16-bit floating point operations for low-end and high-end embedded processors

L. Lacassagne[1], D. Etiemble[2]

Abstract— **We have simulated the implementation of 16-bit floating point instructions on a Pentium4 and PowerPC G5 to evaluate the performance impact of these instructions in embedded processors for graphics and multimedia applications. Both accuracy of the computations and the execution time have been considered. For low-end embedded processors without SIMD instructions, the 16-bit FP instructions deliver a larger dynamic range than 16-bit integer with the same memory footprint. For high-end embedded processors with SIMD instructions, we add the speed up coming from wider SIMD instructions.**

Index Terms—**16-bit floating point instructions, SIMD extensions, vision and multimedia, embedded processors and applications.**

I. Introduction

Graphics and media applications have become the dominant ones for general purpose microprocessors and correspond to a very large segment of embedded applications. They have led to the introduction of specific instruction set extensions such as the SIMD extensions which are available both for general purpose processors (such as SSE/SSE2/SSE3 extensions for IA32, Altivec for PowerPC) or embedded processors (such as. SIMD extensions for ARM ISA implemented in Xscale processors).

Image processing generally need both integer and FP formats. For instance, vImage [1], which is the Apple image processing framework, proposes four image types with four pixel types: the first two pixel types are unsigned byte (0 to 255) and float (0.0 to 1.0) for one color or alpha value and the two other pixel types are a set of four unsigned char or float values for Alpha, Red, Green and Blue. Convolution operations with byte inputs need 32-bit integer formats for the intermediary results. Geometric operations need floating point formats.

For media processing, the debate between integer and FP computing is also open. On one hand, people argue for using fixed-point computation instead of floating point computation. For instance, G. Kolly justifies "using Fixed-Point Instead of Floating Point for Better 3D Performance" in the Intel Graphics Performance Primitives library in [2]. Techniques for automatic floating-point to fixed-point conversions for DSP code generations have been presented [3]. On the other hand, people propose lightweight floating point arithmetic to enable FP signal processing applications in low-power mobile applications [4]. Using IDCT as benchmark, the authors show that FP numbers with 5-bit exponent and 8-bit mantissa are sufficient to get a Peak-Signal-to-Noise-Ratio similar to the PSNR with 32-bit FP numbers. These results illustrate one case for which a format using less than 16 bits would deliver performance equivalent to the 32-bit FP format. This debate is important for embedded applications, for which the needed performance should be obtained with the minimum memory occupation and minimum power dissipation.

A 16-bit floating point format would combine the memory occupation of 16-bit integer and provide a larger dynamic range. Such formats have been defined a long time ago in some DSP processors. For instance, the TMS 320C32 [5] has an internal 16-bit type with 1 sign bit, a 4-bit exponent and an 11-bit fraction that can be used as an immediate value by FP instructions and an external 16-bit type with 1 sign bit, a 8-bit exponent field and a 7-bit fraction which is used for storage purposes. By they are rarely used. Recently, a 16-bit floating point format called "half" has been introduced in the OpenEXP format [6] and in the Cg language [7,8] defined by NVIDIA. It is currently used in some NVIDIA GPU. It is justified by ILM, which developed the OpenEXP graphics format, as a response to the demand for higher color fidelity in the visual effect industry: "16-bit integer based formats typically represent color component values from 0 (black) to 1 (white), but don't account for over-range value (e.g. a chrome highlight) that can be captured by film negative or other HDR displays… Conversely, 32-bit floating-point TIFF is often overkill for visual effects work. 32-bit FP TIFF provides more than sufficient precision and dynamic range for VFX images, but it comes at the cost of storage, both on disk and memory". Similar arguments are used to justify the "half" format in Cg language.

In [9], we have presented some preliminary results on the performance evaluation of a "half" SIMD extension for the Intel IA-32 ISA and the PowerPC Altivec extension. The context was the extension of "multimedia" capabilities of general purpose microprocessors, and we mainly focused on the speed-up resulting from wider SIMD instructions and the

[1] IEF, CNRS and Université Paris Sud, email : lacas@ief.u-psud.fr
[2] LRI, CNRS and Université Paris Sud, email : de@lri.fr

“vectorization issues”. In this paper, after a brief summary of the prior results, we focus on the impact of 16-bit FP instructions for embedded processors. A scalar version of these instructions is considered for low-end embedded processor while the impact of SIMD instructions is still considered for high-end embedded processors. While prior results only considered the execution time, we now both consider the accuracy issues and the execution time for the different benchmarks.

