# 16-bit FP sub-word parallelism to facilitate compiler vectorization and improve performance of image and media processing

Daniel Etiemble *, Lionel Lacassagne**
**LRI, ** IEF, University of Paris Sud*
*91405 Orsay, France*
*{ de@lri.fr , lionel.lacassagne@ief.u-psud.fr }*

**Abstract**

*We consider the implementation of 16-bit floating point instructions on a Pentium 4 and a PowerPC G5 for image and media processing. By measuring the execution time of benchmarks with these new simulated instructions, we show that significant speed-up is obtained compared to 32-bit FP versions. For image processing, the speed-up both comes from doubling the number of operations per SIMD instruction and the better cache behavior with byte storage. For data stream processing with arrays of structures, the speed-up mainly comes from the wider SIMD instructions..*

## 1. Introduction

Graphics and media applications have become the dominant ones for general purpose microprocessors and have led to the introduction of specific instruction set extensions such as the SIMD multimedia extensions now available in most ISAs. Graphics and media applications either use integer or FP computation. While some applications need the dynamic range and accuracy of 32-bit FP numbers, a general trends is to replace FP by integer computations for better performance when hardware resources are limited (embedded applications). However, many integer computations must deal with several integer formats, which forbid the compiler to use SIMD instructions. Using "intrinsics" or assembly language is tiresome and time consuming or even impossible with integer formats. In this introduction, we discuss the motivation for introducing 16-bit floating point operations and instructions in general purpose microprocessors.

### 1.1 16-bit floating point formats

16-bit floating formats have been defined for some DSP processors, but rarely used. Recently, a 16-bit floating point format has been introduced in the OpenEXP format [1] and in the Cg language [2] defined by NVIDIA. This format, called "half", 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 $\neq$ 0). Value 31 is reserved for representing infinite (Fraction = 0) and NaN (Fraction $\neq$ 0). For 0<E<31, the general equation for calculating the value in a floating point number is $(-1)^S \times (1.\text{fraction}) \times 2^{(\text{Exponent field}-15)}$. The range of the format extends from $2^{-24} = 6 \times 10^{-8}$ and $(2^{16}-2^5) = 65504$. In the remaining part of this paper, the 16-bit floating point format will be called *half* or F16. The "half" FP format is justified both by ILM, which developed the OpenEXP graphics format, and NVidia as a trade-off between precision, dynamic range and storage cost.

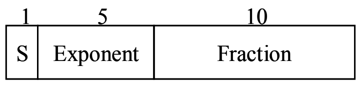

**Figure 1: NVIDIA "half" format**

### 1.2 Data format for image and media processing

Image processing generally need both integer and FP formats. For instance, vImage [3], 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. In many cases, using the "half" format would be a good trade-off: the precision and dynamic range of 32-bit FP numbers is not always needed and 16-bit FP computations are compatible with byte storage if efficient byte to/from half format is available. As input and output operands of floating point operations have the same format, the floating point computation has another potential advantage with SIMD instructions: it is far easier for the compiler to vectorize.

For media processing, the debate between integer and FP computing is also open. In [4], G. Kolly justifies "using Fixed-Point Instead of Floating Point for Better 3D Performance" in the Intel Graphics Performance Primitives library. Techniques for automatic floating-point to fixed-point conversions for DSP code generations have been presented [5]. On the other hand, people propose lightweight floating point arithmetic to enable FP signal processing applications in low-power mobile applications [6]. 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 the "half" format is adequate. If this debate first concerns the embedded applications, it is worth considering the interest of the "half" format for general purpose microprocessor as an extension for the SIMD instructions.

With the same SIMD register set of the currently available SIMD extensions, using SIMD F16 instructions doubles the number of parallel operations compared to SIMD 32-bit integer or FP instructions or provide the same number of parallel operations as the SIMD 16-bit integer instructions but with a larger dynamic range.

