Optimizing DSP and media benchmarks for Pentium 4: hardware and software issues

Daniel Etiemble

Department of Electrical and Computer Engineering
University of Toronto

ABSTRACT

By examining the speed-up resulting from using SIMD instructions for DSP kernels (FFT) and two different multimedia programs (MPEG2 and a Matching Pursuit video codec), we discuss the hardware and software issues that limit performance. Some features in present implementation of Intel SIMD instructions limit the efficiency of dot products. C programmer's habits also complicate the compiler tasks or in-lining of assembly code in many DSP and multimedia applications.

1. INTRODUCTION

SIMD instructions have been introduced in 1995 in general purpose microprocessors to improve the performance of multimedia applications. Intel has successively introduced MMX [1], SSE [2] and SSE2 [3]. The impact of SIMD instructions on DSP or multimedia kernels has been widely covered [4]. Intel has published many application notes [5] for FIR/IIR, FFT, DCT/IDCT, motion estimation, RGB Alpha Saturation, etc. But very few results have been published on the speed-up for the overall applications that result from using SIMD instructions, either automatically by the compiler, or by in-lining assembly code. In this paper, we consider the optimization issues of significant DSP kernels (FFT) and multimedia programs (MPEG2 and a Matching Pursuit Experimental Video CODEC) on a Pentium 4. These benchmarks either use integer data or simple precision floating point numbers

After this introduction, the second section presents the methodology and the overall results. The third section details the optimization of the FFT. The fourth section examines the optimization issues in the MPEG2 decoder, which is representative of the "integer" case. The fifth section examines some specificities of the floating point dot product. The last section concludes our findings.

2. METHODOLOGY AND OVERALL RESULTS

2.1 Benchmarks

The benchmarks that have been used are:

The SP complex FFT benchmark comes from Embree's book [5]. It has different lengths: 64, 256 and 1024 points and don't use the bit reversal step.

The encoder and the decoder for MPEG2 from the Software Simulation Group [7] mainly uses integer formats. For the test, we have used 15 yuv frames of three different sequences: Coast Guard, Paris and Paris_QCIF. The last two sequences come from the Research Group of PictureTel Corporation [8]. The chroma_format is 4:2:0. Only 15 frames have been used as they are sufficient to exercise the different components of the encoder. These frames fit in the Disk cache of the Pentium 4, which means that successive run is executed with data in memory.

The experimental matching pursuit video codec provided by the Video and Image Processing Laboratory of University of California at Berkeley [9] uses simple precision floating point data. The program has been tested with three sequences of frames: cont10.bit, hall10.bit and mom10.bit.

FFT and MPEG2 benchmarks are representative and widely used. The Berkeley MP codec is less known, but raises some interesting issues for vectorization. Together, they illustrate most of the hardware and software issues for an efficient use of SIMD instructions.

2.2 Measurements

All the measurements have been done on a 1.4 GHz Pentium 4 PC with 512MB memory running Windows 2000. We have used the Intel C++ 6.0 beta compiler through Microsoft Visual C++ environment with the "maximize speed" and QxW option. The QxW option generates specialized code for the Pentium 4. Each tested program has been measured at least 10 times and we have taken the averaged value. All the measures have been done with only one running application (Visual C++). The execution time has been measured with the RDTSC (read-time stamp counter) instruction available with IA32 [10]. The results are provided either as execution time (sec) or as speed-ups (old_execution time/new_execution time).

2.3 Overall results

Table 1 presents the overall results. The speed-up is significant for large complex FFTs (1.49 for 256 points, 1.66 for 1024 points). It ranges from 1.17 to 1.28 for the MPEG2 encoder and decoder. It is less than 1.12 for the Berkeley MP encoder (the decoder cannot be optimized). In the next sections, we briefly explain what is needed to get these results and why better results are not obtained.