II. SUMMARY OF PREVIOUS RESULTS

In [9,10], we have considered the speed-up between versions using SIMD 16-bit floating point instructions (called F16) and versions using 32-bit floating point instructions (called F32). The speed-ups have been measured on general purpose processors: Pentium 4 and PowerPC G5. The following benchmarks have been used:

- Horizontal and Horizontal-Vertical versions of the Deriche filters.
- Deriche gradient
- Scans: +scan and +*scan.
- Bounding boxes of triangle in tri-strip format (OpenGL).

For Deriche filters, the Pentium SIMD F16 versions exhibit a speed-up close to 2 versus the float versions and the speed-up between F16 and 32-bit FP versions ranges from 2.2 to 4.2 for the G5 processors.

For the +scan, which is memory-bounded and has execution times close to the execution time of a simple copy, F16 instructions are useless. For +*scan, which accumulate both the values and the square of values of all the image pixels, both Pentium 4 and PowerPC G5 exhibit a speed-up slightly less than 2 with F16 versus F32.

For the OpenGL benchmark, the speed-up is 1.8 for Pentium 4 and 2 for the G5.

In any case, the speed-up derives from the wider SIMD instructions (two times more operations per SIMD instruction) and the smaller data cache footprint (16-bit format versus 32-bit format). Due to the instruction latencies that are used, the Pentium 4 results are slightly pessimistic while the G5 results are slightly optimistic.

A rough evaluation of the 16-bit FP operators has also been given. Based on a cell-based estimated implementation derived from a VHDL description of the operators, it was shown that the chip area for 8 16-bit FP functional units (Addition-subtraction, Multiplication, Division and Square root) would be about 11% of the corresponding chip area of 4 64-bit FP functional units.

III. METHODOLOGY

To evaluate the interest of adding one F16 format within embedded processors, several issues must be considered:

- The accuracy of F16 computations compared to integer ones or usual single precision or double precision IEEE floating point formats for graphics and multimedia applications

- The execution times for different benchmarks when using F16 formats compared to F32 formats without and with SIMD instructions.

- The hardware cost for the F16 operators in embedded systems. As F16 will be used for specific applications, the hardware evaluation must be considered for reconfigurable hardware (FPGAs including processor on chip), SoCs or customizable architectures such as the Xtensa processor.

- Only the two first issues are considered in this paper. The third issue has not yet been extensively studied. The results will be published later.

A. *The NVIDIA Half format*

The “half” format is presented in figure 1. A number is interpreted exactly as in the other IEEE FP formats. The exponent is biased with an excess value of 15. Value 0 is reserved for the representation of 0 (Fraction =0) and of the denormalized numbers (Fraction ≠ 0). Value 31 is reserved for representing infinite (Fraction = 0) and NaN (Fraction ≠ 0). For 0<E<31, the general equation for calculating the value in a floating point number is (-1)S x (1.fraction) x $2^{(\text{Exponent field-15})}$. When using denormals, the range of the format extends from $2^{-24} = 6 \times 10^{-8}$ and $(2^{16}-2^{5}) = 65504$. Without denormals, the range extends from 2^{-15} to 65504. To tackle the dynamic problem, the F16 numbers can be "normalized" during conversion by changing the pixel range from [0 - 255] to [0.0-1.0] as for the other floating-point formats. In the remaining part of this paper, this 16-bit floating point format will be called half or F16 and the IEEE-754 32-bit FP simple precision format will be called F32.

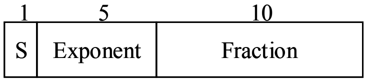

FIGURE 1 : NVIDIA “HALF” FORMAT

B. *Accuracy issues*

For checking the F16 accuracy compared to usual 32-bit FP accuracy, we started with the original 32-bit FP version of different benchmarks and we wrote different functions to simulate the 16-bit FP range within 32-bit FP numbers. With this approach, we were able to run the different benchmarks by just inserting the appropriate function before any program use of a “float” variable and after any program computation on “float” numbers: any “float” source operand and any “float” result of a “float” operation remains “float” numbers, but with the precision and accuracy of “half” number. This approach is machine independent. The following functions have been tested:

- Truncation without denormals: the fraction is truncated from 23 bits to 10 bits by discarding the 13 low-order bits. Denormals are transformed into zero.
- Rounding to nearest without denormals: the fraction is reduced from 23 bits to 10 bits with rounding to nearest according to IEEE-754 standard. Denormals are transformed into zero.

