int i;
  for (i = 0; i < 16; i++)
    {
      c[i] = a[i] + b[i];
    }
}
```

and this other one that uses the intrinsic `__mm_add_epi8`:

```
#include <xmmintrin.h>
void vector_add (__m128i * a, __m128i * b, __m128i * c)
{
  *c = _mm_add_epi8 (*a, *b);
}
```

The intrinsics help the compiler to generate a good assembly code which, in our case, would be:

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
movdqa  (%eax), %xmm0
movl    12(%ebp), %eax
paddb   (%eax), %xmm0
movl    16(%ebp), %eax
movdqa  %xmm0, (%eax)
popl    %ebp
ret
```

The second step of our work was to execute those two programs for each SIMD integer instruction. The C programs with the intrinsics are used as a reference. For our tests, we use two different C compilers: GCC 4.1.0[1] and ICC 9.0[2]. All executions have been done on a computer with a 3 GHz Intel Pentium

[1] `gcc (GCC) 4.1.0 20051015 (experimental)`
[2] `icc 9.0 Build 20050430`

4 and 2 Go of RAM. Each program has been executed 10 times and inside each program, the "regular" and the intrinsic functions are called several million times to ensure reliable results.

The third step of our work was to determine whether the selected compilers are capable of automatically vectorize the "regular" C codes and to generate code with SSE instructions. In fact, we have to help the compilers: the C loops need some code transformations so that SIMD code can be generated. As a matter of fact, the code is not vectorized if the data are not explicitly declared as `restrict`[6].

4 Results of the benchmarks

In this section, we will give the simplified results of all our benchmarks and detail two cases: one where the vectorization is detected and achieved, and one where it is not. The complete report for the benchmarks is also available in [5].

Figure 1 gives the results of all the benchmarks. For GCC, base times are expressed in seconds and have been obtained by running a code compiled with the options `-O2 -fno-strict-aliasing`. For ICC, base times have been obtained by running a code compiled with the options `-O2`. The autovect times have been obtained with the options

```
-O2 -fno-strict-aliasing -msse2 -ftree-vectorize
```

for GCC, and with the options

```
-march=pentium4 -axN -nolib\_inline -ip\_no\_inlining
-O2 -vec\_report2
```

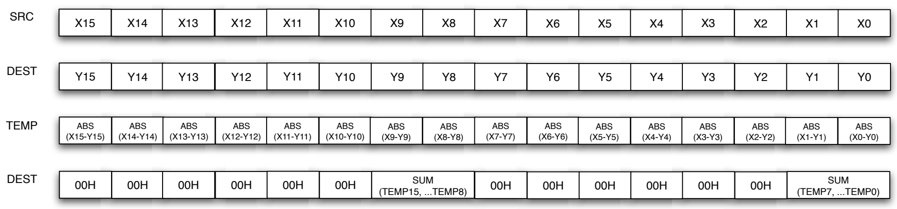
for ICC. The intrinsics columns give the execution time for the C code with intrinsics compiled with GCC and ICC. The **speedup** columns are obtained by dividing the *base* times with autovect and intrinsics times for GCC and ICC. A sign ✔ in the **status** columns indicates that the compiler uses SSE instructions and the sign ✘ indicates that it does not.

A "good" case: PADDB. For the benchmark of the instruction `paddb`, ICC is able to generate an assembly code that is very similar to the one generated by the intrinsics. On the contrary, GCC, while being able to detect a vectorization and generating a `paddb` instruction, generates a much more cumbersome and a much longer assembly code (more than 10 times longer) by partly unrolling the loop before vectorizing it. It is worth noting that, while it is possible to force the alignment of the data with GCC (using the `((aligned))` attribute), it is to our knowledge impossible to tell the compiler the data are actually aligned. This would be useful when the data is defined in a function and used in another one.

When using unaligned data, ICC is able to generate a correct SIMD code. The loads and store are unaligned (`movdqu` assembly instruction) and are less efficient than aligned memory accesses. The impact on the speedup compared to C code without vectorization stays limited.

For this case, the two compilers are able to detect and apply the vectorization. However, as we said before, the vectorization achieved by GCC is clearly not very good – and we can see that in the performances of vectorized code. On the contrary, ICC's performances for vectorized code and for intrinsics coding are very similar. Moreover, we can also see that GCC's assembly code generated for intrinsics seems to be more efficient than the one generated by ICC. We can observe this in all our benchmarks.