BENCH.	INPUT	ORIG.	OPT.	SU
FFT	64	6,500 c	5,700 c	1.15
FFT	256	29,800 c	20,000 c	1.49
FFT	1024	153,900 c	92,700 c	1.66
MPEG-E	CG	2.44 s	1.93 s	1.26
	Paris	1.95 s	1.61 s	1.21
	Paris_qcif	0.54 s	0.46 s	1.17
MPEG-D	CG	277.1 ms	216.5 ms	1.28
	Paris	272.8 ms	212.8 ms	1.28
	Paris_qcif	104.3 ms	87.2 ms	1.19
MP- E	Cont10	5.94 s	5.56 s	1.07
	Hall10	6.78 s	6.30 s	1.08
	Mom10	2.81 s	2.50 s	1.12

Table 1: Execution times (clock cycles, ms or s) for original and optimized version with speed-up

3. OPTIMIZATION OF THE FFT

In the original version of the complex FFT [6],. three features prevent its vectorization by the compiler. First, it uses pointer to access arrays, according to the common mistake that believes that the compiler will always generate better target code [11]. Second, complex numbers are defined as a structure. Third, inner loops are non-unit stride. To allow vectorization, we have converted pointer to array accesses and replaced any occurrence of complex structures by two separated arrays of real and imaginary parts. It is more challenging to get unit-stride for the inner loop. Middle and inner loops must be interchanged and the coefficient array must be pre-computed in such a way that accesses to the coefficient are also unit-stride, as shown in the final simplified version of the vectorized code (Figure 1). In the complete version, the vectorization is prevented for the last two iterations of the outer loop (le=2 and le=1).

For the FFT, the first obstacle to the efficient use of the SIMD instructions is the DSP programmer's habits to use pointers to access arrays and structure to define complex numbers. Then restructuring code to get inner loops with unit-stride is a classical vectorization issue.

```
void fft_c(int n, float x_r[], float x_i[], float w_r[], float w_i[])
{register float t_r,t_i;
  int i,j,le,windex, k;
   windex = 1;k=0;
for(le=n/2 ; le > 0 ; le/=2,k++) {
  for (i = 0 ; i < n ; i = i + 2*le) {
     for (j = 0 ; j < le ; j++) {
            t_r=x_r[i+j]-x_r[i+j+le];
            t_i=x_i[i+j]-x_i[i+j+le];
            x_r[i+j]=x_r[i+j]+x_r[i+j+le];
            x_i[i+j]=x_i[i+j]+x_i[i+j+le];
x_r[i+j+le]=t_r*w_r[(n/2)*k+j]-t_i*w_i[(n/2)*k+j];
x_i[i+j+le]=t_r*w_i[(n/2)*k+j]+t_i*w_r[(n/2)*k+j];}}
          windex = 2*windex }}
```

Figure 1:Vectorized code for the FFT

4. MPEG2 CODEC

MPEG2 encoder and decoder programs raise issues that are characteristic of integer multimedia programs. In the encoder, one function (the motion estimation called dist1) consumes more than 50% of the execution time. It basically computes the sum of absolute values of the difference between two array of 16 bytes (chars). The function is so important that a specific SIMD instruction has been defined in the IA32 ISA, PSADBW, to compute the sum of absolute differences between 8 (resp. 16) successive unsigned bytes in the Pentium III (resp. Pentium 4) and to deliver an unsigned integer word (resp. two successive integer words). Using intrinsics to in-line SIMD instructions, the speed-up for the critical part of the dist function is 3 for the frames that we have used. One problem remains with this optimized version. The arrays of unsigned bytes are accessed by pointers and there is no way to insure that these pointers are aligned on 16-byte boundaries. Unaligned 16-byte transfers are far less efficient than aligned ones. We will be faced again to this problem with the post-filter functions of the Matching Pursuit Codec.

The dist function is a very specific case. A most common situation is illustrated by an extract of the code of the dct_type_estimation function in the encoder (Figure 2). This function is not a critical one but it outlines an intrinsic problem that prevents a direct use of integer SIMD instructions. The second loop computes several dot products of short integers. The results are 32-bit integers. These integers are converted to double FP by the next C line. Arithmetic operations on integer formats always deliver results that don't fit into the source format. With integer format, addition and multiplication of n-bit operands provide respectively n+1 and 2n bits results. Packed addition is possible when the final result allows either wraparound or saturation. When the exact result is needed, there is no possibility to use SIMD instruction as the carry out of each sub-word is lost. The normal version of integer multiplication or division is not easy to manipulate and is slower than the corresponding FP version. On the other hand, addition, multiplication and division of FP number always deliver a result with the same number of bits as the source operands.

```
void dct_type_estimation(pred,cur,mbi)
unsigned char *pred,*cur;
 {short blk0[128], blk1[128];
int i, j, i0, j0, k, offs, s0, s1, sq0, sq1, s01;
double d, r;
…
for (j=0; j<8; j++)
{offs = width*((j<<1)+j0) + i0;
for (i=0; i<16; i++)
{blk0[16*j+i] = cur[offs] - pred[offs];
blk1[16*j+i] = cur[offs+width] - pred[offs+width];
offs++;}}
…
s0=s1=sq0=sq1=s01=0;
for (i=0; i<128; i++)
{s0+= blk0[i];
sq0+= blk0[i]*blk0[i];
s1+= blk1[i];
sq1+= blk1[i]*blk1[i];
s01+= blk0[i]*blk1[i];}
d = (sq0-(s0*s0)/128.0)*(sq1-(s1*s1)/128.0); …}
```