- Truncation with denormals: truncation of the fraction and standard definition for denormals
- Rounding to nearest with denormals: rounding of the fraction and standard definition of denormals.

By using each one of these functions on the different benchmarks, we compared the different images that were transformed by each benchmark with the "Peak-Signal-to-Noise-Ratio" metrics. Different hardware implementations of the F16 arithmetic operators would correspond to the different function. If truncation without denormals gives similar results compared to the other solutions, then the simplest hardware implementation would be possible, which is the ideal situation for embedded hardware.

C. *Execution time of scalar versions*

As in [9,10], we used measured execution times on actual processors to evaluate the performance of the F16 version of the considered benchmarks. Pentium 4 and PowerPC G5 processors were used. The Pentium 4 is a 2.4 GHz Xeon processor with 1 GB memory running Windows XP. The benchmarks have been compiled with the Intel C++ 8 compiler and the QxW. The execution time has been measured with the RDTSC (read-time stamp counter) instruction available with IA-32. All the measures have been done with only one running application (Visual C++). The PowerPC is a 1.6 GHz PowerPC G5 with 768 MB DDR400 running Mac OS X.3. The programs have been compiled with the Xcode programming environment including gcc 3.3.

It may look strange to use general purposed processors to evaluate extensions for embedded processors, but we do feel that most results that we obtained can be extrapolated to embedded processors. By using these processors, we basically use the execution times of the different instructions that are specific to these processors, and the cache hierarchies that are also specific to these processors. Embedded processors would have different instructions, different instruction latencies and different data cache sizes. The quantitative results would be different, but we do believe that the qualitative results would be similar as the impact of the data cache and larger SIMD instructions would be qualitatively the same when using 16-bit floating point data instead of 32-bit floating point data. As previously mentioned, further works will consider the actual implementation of F16 operators in configurable hardware or SoCs. One advantage of using general purpose processor is to profit from the richer available SIMD extensions and software tools (vectorizing compilers, SIMD intrinsics).

We are basically interested in the differences between the execution times of the F16 version andthe 32-bit FP version of the different benchmarks. Sometimes, the 16-bit integer version (called I16) can be executed without transforming the benchmark. In that case, the accuracy is generally insufficient (and the program results are wrong) but it gives a good insight on the lower bound for the execution time with 16-bit formats.

For scalar and SIMD versions of the F16 instructions, we consider that the F16 versions of the arithmetic instructions have the same latency and throughput than the F32 instructions. This assumption is pessimistic because F32 instructions are executed in a Pentium 4 by the same operators as the 64-bit double precision instructions (SP and DP instructions have the same latency and throughput). 16-bit FP operators are obviously faster than 64-bit operators, but the operation latency is only part of the overall instruction latency, which is mainly determined by the insertion of the operator in the processor data-path. Anyway, considering that F16 and F32 instructions have the same latencies means that our results are more pessimistic than optimistic.

When the 16-bit and 32-bit integer versions can be executed on a benchmark, the execution time of the F16 version is evaluated as follow:

$T_{F16} = T_{I16} + T_{F32} - T_{I32}$

This formula derives from the following assumptions: The F16 instructions have the same latencies as the F32 ones, while the cache behavior is the same as with I16 instructions. The term (T_{F32}-T_{I32}) expresses the difference between the FP and integer execution times, while T_{I16} considers the cache behavior with 16-bit integer. This cache behavior is the same with F16 data and I16 data.

D. *Execution time of SIMD versions*

The methodology for SIMD versions is the same as in [9,10]. For using SIMD instructions, we cannot only rely on the compiler to vectorize. Intrinsics are assembly-coded functions that enable the programmer to use expanded inline C/C++ function calls and variables in place of assembly mnemonics and registers. With "intrinsics" within C programs to manually use the SIMD instructions, we can use different "intrinsics" to simulate the F16 SIMD instructions with different latencies.