## 1.3 Integer, FP and compiler use of SIMD instructions

The choice between integer or FP data doesn't only depend on dynamic range and accuracy of computed values. As a matter of fact, using SIMD instructions and optimizing data accesses to improve cache performance are the two principal techniques that a programmer and/or a compiler can use to significantly reduce execution time. Compiler capabilities to vectorize is a key issue to improve performance. Presently, automatic compiler vectorization impact is generally disappointing even if compiler techniques improve steadily. One reason is that integer and FP instructions don't have the same features for compiler vectorization. Integer arithmetic operations have different input and output formats: adding two N-bit numbers provide an N+1-bit number while multiplying two N-bits numbers provide a 2N-bit number. This property leads to specific characteristics in the SIMD extensions. Saturating arithmetic (by definition) and 2's complement arithmetic discard carry outputs and cannot be used for arithmetic operations for which output dynamic range is greater than the input one. SIMD multiplications generally provide either the lower part or the higher part of the 2N-bit result for N-bit inputs. In IA-32 SIMD extension, the only instruction to get round the issue is the multiplication-add instruction (PMADDWD) that multiplies 4 or 8 signed short integers to deliver 32-bit intermediate results and horizontally adds two partial products to deliver 2 or 4 32-bit results (when using 64-bit or 128-bit SIMD registers). It is a specific exception with different input and output formats. A compiler can hardly find the opportunity to use such an instruction, which has obviously been defined for the dot product of 16-bit vectors widely used in signal processing. Such instructions are ad-hoc ones. The PSABDW instruction is the most famous example. Computing the sum of absolute values of differences between adjacent bytes, its only use is for motion estimation for which it has been defined. On the other hand, FP SIMD instructions have the same input and output formats and can be easily used by compilers. To optimize portable programs (without using assembly language or intrinsics), FP formats exhibit a significant advantage.

## 1.4 Organization of this presentation

In this paper, we only focus on the performance evaluation of the 16-bit FP operations and the vectorization issues. We don't discuss the precision and dynamic range issues for graphics and media applications, which are the programmer's responsibility. We will show that the proposed F16 format has performance close to the performance of the SIMD 16-bit integer version (whether this version has or not enough dynamic range) and more or less than two times the performance of the 32-bit FP version because the number of SIMD operations is double and of the smaller cache footprint. The second characteristic is the easy vectorization, which is similar to "float" or "double" compiler vectorization while different integer formats generally prevent any vectorization.

After this introduction motivating the introduction of 16-bit FP operations, section II presents the methodology that has been used including the benchmarks and the technique to "simulate" the execution of 16-bit FP operations on general purpose microprocessors and measure the execution time of the benchmarks. Section III presents the microarchitectural assumptions and the defined 16-bit operations on a Pentium 4 and on a PowerPC G5. Section IV presents the performance evaluation for the different benchmarks both for the Pentium 4 and the Power PC G5. Section V presents a preliminary evaluation of the chip area for the 16-bit FP operators compared to the actual chip area of the 64-bit FP operators used in general purpose microprocessors.

## 2. Methodology

In this section, we describe the benchmarks and the simulation methodology that have been used.

### 2.1. Description of benchmarks

For image processing, we first consider convolution operators: the horizontal and horizontal-vertical versions of Deriche filters [7] and a gradient: these filters operate on 2D arrays of pixels (*unsigned char*), do some computation by using integers and deliver byte (or integer) results. They are representative of spatial filters and have a relatively high computation to memory accesses ratio. The two next benchmarks are variants of scan algorithms. Given an associative operator o, and a vector v(x), the scan operation returns a vector w(x) such as w(x) = v (0) o v (1) o … o v(x). For example, the +scan returns for every pixel the 2D accumulation of the "previous" pixels of an image. The first benchmark implements a 2D accumulation and focuses on memory bandwidth limitation; the second one is based on a new segmentation algorithm proposed by Mérigot [8]. The classical image segmentation algorithm uses a quad tree computation based on the region's average and the variance (split and merge algorithm) [9]. Mérigot uses a +*scan operation to optimize both the speed and the quality of the segmentation. This +*scan operator accumulates the sum and the sum of the squares of the pixel area. These two last benchmarks have a lower computation to memory access ratio than our first set of benchmarks. For intermediate results, they need a dynamic range that implies floating point data.