A case that is not vectorized: PSADBW. Let us now consider a more complex assembly instruction, `psadbw`. Its operating mode is described in the following schema:


SRC
X15 X14 X13 X12 X11 X10 X9 X8 X7 X6 X5 X4 X3 X2 X1 X0
DEST
Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
TEMP
ABS (X15-Y15) ABS (X14-Y14) ABS (X13-Y13) ABS (X12-Y12) ABS (X11-Y11) ABS (X10-Y10) ABS (X9-Y9) ABS (X8-Y8) ABS (X7-Y7) ABS (X6-Y6) ABS (X5-Y5) ABS (X4-Y4) ABS (X3-Y3) ABS (X2-Y2) ABS (X1-Y1) ABS (X0-Y0)
DEST
00H 00H 00H 00H 00H 00H SUM (TEMP15, ...TEMP8) 00H 00H 00H 00H 00H 00H SUM (TEMP7, ...TEMP0)

The corresponding C code is:

```
void test_loop_c(char a[16], char b[16], short int c[8]) {
  int i;
  unsigned char tmparray[16];
  for(i=0; i<4; i++) {
    c[i] = 0;
    c[i+4] = 0;
  }
  for(i=0; i<16; i++) {
    tmparray[i] = (abs(a[i] - b[i]));
  }
  for(i=0; i<8; i++) {
    c[0] += tmparray[i];
    c[4] += tmparray[8+i];
  }
}
```

This code is *not* detected as vectorizable by either compiler. This may have several reasons:

- the pattern is composed of three loops that are not necessarily easily vectorizable by themselves;
- the type of the input and output data is not homogeneous, which can lead to recognition problems;

– the operation is mainly a reduction, an operation usually not well handled by compilers;
– as the operation is complex, the pattern may not be recognized “as is” but another equivalent one may be. This is especially hard to know for “black-box” compilers such as ICC.

As we can see, when we ask for vectorization to the compiler, the results are equivalent to the “regular” code – but the intrinsics code greatly improves performance. It is worth noting that this algorithm is extensively used in multimedia applications (video encoding in particular).

5 Conclusion

Looking at our results, it is clear that the SSE instructions can dramatically improve the performance of programs that operate on dense array of integers. For instance, the instruction `psadbw` gives a speedup of 21.5 over the “regular” C code.

For simple vectorization, ICC is a clear winner, even if it needs some help (with `restrict` informations). Both compilers fail on more tricky cases, but it is easier to understand why with an open source software like GCC and it gives more potential for improvement from the research community than a “black-box” software like ICC.

For all 32 SIMD integer extensions, GCC generates better assembly code from the SSE intrinsics. Our study shows than the best way to obtain good performance is still to use directly the SSE intrinsics. It is an interesting fact for the research community on parallel compilers because it shows that it is possible to achieve very good performance with source to source transformations.

References

1. Intel: IA-32 Intel Architecture Software Developer’s Manual (2004)
2. Juyup Lee, Sungkun Moon, Wonyong Sung: H.264 Decoder Optimization Exploiting SIMD Instructions - IEEE Asia-Pacific Conf. on Circuits And Systems, Dec. 2004, Vol 2., pp. 1149-1152
3. Xiaosong Zhou, Eric Q. Li, Yen-Kuang Chen: Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions - SPIE Conf. on Image and Video Communications and Processing, Jan. 2003, Vol. 5022, pp. 224-235
4. Gang Ren, Peng Wu, David Padua: An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions - IEEE International Parallel and Distributed Processing Symposium, 2005
5. Isabelle Hurbain: An Evaluation of the Automatic Generation of Parallel x86 SIMD Instructions by GCC and ICC - Technical report, 2005. Available at `http://www.cri.ensmp.fr/classement/2005.html`.
6. Douglas Walls: How to Use the restrict Qualifier in C - Sun Developer Network, Sun Microsystems, July 2003. Available at `http://developers.sun.com/prodtech/cc/articles/cc_restrict.html`.