The graphics and media benchmarks have a nice specificity: the kernel computation consists in loop nests which are not data dependant (the loop iterations only depend on the loop bounds that are defined at compile time). In this situation, the "simulated" instructions can be replaced by any "actual" instruction with given latency and throughput figures. There are basically three constraints:

- the cache accesses should be the same for the simulated and actual memory instructions
- The data dependencies should be strictly enforced.
- As it is no longer possible to check the results, we must carefully check that the compiler generates all the required instructions according to the data dependencies.

For instance, the IA32 SIMD MULPS (packed floating point multiplication) can be used to simulate a MULF16 (packed *half* multiplication with the same latency (6 cycles) and throughput (2 cycles). The situation is the same for the Altivec instructions of the PowerPC.

1) *Pentium 4*

As the reference processor, we considered the current version of the Pentium 4 (without hyperthreading technology) including the SSE3 extension.

We assume the currently available 128-bit XMM register set. With 128-bit registers and data path, the number of functional units for each SIMD F16 operation is 8. The issues to solve are: the conversions between bytes to/from F16 data, the type of FP operators and the permutation and formatting operations that must be added for the F16 format.

Byte to F16 conversion means converting 8 packed bytes into 8 packed F16 "half". IA32 ISA having a (2,1) instruction format, the source operand can be either a register or a memory operand. One could define a conversion from a 64-bit memory operand (or the lower part of an XMM register) into an XMM register. The other option consists in defining two different byte-to-F16 conversion instructions as shown in Figure 2. After loading the XMM register with a 16-byte memory access, the first conversion instruction would convert the lower part of the XMM register into 8 packed F16 in another register and the second one would convert the higher part. This second conversion instruction is not absolutely necessary, but avoids an intermediate move/shift from the upper part to the lower part of the source register. These conversion instructions are register only instructions. The opposite F16 to byte conversion are needed. The second option is more efficient. It leads to implicitly unroll two times any loop (the lower 8-byte operands first, then the higher ones).

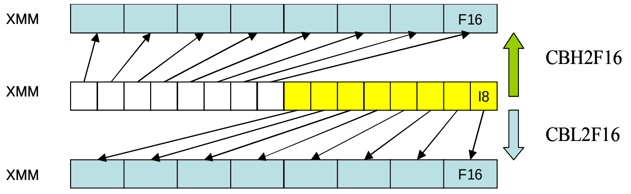

FIGURE 2: BYTE TO F16 CONVERSION INSTRUCTIONS

To be able to deal with the complete F16 format, the FP operators that are needed are the same as the ones that are available for single and double precision formats: the packed F16 addition/subtraction, multiplication, division and square root operators. Obviously, only the needed operations would be implemented in embedded applications. Bitwise logical operations are the same for F16 formats as for any other format. Shuffle and pack/unpack instructions can be more efficiently executed on the original byte data before conversion for byte stored arrays, but they are needed for "half" stored arrays. When F16 data are stored, all the shuffle or packing/unpacking instructions that are now available for 16-bit data can be used but these operations should be extended to the 8 different slots, which raise a small difficulty. The shuffle or packing/unpacking operations are defined by an 8-bit immediate in the IA-32 ISA, which is OK with four slots. Keeping an 8-bit immediate with eight slots would need a coding of the different operations on the eight slots. Table 1 lists the different F16 IA-32 instructions that we used in our benchmarks. All the proposed instructions have a throughput value of 2.

To be able to completely treat the 16-bit FP format, *short* from/to *half* conversions are also needed. Load and store packed instructions for *half* data are similar to the already packed integer instructions.

A program using the F16 instructions is directly derived from the corresponding F32 one by replacing the F32 instructions by F16 instructions. As previously mentioned, the F16 instructions are presently simulated by actual instructions fo the IA-32 or PowerPC ISA.

TABLE 1: 16-BIT FP INSTRUCTIONS FOR PENTIUM 4

Instruction	Latency (Max)	Meaning
ADDF16	4	Xmmd <- Xmmd+Xmms
SUBF16	4	Xmmd <- Xmmd-Xmms
MULF16	6	Xmmd <- Xmmd * Xmms
MAXF16	4	Xmmd <- Xmmd max Xmms
MINF16	4	Xmmd <= Xmmd min Xmms
CBL2F16	4	Xmmd <= BytetoF16 (Xmms low)
CBH2F16	4	Xmmd<= BytetoF16 (Xmms high)
CF162BL	4	Xmmd low<= F16toByte (Xmms)
CF162BH	4	Xmmd high<= F16toByte(Xmms)
SHUFF16	4	Xmmd <= shuffle (8 slots) Xmms