For media processing, we consider the OpenGL data stream case study presented by Intel in [10]. The benchmark considers an OpenGL stream of triangles and computes the smallest box that bounds each triangle. We only considered the stream for which the triangle data are arranged in a tri-strip format (Figure 2) where the starting triangle is represented with all three vertices, but for each additional triangle that shares an edge with the triangle, only the third new vertex is stored (for N triangles, N+2 vertices are stored). Intel original assembly code has been converted into "intrinsics" code for the reference version to compare to the F16 version.

The code for all the benchmarks is provided in [11].

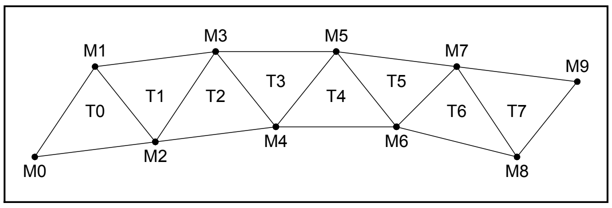

**Figure 2: Triangle in tri-strip format**

## 2.2. Simulation technique

Instead of using a software simulator, we have used a "hardware" one by measuring the execution time of the simulated instructions on actual hardware (Pentium 4 /PowerPC G5) for which the instructions are defined. The graphics and media benchmarks that we use 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: a) the cache accesses should be the same for the simulated and actual memory instructions. b) The data dependencies should be strictly enforced. c) 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. With a Pentium 4, the 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 drawback of this technique is that there is less degrees of freedom to choose latency and throughput values than with a software simulator. But one can easily argue that these values, corresponding to the actual values of the Pentium 4 instructions are more realistic of VLSI technical constraints than the values generally assumed in software simulators. By using the same latency and throughput figures for F16 instructions as the Pentium 4 single (and double) precision ones, we get an upper bound of the execution time for the F16 instructions. Using realistic lower latency figures for the F16 instructions give a good insight of possible improved results. The situation is the same for the Altivec instructions of the PowerPC.

## 2.3. Measures

For each benchmark, the execution time has been measured at least 10 times and we have taken the averaged value. For the Pentium 4, we have used a 2.4 GHz processor with 768 MB memory running Windows 2000. We have used the Intel C++ 8 compiler The QxW option generates specialized code for the Pentium 4. 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++). For the PowerPC, we have used 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.

For all the image benchmarks, the results are presented as the number of cycles per pixel (CPP) which is the execution time of the overall benchmark divided by the number of pixels. For the OpenGL stream case study, the results are presented as the number of cycles per triangle.. For all image benchmarks, we have followed the same approach. First, the naïve integer version has been transformed into an optimized integer SIMD version when this version can be implemented. Otherwise, it has been transformed into a float version for which each pixel is represented by a float. Then we have written another float version with Intel intrinsics which performance is equivalent or better than the compiler version. The SIMD integer or float versions with intrinsics have then been converted into a F16 version that operates on 16-bit FP data after conversion from the original byte array layout. This version is compared with the best optimized SIMD integer version and the best SIMD float version.

## 3. ISA and microarchitectural assumptions

As the reference processor, we considered the presently available version of the Pentium 4 As the MMX instructions are somewhat special by sharing the MMX registers with the x87 FP instructions, we only considered the XMM register set with 128-bit integer and SP FP instructions.

We assume the currently available 128-bit XMM register set. Extending to 256 bits would be a dramatic change on many aspects of the architecture: data cache access, doubling the number of functional units for all the SIMD operations, etc. 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. 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 looks more efficient. It implicitly unrolls two times any loop (the lower 8-byte operands first, then the higher ones). It avoids the alignment issues when dealing with byte accesses such as X[i][j] and X[i][j+1] which are easier to treat within the XMM registers.

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. 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 as each couple of bits controlling one out of four slots. Keeping an 8-bit immediate with eight slots would need a coding of the different operations on the eight slots.

**Table 1: 16-bit FP instructions for Pentium 4**

| INSTRUC TION | LATEN CY | 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 <=I8toF16 (Xmms low) |
| CBH2F16 | 4 | Xmmd<= I8toF16 (Xmms high) |
| CF162BL | 4 | Xmmd low<= F16toI8 (Xmms) |
| CF162BH | 4 | Xmmd high<= F16toI8(Xmms) |
| SHUFF16 | 4 | Xmmd <= shuffle (8 slots) Xmms |

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. *Short* from/to *half* conversions are also needed. Load and store packed instructions for *half* data are similar to the already packed integer instructions.