OPERATION	GCC 4.1.0						ICC 9.0					
	base (s)	autovect (s)	speedup	status	intrinsics	speedup	base (s)	autovect (s)	speedup	status	intrinsics	speedup
`paddb`	6.79	4.16	1.63	✔	**1.08**	6.26	8.09	2.33	3.48	✔	2.29	3.53
`paddw`	5.70	3.48	1.64	✔	**0.92**	6.18	3.86	1.80	2.14	✔	1.81	2.13
`paddd`	3.05	4.99	0.61	✔	**1.00**	3.04	2.39	1.80	1.32	✔	1.72	1.39
`paddq`	3.10	4.04	0.77	✔	**1.08**	2.88	3.50	1.73	2.03	✔	1.80	1.95
`psubb`	7.25	3.83	1.89	✔	**1.09**	6.68	7.70	2.34	3.28	✔	2.14	3.60
`psubw`	5.57	3.82	1.46	✔	**0.91**	6.12	3.36	1.60	2.10	✔	1.66	2.03
`psubd`	3.15	5.09	0.62	✔	**1.05**	3.01	2.69	1.81	1.49	✔	1.87	1.44
`psubq`	2.59	5.02	0.52	✔	**1.13**	2.29	3.62	1.98	1.83	✔	1.98	1.83
`pmulhw`	10.20	8.73	1.17	✘	**1.11**	9.16	4.78	1.86	2.58	✔	1.88	2.55
`pmulhuw`	6.50	7.24	0.90	✘	**1.04**	6.26	5.31	2.03	2.62	✔	2.00	2.65
`paddsb`	14.56	14.87	0.98	✘	**1.06**	13.79	15.06	2.39	6.29	✔	2.01	7.50
`paddsw`	11.33	10.44	1.09	✘	**0.91**	12.45	9.39	9.65	0.97	✘	1.84	5.10
`paddusb`	10.94	10.20	1.07	✘	**0.84**	12.96	11.96	1.91	6.26	✔	1.93	6.19
`paddusw`	5.96	5.98	1.00	✘	**1.08**	5.50	6.18	1.74	3.55	✔	1.73	3.58
`psubsb`	14.55	13.79	1.06	✘	**0.96**	15.21	14.87	1.85	8.06	✔	1.78	8.35
`psubsw`	15.34	14.09	1.09	✘	**1.01**	15.17	10.21	10.86	0.94	✘	1.78	5.73
`psubusb`	10.48	10.41	1.01	✘	**1.06**	9.86	12.88	2.17	5.94	✔	2.16	5.97
`psubusw`	6.00	5.41	1.11	✘	**0.94**	6.36	4.88	1.60	3.04	✔	1.63	3.00
`pcmpeqb`	15.19	16.26	0.93	✘	**0.88**	17.18	11.23	1.69	6.64	✔	1.85	6.07
`pcmpeqw`	6.14	5.96	1.03	✘	**0.97**	6.36	7.66	1.60	4.80	✔	1.66	4.63
`pcmpeqd`	3.87	3.71	1.04	✘	**1.02**	3.80	3.68	1.62	2.27	✔	1.70	2.17
`pcmpgtb`	17.61	14.26	1.23	✘	**0.87**	20.31	13.05	1.82	7.16	✔	1.93	6.75
`pcmpgtw`	6.13	6.05	1.01	✘	**0.87**	7.08	7.69	2.07	3.71	✔	1.91	4.03
`pcmpgtd`	3.47	3.82	0.91	✘	**1.00**	3.46	3.99	1.67	2.39	✔	1.68	2.38
`pmaxub`	11.36	10.90	1.04	✘	**0.97**	11.74	10.59	11.50	0.92	✘	1.94	5.46
`pmaxsw`	6.76	6.50	1.04	✘	**1.05**	6.42	5.95	1.68	3.55	✔	1.75	3.41
`pminub`	10.60	12.11	0.88	✘	**1.02**	10.41	10.38	11.66	0.89	✘	1.82	5.71
`pminsw`	6.46	5.84	1.11	✘	**1.04**	6.20	6.01	1.85	3.25	✔	1.84	3.26
`pavgb`	9.95	9.79	1.02	✘	**1.33**	7.50	10.98	1.78	6.18	✔	1.82	6.04
`pavgw`	6.18	5.88	1.05	✘	**0.93**	6.62	5.48	2.08	2.63	✔	2.11	2.59
`psadbw`	21.17	22.54	0.94	✘	**0.99**	21.50	14.93	18.37	0.81	✘	1.84	8.13
`pmaddwd`	5.00	5.39	0.93	✘	**1.22**	4.10	4.77	6.39	0.75	✘	2.07	2.31

Fig. 1. Summary of the results for 32 SIMD Integer Extensions of the x86 ISA.