2) *PowerPC G5*

We consider the actual implementation of the G5 processor [11] with Altivec extension and the instruction latencies [12] given in Table 2

TABLE 2: G5 INSTRUCTION LATENCIES

Execution Unit:	cycles
IU (+, -, logical, shift)	2-3
IU (multiplication)	5-7
FPU (+, -, *, MAF)	6
LSU (L1 hit) to GPR, FPR, VR	3,5,4-5
LSU (L2 hit, loads only)	11
VPERM	2
VSIU (part of VALU)	2
VCIU (part of VALU)	5
VFPU (part of VALU)	8

As the Altivec extension is rather complete, the only supplementary needed F16 instructions are the F16 version of the vector FP instructions and the byte to/from F16 conversion instructions. All the packing/unpacking and permutation instructions that are needed are already available for short integer operands. The simulated conversion instructions have a latency of 2, which may be a little bit optimistic. The F16 multiplication-accumulation instruction, which is used for F16 add, mul and mul-add, has a latency of 5. Compared to our Pentium 4 simulation of F16 instructions that were pessimistic, our G5 simulation are slightly optimistic.

E. Benchmarks

We considered three different benchmarks.

The first one has been furnished by Prof. A. Montanvert (LIS, Grenoble) and has been written by a Ph. D student (L. Condat). The benchmark consists in Image zooming and interpolation by power of 2 by using splines [13]. The benchmark has three zoom values: 1, 2 and 4. The first one is used to validate the filters that are used.

The second one is the JPEG codec of Mediabench suite [14]. It includes both a DCT (coder) and IDCT (decoder) with three different versions: float, integer and "fast" integer. We added the F16 version for the DCT and the IDCT. The third one is a Wavelet benchmark, which uses the SPHIT (Set Partitioning in Hierarchical Trees) [15]. It uses the same set of filters than the wavelet transform used in JPEG2000, but the compression stage is smaller than the one in the JPEG encoder.

We used a large set of images of different sizes:

- 1024x1024 images: man_1k
- 512x512 images: baboon, Barbara, Lena, lighthouse, peppers
- 128x128 images: corridor, Einstein, Grenoble, Lena, office, Titanic.

IV. MEASURED RESULTS

A. *Zooming and interpolation benchmark*

We focused on accuracy issues. With the x2 and x4 zoom factors, the differences between the zoomed images computed with F32 and F16 data is always greater than 55 dB. More, the PSNR values are the same for truncated and rounded F16, with or without denormals. This result is important as it shows that the simplest version of F16 operators is sufficient.

For the useful zoom values, the F16 version is 19% faster than the corresponding F32 one. Without zoom (x1), the speed up is 1.8, which shows the impact of the reduced cache footprint as there is few computation for this zoom value. As expected, the speed-up comes from the better cache behavior. The resulting speed-up depends on the ratio between data accesses and computations.

The most time consuming function in the zooming application exhibits a loop-carried dependency which prevents using SIMD instructions.

B. *JPEG codec*

On this benchmark, we only considered the accuracy of computations. Figures 3 and 4 and show the difference in dB between the compressed/decompressed images and the original one according to image sizes and the different versions of the DCT and IDCT implementation. For 256x256 images, there are no significant differences, which means that integer implementation is the most efficient (even the faster one which degrades accuracy for speed). For 512x512 images, the result is image dependant. If Lighthouse exhibits the same behavior as smaller images, Baboon and Lena shows the improvement resulting from FP computations and that F16 has the same accuracy as F32 while using half the memory occupation. The evaluation of execution times remains to be done for this benchmark.