**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 |

We also considered the presently available G5 processor with Altivec extension and the instruction latencies [12] given in Table 2. As the Altivec extension is rather complete, we only need to add the vector F16 instructions and the byte to/from F16 conversion instructions. All the packing/unpacking and permutation instructions 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 are pessimistic, our G5 simulation are slightly optimistic.

## 4. Measured results

### 4.1. Deriche benchmarks

Both the horizontal (H) and horizontal-vertical (HV) versions of Deriche filters and the Deriche gradient can use one or two arrays. In the first case, the original array is replaced by the final array. The results presented in Table 3 for the Pentium 4 and Table 4 for the PowerPC G5 correspond to only one array.

Both tables give the best scalar and the best SIMD versions for 16-bit integer and 32-bit FP formats together with the F16 version. The crossed out values correspond to integer versions for which it is necessary to first check the coefficient values to avoid overflow (as detailed below for Deriche filters) or for which overflow can occur (gradient). F16 versions assume the instructions latencies previously defined in Tables 1 and 2.

The horizontal filter exhibits a loop-carried dependency. The best integer scalar version unrolls 3 times the inner loop and 2 times the outer loop. The best float scalar version unrolls 4 times the outer loop. There are two obstacles to manual vectorization with intrinsics: first the loop-carried dependency and second the dynamic range of the results when multiplying pixels by b0, a0 and a1 coefficients. The loop-carried dependency is suppressed by transposing the initial array before applying the filter and transposing back the resulting array. A 16 x 16 byte block transposition can be implemented with SIMD unpack instructions. To reduce the cache misses, we transpose a horizontal tile of 16 x 16 byte blocks into a vertical tile of 16 x 16 byte blocks, apply the filter on this vertical buffer and transpose back the resulting column into the initial horizontal tile. Vectorizing the multiplications is possible if coefficient values are known. If all coefficients values are less than 256, then multiplying a coefficient by a pixel value fits in a 16-bit value and using the Pentium 4 SIMD multiplication on 16-bit inputs and the lower 16-bit of the 32-bit output is OK. As it turns out that 2 coefficients are positive while the third one is negative, the inner loop can be implemented as a subtraction followed by an addition to avoid any overflow. If one coefficient is greater than 256, the expression computed in the inner loop can be transformed to become similar to the previous case. For instance, if 256 <b0 < 512, then b0 * X[i][j] >> 8 = X[i][j] + (b0 – 256) b0 * X[i][j] >> 8 and the multiplication of the pixel value by b0 – 256 is similar to the previously described case, and so on. We only give these details to outline that using SIMD integer instructions generally needs a detailed knowledge of the application including the dynamic range of the different coefficients. With 16-bit or 32-bit FP values, after the horizontal tile to vertical tile transposition, the inner loop easily vectorizes.

**Table 3: Execution time (CPP) on a 512x512 image on a Pentium 4**

| Benchmark | Scalar integer | Scalar F32 | SIMD integer | SIMD F32 | F16 |
|---|---|---|---|---|---|
| Deriche H | 35.5 | 30 | ~~9.1~~ | 19.7 | 9.3 |
| Deriche HV | 33.3 | 17 | ~~6.1~~ | 16.9 | 7 |
| Gradient | 17 | 11 | ~~4.1~~ | 6.8 | 3.5/5.3 |

**Table 4: Execution time (CPP) on a 512x512 image on a PowerPC G5**

| Benchmark | Scalar integer | Scalar F32 | SIMD integer | SIMD F32 | F16 |
|---|---|---|---|---|---|
| Deriche H | 27.7 | 10.3 | ~~4.2~~ | 13.8 | 5 |
| Deriche HV | 25 | 23 | ~~2.2~~ | 11.1 | 2.6 |
| Gradient | 17.6 | 73.6 | ~~2.4~~ | 5.7 | 2.5 |

For the two processors, the 16-bit integer or FP versions for filters and gradient outperform the 32-bit scalar integer and FP versions. The 16-bit integer and FP versions have similar performance, as memory accesses are the same and the conversions are similar (byte to short by unpacking with zero expansion, byte to F16 by actual conversion). The performance differences come from the difference in integer or F16 operation latencies.