Figure 2: Extract from function dct_type_estimation

Each loop in Figure 2 iterates 128 times. In the second loop, converting the *short* integers into *float* before entering the loop allows the vectorization of the dot products. With the "coast guard" sequence, the execution time of the original version of the loop and the next C line is 2060 cycles, while the modified version execution time is 1060 cycles. The speedup is close to 2. This should be balanced with the overhead of the conversion. In that specific function, the arrays blk0[128], blk1[128] are computed as the difference between arrays of chars. The first loop can be slightly modified to store its results into integers instead of shorts. A supplementary loop is used to convert the two integer arrays into the float arrays needed for the third loop. This conversion is realized by the corresponding packed conversion instruction (4 conversions per instruction). The original version of the 2 loops use 3850 cycles while the optimized version with 3 loops use 2550 cycles (speedup = 1.51).

In many cases where arithmetic operations on different formats of integers are involved, converting the integers to floats benefits loop vectorization and significantly reduces the execution time.

Another issue with integer multimedia programs is that declaration of integer variables. Programmers will generally distinguish between *char* and *int*, but the actual integer length is generally unspecified. For using SIMD instructions, it makes a huge difference if the integer is 16-bit or 32-bit long. Without an in-depth knowledge of the program, it is not straightforward to determine the exact range of each integer variable.

5. BERKELEY MATCHING PURSUIT CODEC

The Berkeley MP codec has also the motion estimation as a critical function. Other critical functions are the post-filter functions that operate on FP data. There are several instances of the post-filter functions, each implementing 3 different instances of the dot products with different array lengths. Figure 3 shows a simplified version of the corresponding code (only one of the three middle loops is shown).

```
void postfiltn(//……..//)
{float *basis;  *ppm,*ppm_end;
for(ppm=premult+bshift,y=0; y<sr; ppm+=ppm_yinc, ++y) {
    for (ppm_end = ppm+((tmp3<sr)?tmp3:sr);
          ppm < ppm_end;  ++ppm,  --tmp3){
          norm   = mpb->leftnorm[hbase][tmp3-1]*ynorm;
          ip = 0;
          for (k=0; k<n; k++)
                 ip+= basis[k]*ppm[k];…..}
   // two similar "middle" loops}
```

Figure 3: Extract of the postfilter function

This extract of the code is sufficient to understand the optimization issues. There are two important features. First, the array lengths are small: 1, 3, 5, 7, 9, 11, 13, 15, 21, 23, 27, 29 and 35. Second, all the dot products are inserted into middle loop that is controlled by a float pointer (ppm), which is used to access one of the two arrays of the dot products. As the middle loop increments the pointer, it means that the access to ppm cannot be aligned on a 16-byte boundary, which is a requirement for an efficient use of 16-byte data transfer instructions.

There are two ways to "vectorize" the dot products. The first one is equivalent to unroll 4 times the inner loop as shown in Figure 4. The final dot product needs to sum the four parts of an XMM register. As there is no MMX instruction to sum the four parts of an XMM register, six data-dependent instructions are needed to get the final sum in the XMM register and one more is needed for the memory transfer. The second technique computes four successive dot products simultaneously by unrolling four times the middle loop (Figure 5). The overhead of the final sum is suppressed. The alignment issue can also be suppressed.