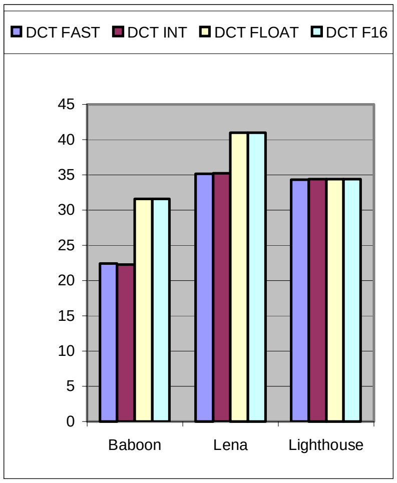

FIGURE 3: PNSR DIFFERENCE BETWEEN COMPUTED AND ORIGINAL 512X512 IMAGES

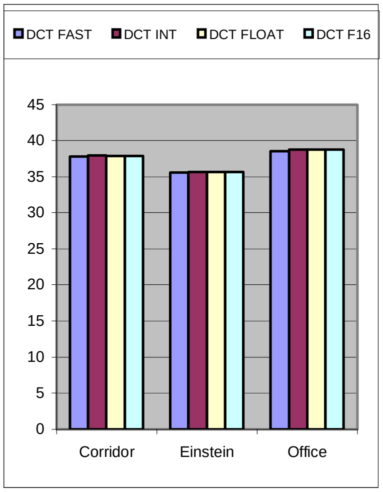

FIGURE 4: PSNR DIFFERENCE BETWEEN COMPUTED AND ORIGINAL 256X256 images

C. *Wavelet transform*

The wavelet transform is different from the previous benchmarks, as the transform (low-pass & high-pass filter) is recursively applied on a sub-quadrant of the image (figure 5).

$$L(n) = l_0 x_n + l_1(x_{n+1}+x_{n-1}) + l_2(x_{n+2}+x_{n-2}) + l_3(x_{n+3}+x_{n-3}) + l_4(x_{n+4}+x_{n-4})$$
$$H(n) = h_0 x_n + h_1(x_{n+1}+x_{n-1}) + h_2(x_{n+2}+x_{n-2}) + h_3(x_{n+3}+x_{n-3})$$

L filter is applied on even pixels and *H* filter to odd pixels to create 2 half size images. *ImageL(n) = L(X(2n))*, *ImageH(n) = H(X(2n+1)*. To limit memory occupation, computations are done in situ: a line X (respectively a column) is copied into a T buffer for border symmetry; the filters are then applied as follow: *Y(k) = L(T(2k)),* Y(k+k) = H(T(2k+1), with k∈[0..*n/2*], or *Y(k/2) = L(T(k)), Y(n/2+k/2) = H(T(k+1))* with k∈[0..*n*]. Filters (figure 6) are applied down to 32x32 image size, which means 6 iterations for 1024 x 1024 images, 5 for 512x512 and 4 for 256x256 images.

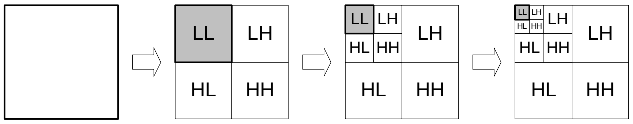

FIGURE 5 : WAVELET TRANSFORM

Compared to the F32 version, the PSNR difference is about 3.5% for low compression factor (1 to 20) and less than 0.5% for higher factors when using F16. In any case, the difference is smaller than 3 dB.

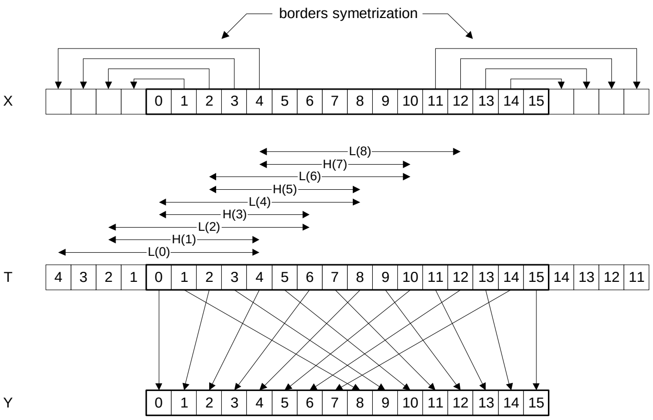

FIGURE 6: WAVELET FILTERS

Because of the interlaced filter applications, vectorization is a little bit more difficult than usually. There are 5 and 4 multiplications instead of 9 and 7 for non symmetric filter coefficients. There is a large overlap of filter applications, with each filter contributing to 9 filter computations. As the filter sizes are not a power of 2, the SIMD reduction instructions are un-efficient. We used the same technique as for the Deriche benchmark (figure 7): a horizontal (respectively vertical) band of pixels is copied into a B band for which the memory accesses are horizontal. Such a band greatly reduces the cache misses during the vertical application of filters (if the Loads are not factorized, each pixel is reloaded 9 times). Then a block transposition is applied to B to load the registers with values orthogonal to the filter.