For the Pentium, the F16 version is slightly slower than the 16-bit integer version as the additions (latency of 4 for F16 versus 2 for int16) are more frequent than the multiplications (latency of 6 for F16 versus 8 for int16).

However, the integer versions are specific and depend on the coefficient values when the F16 version is generic. The F16 versions exhibit a speed-up close to 2 versus the float versions for two reasons: there are 8 operations per instruction instead of 4, and storing byte instead of float reduces the cache footprint. There are two values for the F16 version in Table 3: the first one supposes one specific FABS16 instruction to compute the absolute value while the second one has not this instruction that doesn't exist in the IA-32 SIMD float instructions.

For the G5 processor, the speed-up between F16 and 32-bit FP versions ranges from 2.2 to 4.2. It comes from the number of operations per instruction, the cache footprint plus smaller latency values (F16 arithmetic operations have a latency of 5 instead of 8 and the latency of conversion instructions is 2, which is less than the latency of integer to float conversion for the G5 processor).

The best scalar version of the horizontal-vertical Deriche filter unrolls two times the inner loop. This filter has no dependency, but the same coefficient problems as the horizontal one. The only difference between the horizontal filter and the horizontal vertical execution times comes from the transposition execution times.

## 4.2. Scan benchmarks

We now consider the performance for the scan benchmarks that have a lower computation to memory access ratio than the previous benchmarks: the +scan has two additions for three memory accesses. Because of the accumulation, the accumulator must be large enough to avoid any overflow. 32-bit integers or floats are typically used for large images and 16-bit integer for small images. For the +*scan, the 32-bit integer format has not enough range: a $2^{34}$ range is needed for a 512x512 image and a $2^{36}$ one for a 1024x1024 image. Either the *half* or the *float* format is needed. The outputs can be stored in two separated matrices of floats, or interleaved in a single matrix. The F16 version was simulated from an integer *short* version and adding extra code for square computations and accumulations. The +scan (horizontal add) within a 128-bit register was implemented with 3 couples of addition/shift instructions.

The results for copy, +scan and +* scan are given in Table 5 for the Pentium 4 and in Table 6 for the PowerPC G5. The crossed out values correspond to versions for which the dynamic range is insufficient to get correct results.

It turns out that the +scan has performance that is close to the performance of the copy benchmark, which mean that the +scan is clearly memory-bounded. It is quite obvious in that situation that using FP formats cannot bring any advantage and that simply using integer formats is the best solution.

For the +* scan, the F16 version on the Pentium 4 has a speed-up slightly less than 2, which is easily explained by the same reasons as previously. The best 32-bit integer version, which has not enough range, would run at 12.1 CPP versus 7.5 for the F16 version. We outline that the SIMD instructions are not suitable to implement the + scan and +* scan operations as each result correspond to the sum (or sum of squares) of all the previous results: this typical recurrence situation prevents automatic vectorization. A significant number of SIMD instructions are needed to accumulate the previous sums into a SIMD register. With the Pentium 4, the SIMD version is always slower than the scalar one. Only the F16 version is faster than the scalar 32-bit FP version. The F16 version of +* scan has 1.3 times the execution time of the fastest copy with 16-bit integers, which is the lower bound. The overhead comes from the bookkeeping instructions and the significant number of copy instructions that results from the two-operand format of IA-32 ISA (when destination operand must be preserved, it should first be copied into another operand). Going further to improve performance means improving the memory bandwidth and/or reducing the instruction overhead for loop nests, as suggested by the MediaBreeze architecture [19].