Each "packed" iteration of the inner loop needs four times *basis[k]* in a *XMM* register. At the beginning of the function, the basis array is 4 times expanded and copied in another data-aligned array. The other *XMM* register must contain *ppm[k] ppm[k+1], ppm[k+2], ppm[k+3]*. For the next iteration, the register is shifted left by 8 bytes and the new *ppm[k+3]* is loaded by the scalar *movss* instruction. Except for the initialization phase that needs four scalar loads (and three shifts), all the subsequent accesses to *ppm* only require one scalar load and one shift per iteration. The final *ppm* iterations (when *ppm* % 4 != 0) are computed by using the first version. Although this version is more efficient than the previous one, there is still cost for generating the data-aligned duplicated *basis* array and to initially fill up the "*ppm*" *XMM* register. Both versions have similar drawbacks: the unaligned 16-byte access that comes from the pointer loop index and the overheads make vectorization inefficient when the array length is less than 23.

```
ip[0] = 0.0; ip[1] = 0.0; ip[2] = 0.0; ip[3] = 0.0;
    for (k=0; k<n/4; k+=4){
      ip[0]+= basis[k]*ppm[k];
      ip[1]+= basis[k+1]*ppm[k+1];
      ip[2]+= basis[k+2]*ppm[k+2];
      ip[3]+= basis[k+3]*ppm[k+3]; }
      ip=  ip[0]+ip[1]+ip[2]+ip[3];
    for(;k<n;k++)
     ip+= basis[k]*ppm[k]; //epilogue when n%4 != 0
```

Figure 4: Unrolling 4 times the inner loop

```
for(ppm_end=ppm+((tmp3<sr)?tmp3:sr); ppm<ppm_end-4;
ppm+=4, tmp3-=4){
…..
    ip[0] = 0; ip[1] = 0; ip[2] = 0; ip[3] = 0;
    for (k=0; k<n; k++){
      ip[0]+= basis[k]*ppm[k];
      ip[1]+= basis[k]*ppm[k+1];
      ip[2]+= basis[k]*ppm[k+2];
      ip[3]+= basis[k]*ppm[k+3]; }  …}
```

Figure 5: Unrolling four times the middle loop

6. CONCLUSION

SIMD instructions can provide a spectacular speedup on some specific functions as the motion estimation of the MPEG encoder. The impact of SIMD instructions is more limited overall. The Amdahl law is a natural explanation as only part of the code can be vectorized. However, there are some issues to solve to improve the efficiency of SIMD instructions for DSP and multimedia applications.

A limitation comes from the present implementation of SIMD instructions in the Pentium 4. The most critical issues are the poor performance of the unaligned 16-byte move instructions and the lack of "horizontal" operations such as the sum of the 4 32-bit words of a XMM register, which makes the vectorization of the dot products inefficient for small arrays.

Another issue is the nature of integer arithmetic computations that prevent using SIMD instructions when exact computation is needed. Converting integer data into FP data can overcome the issue as SIMD instructions can be employed.

One other limitation comes from the C programmer's habits. The systematic use of pointers and structure prevents compiler vectorization. A precise choice of integer formats would also help. As the present programs have not been written considering the SIMD instructions, the optimization is difficult. Considering the SIMD instructions while writing the programs will significantly improve their efficiencies.

7. REFERENCES

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

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

[3] http://developer.intel.com/software/products/college/ia32/sse2/

[3] R.Bhargava, L.K. John, B. L. Evans, R.Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications", *In Proc. of the IEEE Symposium on Microarchitecture*, pages 37-46, 1998.

[4] http://www.intel.com/software/products/itc/strmsimd/

[5] P.M. Embree, "C Algorithms for Real-Time DSP", *Prentice-Hall*, ISBN 0-13-337353-3, 1995

[6] http://www.mpeg.org/MPEG/MSSG/#source

[7] http://standard.pictel.com/ftp/video-site/sequences/.

[8] Sen-Ching Sanson Cheung, Avideh Zahkor, "Matching Pursuit Experimental Video Codec", http://www-video.eecs.berkeley.edu/download/mp.

[9] Intel, "Using the RDTSC Instruction for Performance Monitoring", Application Note, www.intel.com

[10] B. Frande and M. O'Brien, Compiler transformation of pointers to explicit array accesses in DSP applications, *in Proc. of the ETAPS Conf. On Compiler Construction*, LNCS 2027, pages 69-85, 2001.