To explain the caches behavior, the non recursive wavelet transform (only one iteration) has been applied to different size images, which sizes vary from 32x32 to 1024*1024 (figure 8 and 9).

SIMD performances are shown in Tables 3 and 4 for PowerPC and Pentium 4. The metrics is CPP (i.e. number of clock cycles per pixel). The SIMD versus scalar speed-up is shown in table 3 and 4. These results can be explained as follow: the block transposition needs 8 instructions for four vector 32-bit numbers and 24 instructions for 8 vectors of 16-bits, that is 0.5 instruction per 32-bit pixel and 0.375 instruction per 16-bit. The speedup comes from the parallelism increase and from the cache footprints. F16 parallelism provides a speedup of about x2 compared to F32 parallelism (same number of instructions and operation per vector, but twice the number of processed pixels), while cache footprint modifies this speedup from 1.8 to 4 for PowerPC. The maximal speedup is achieved when F16 data fit in the cache, while F32 data do not (for 512x512 images). For the recursive wavelet transform, the same remarks can be done.

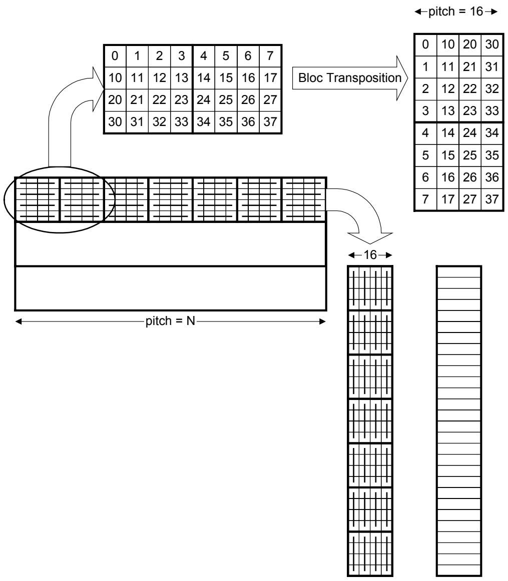

FIGURE 7: BLOCK TRANSPOSITION USED IN DERICHE AND WAVELET BENCHMARKS

TABLE 3: SIMD CPP FOR POWERPC AND PENTIUM 4

Image size	256	512	1024
PowerPC F32	7.9	15.7	22.8
PowerPC F16	4.2	5.2	9.3
PowerPC gain	1.9	3.0	2.5
Pentium F32	13.4	18.9	20.0
Pentium F16	6.9	8.2	11.6
Pentium gain	1.9	2.3	1.7

TABLE 4: SIMD VERSUS SCALAR SPEED-UP FOR POWERPC AND PENTIUM 4

Image size	256	512	1024
PowerPC F32/f32	3.3	2.5	2.4
PowerPC F16/f16	6.7	5.5	4.2
Pentium F32/f32	2.9	2.6	2.9
Pentium F16/f16	5.5	5.1	4.1

V. CONCLUDING REMARKS

We have continued the evaluation of 16-bit floating point operators and instructions for graphics and multimedia applications. While our prior results [9,10] mainly focused on the impact of SIMD 16-bit FP instructions on general purpose processors, we have extended the evaluation to the accuracy issues and the impact of scalar versions for embedded processors and applications.

It is obvious that the F16 format will be useful only if it closes a gap between the 16-bit integer format and the 32-bit floating point format by having the memory occupation of the

16-bit format and a larger dynamic range than the 16-bit integer format. When “short” format is sufficient, there is no need for the “half” format. This is true for scalar computation. For SIMD computation, the situation is somewhat different as floating point vectorization is far easier than integer vectorization.