**Table 5: Execution time (CPP) for the +*scan on a 512 x 512 image on a Pentium 4**

| Data formats | I8-I16 | I8-I16 | I8-F32 | F32-F32 | I8–F16 |
|---|---|---|---|---|---|
| Scalar Copy | 4.9 | 9.4 | 9.5 | 13.6 | |
| SIMD Copy | 4.7 | 9.2 | 9.4 | 12.5 | |
| Scalar +scan | ~~5.6~~ | 9.6 | 10 | 15.3 | |
| SIMD +scan | ~~7.2~~ | 10.5 | 10.6 | 17.5 | 7.8 |
| Scalar +*scan | | ~~18.8~~ | 19 | 17 | |
| SIMD +*scan | ~~9.9~~ | ~~18.7~~ | 18.7 | 21.3 | 10.5 |

**Table 6 : Execution time (CPP) of the +scan for a 512 x 512 image on a PowerPC G5**

| Data formats | I8-I16 | I8-I16 | I8-F32 | F32-F32 | I8–F16 |
|---|---|---|---|---|---|
| Scalar Copy | 5.5 | 9.3 | 62.4 | 10.4 | |
| SIMD Copy | 4.5 | 6.7 | 7 | 7.6 | |
| Scalar +scan | ~~24.3~~ | 10.4 | 95 | 18 | |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| SIMD +scan | ~~5.1~~ | 7 | 7.7 | 15 | 5.1 |
| Scalar +*scan |  | ~~17.7~~ | 26.7 | 18.5 |  |
| SIMD +*scan | ~~7.8~~ | ~~13~~ | 15 | 15.8 | 7.8 |

With the G5, F16 delivers a 1.5 speed-up versus the 32-bit FP format and 1.4 versus the 32-bit integer format for the +scan. For the +*scan, the speed-up is 1.9 versus the 32-bit FP format. Opposed to the Pentium 4, the SIMD versions are better than the scalar ones. As the Altivec extension is far more complete than the SSE/SSE2 extension, it is far easier to manually vectorize the scan benchmarks, which leads to better performance.

### 4.3. OpenGL benchmark

For the OpenGL stream case study, the reference version has vertices with float coordinates and stores each bounded box coordinates as 3 10-bit values packed into a 32-bit integer. The Pentium 4 execution time of the function is 195 cycles per triangle, which correspond to CPI=7. The F16 version has vertices with F16 coordinates and store each bounded-box coordinates as a 64 word (3 x F16 + padding). The execution time is 107.5 cycles per triangle and the CPI is 5. The speed-up is 1.8 and results from the eight parallel operations per SIMD instruction.

The PowerPC G5 execution time of the original version is 21.5 cycles per triangle versus 10.5 cycles per triangle for the F16 version. The speed-up (2.0) is close to the Pentium speed-up, but the G5 uses 9 times less cycles because it uses half the number of instructions of the Pentium to transform the initial input data structure into a structure suitable for SIMD operations and because the CPI is better (For the initial 32-bit FP version, the G5 CPI equals 1.6 versus 7 for the Pentium 4). Although, the overall result is very different because of the difference in the ISA extensions, the relative performance between F16 and 32-bit FP versions are very similar.

## 6. Chip area evaluation of F16 functional units

Only a VLSI implementation in the framework of the actual microprocessor (Pentium 4 or G5) could provide significant figures to estimate the area, power and timing features of the 16-bit floating point functional units. To get a rough preliminary approximation, we used VHDL models of floating point operators and a 0.18 µm cell-based library from ST (HCMOS8D technology). The same approach has been used by Talla et al [13] to evaluate the hardware cost of the MediaBreeze architecture. The VHDL models have been developed by J. Detrey and F. De Dinechin [14]: they include non pipelined and pipelined versions for the addition, the multiplication, the division and the square root operation. The adder uses a close path when the exponent values are close and a far path when their difference is large. The divider uses a radix-4 SRT algorithm while the square root operator uses a radix-2 SRT algorithm. In Table 7, we show the chip area of the different operators that is estimated by the Cadence 4.4.3 synthesis tool before placement and routing. For eight such 16-bit FP functional units, the chip area would be less than 11% than the chip area for the four 64-bit FP functional units that are implemented in the general purpose microprocessors (we assumed that the same FP units are used both for single and double precision FP numbers, as the corresponding instructions have the same latency). In our evaluation, the 16-bit FP adder is rather large compared to the other 16-bit operators. The dual path approach that gives the best results for 64-bit addition is probably unnecessary for 16-bit addition. A smaller 16-bit FP adder with a straightforward approach could be used as mentioned in [6].

**Table 7: Estimation of chip area ($mm^2$) for non pipelined FP operators in a 0.18µm CMOS technology**

| Op. | Add. | Mul. | Div. | Sqrt | Overall |
|---|---|---|---|---|---|
| 16-bit | 0.019 | 0.016 | 0.047 | 0.027 | 0,110 |
| 64-bit | 0.097 | 0.276 | 1.008 | 0.679 | 2,059 |
| Ratio | 19.90% | 5.91% | 4.64% | 4.04% | 5.33% |

## 7. Concluding remarks

For graphics applications on CPUs and GPUs, there is a common trade-off between precision and dynamic range on one hand and cost of storage on the other hand. Many graphics applications have better performance with floating point formats than with integer ones. One reason is that using FP formats make easier compiler or manual vectorization as FP operations have the same input and output formats. However, the single precision floating point format uses four times more memory as a byte format. When it provides enough precision and dynamic range, the 16-bit floating point format defined by ILM for the OpenEXP format and NVIDIA seems a good trade-off.

In this paper, we have considered a limited set of 16-bit FP operations and a set of conversion instructions between byte and 16-bit FP format for two common general purpose microprocessors. We have measured the execution time of different versions of typical graphics benchmarks (Deriche filters, Gradient, scan) with integer, float and "half" formats. For this last format, we have simulated the "half" instructions by using actual Pentium 4 or PowerPC G5 instructions having the same latency and throughput as the simulated instructions. For the compute bounded benchmarks that can be vectorized, the speed-up is greater or less than 2 (but close to 2) compared to the best "float" version. For the +*scan that needs floating point formats, the speed-up is greater than 2.

A very preliminary evaluation of the chip area shows that for eight 16-bit FP functional units, the chip area should not exceed 11% of the chip area currently devoted to the FP functional units in a Pentium 4 or a G5.

This work will be completed by considering a more significant lot of graphics and media applications. For graphics or media applications that are compute-bounded, F16 speed-up is close to 2 versus 32-bit FP versions. For graphics applications with byte stored pixels, the F16 versions have performance close to 16-bit integer versions, but have two significant advantages: compiler vectorization is greatly facilitated and the dynamic range is larger for intermediate results. Even when significant differences between the two considered SIMD extensions (SSE/SSE2 and Altivec) can lead to significant differences in performance figures (clock cycles per pixel), the performance gain between F16 and 32-bit FP versions are consistent for the two processors for all the benchmarks.

## 8. Acknowlegdements

A. Dupret and J.O. Klein, from the IEF laboratory (University Paris Sud) have provided us the chip area evaluations using the Cadence synthesis tool with the 0.18 µm ST technology. Their help was greatly appreciated. F. Tourant, from Apple France Edu, provided us for a couple of weeks the Dual 2GHz PowerPC G5 which allows starting experiments on G5.

## 9. References

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

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

[3] Apple, "Introduction to vImage", http://developer.apple.com/documentation/Performance/Conceptual/vImage/

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

[5] 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)

[6] 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

[7] R. Deriche. "Using Canny's criteria to derive a recursively implemented optimal edge detector". The International Journal of Computer Vision, 1(2):167-187, May 1987.

[8] A. Mérigot, Revisiting image splitting, 12th International Conference on Image Analysing and Processing, pp 314-319, ICIAP 2003.

[9] S. Horowitz, T. Pavlidis. Picture segmentation by a tree traversal algorithm. Journal of the ACM, 22:368-388, 1976.

[10] A. Kumar, "SSE2 Optimization – OpenGL Data Stream Case Study", Intel application notes, http://www.intel.com/cd/ids/developer/asmo-na/eng/segments/games/resources/graphics/19224.htm

[11] Sample code for the benchmarks available: http://www.lri.fr/~de/F16/codetsi

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

[13] D. Talla, L.K.John and D. Burger, "Bottlenecks in Multimedia processing with SIMD Style Extensions and Architectural Enhancements", in IEEE Transactions on Computers, Vol 52, N° 8, August 2003, pp 1015-1031.

[14] J. Detrey and F. De Dinechin, "A VHDL Library of Parametrisable Floating Point and LSN Operators for FPGA", http//www.ens-lyon.fr/~jdetrey/FPLibrary