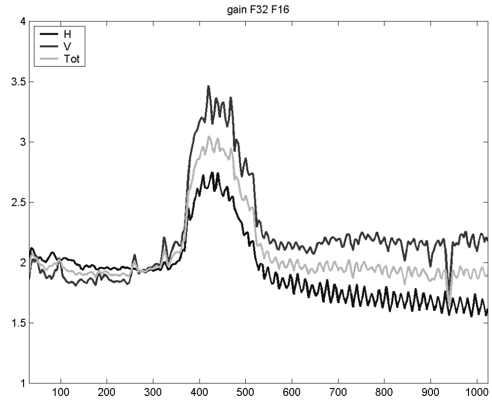

FIGURE 8 : F16/F32 SIMD SPEED-UP FOR PENTIUM 4 ACCORDING TO IMAGE SIZES

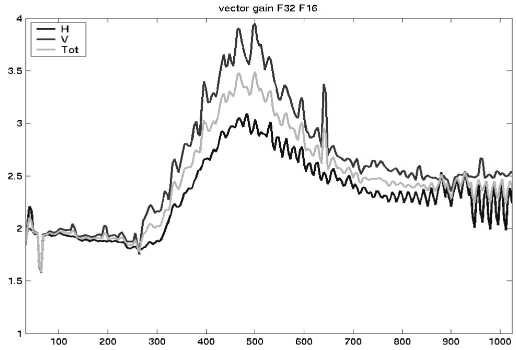

FIGURE 9: F16/F32 SIMD SPEED-UP FOR POWERPC ACCORDING TO IMAGE SIZES

For all the benchmarks that we have considered, F16 format delivers the same accuracy as F32 when FP computations are needed. More, we have shown than the F16 operators could be simple. As truncation can be used instead of rounding, denormals are useless. As embedded applications don’t need the complete set of FP operators, this means that implementing F16 computation in low-end embedded processors could be a valuable solution, while the usual FP 32-bit format is too costly. Compared to the F32 scalar execution time, the F16 scalar execution time speed-up is limited: it only comes from the better cache behavior resulting from the smaller cache footprint.

Considering the impact of SIMD F16 instructions, the Wavelet benchmark confirms the previous results on Deriche, scan, and OpenGL benchmarks: the combined effect of wider SIMD instructions and smaller cache footprint gives a significant speed-up, generally ranging from slightly less than 2 to more than 2.

This work will be continued by considering more benchmarks. Experiments on actual embedded processors will be done. Evolution of SoC technologies, including customizable CPU architectures such as the Xtensa processor [16], opens new opportunities for a more precise evaluation of 16-bit floating point instructions.

REFERENCES

[1] Apple, “Introduction to vImage”, http://developer.apple.com/documentation/Performance/Conceptual/vImage/

[2] G. Kolli, “Using Fixed-Point Instead of Floating Point for Better 3D Performance”, Intel Optimizing Center, http://www.devx.com/Intel/article/16478

[3] D. Menard, D. Chillet, F.Charot and O. Sentieys, “Automatic Floating-point to Fixed-point Conversion for DSP Code Generation”, in International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2002)

[4] F. Fang, Tsuhan Chen, Rob A. Rutenbar, “Lightweight Floating-Point Arithmetic: Case Study of Inverse Discrete Cosine Transform” in EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication Systems

[5] Texas Instruments,TMS 320C3x User’s guide, http://focus.ti.com/lit/ug/spru031e/spru031e.pdf

[6] OpenEXP, http://www.openexr.org/details.html

[7] W.R. Mark, R.S.Glanville, K. Akeley and M.J. Kilgard, “Cg: A system for programming graphics hardware in a C-like language.

[8] NVIDIA, Cg User’s manual, http://developer.nvidia.com/view.asp?IO=cg_toolkit

[9] D. Etiemble, L. Lacassagne,“SIMD 16-bit FP instructions for image and media processing on general purpose microprocessors”, ODES 2004, Palo Alto, March 2004

[10] D. Etiemble, L. Lacassagne, “16-bit FP sub-word parallelism to facilitate compiler vectorization and improve performance of image and media processing”, in Proceedings ICPP 2004, Montreal, Canada

[11] T. R. Halfhill, “IBM trims Power4, adds Altivec”, in Microprocessor Report, 10/08/02

[12] Apple Developer Connection, “G5 performance programming”, http://developer.apple.com/hardware/ve/g5.html

[13] M. Unser, “Spline, A perfect fit for signal and image processing”, in IEEE Signal Processing Magazine, November 99, pp 22-38.

[14] C. Lee, M. Potkonjak, W.H. Mongione-Smith, “Mediabench : A Tool for Evaluating and Synthetising Multimedia and Communication Systems”, Proceeding Micro-30 conference, Research Triangle Park, NC, December 1995.

[15] A. Said, W. A. Pearlman, "A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 6, pp. 243-250, June 1996.

[16] T.R. Haflhill, “Tensilica tackles bottleneck”, in Microprocessor Report, May 31, 2004