Customizing CPU instructions for embedded vision systems

Stéphane Piskorski
LRI
Université Paris Sud
stephane.piskorski@lri.fr

Lionel Lacassagne
IEF
Université Paris Sud
lacas@ief.u-psud.fr

Samir Bouaziz
IEF
Université Paris Sud
samir.bouaziz@ief.u-psud.fr

Daniel Etiemble
LRI
Université Paris Sud
de@lri.fr

***Abstract*— This paper presents the customization of two processors: the Altera NIOS2 and the Tensilica Xtensa, for fundamental algorithms in embedded vision systems: the salient point extraction and the optical flow computation. Both can be used for image stabilization, for drones and autonomous robots. Using 16-bit floating-point instructions, the architecture optimization is done in terms of accuracy, speed and power consumption. A comparison with a PowerPC Altivec is also done.**

I. Introduction

Embedded vision systems and SoCs are great scientific challenges. Designers have to integrate more and more CPU power to run still hungrier algorithms while limiting power consumption. Adding SIMD instructions within a processor is a good way to tackle the problem of power consumption.

For vision and media processing, the debate between integer and FP computing is still open. On one hand, people argue about using fixed-point computation instead of FP computation [7]. Techniques for automatic FP to fixed-point conversions for DSP code generations have been presented [11]. On the other hand, people propose lightweight FP arithmetic to enable FP signal processing in low-power mobile applications [4].

This paper follows previous ones dealing with optimized 16-bit SIMD floating-point FP instructions and presents a consistent Algorithm-Architecture Adequation where both architectures and algorithms are customized to provide high performance embedded systems. Our presentation uses two examples: the Harris' Points of Interest (PoI) and the Horn and Shunk's optical flow computation.

The specificity of these algorithms is that computations cannot be done with only 8-bit integers as they lack the needed accuracy and dynamic range. A fine study of variable width FP format - 16-bit floats and smaller ones - and their impact on algorithm accuracy is done. As optical flow is an iterative algorithm, mastering the error accumulation and propagation, from one iteration to the next one, is very important.

We then present benchmark results for the PowerPC using Altivec, NIOS2 and Xtensa processors, which provide accurate information on the implementation of embedded systems. We finally focus on the power consumption and processor area issues.

II. F16 floating-point formats

Some years ago, a 16-bit FP format called "half" has been introduced in the OpenEXR format [13] and in the Cg language [10] defined by NVIDIA. It is currently being used in some NVIDIA GPUs. It is justified by ILM, which developed the OpenEXR graphics format, as a response to the demand for higher color fidelity in the visual effect industry.

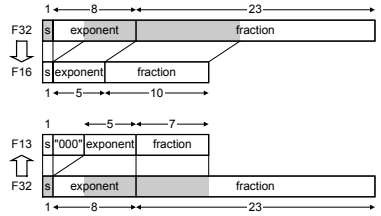

Fig. 1. F_{32}, F_{16} and F_{13} floating-point numbers

In the remaining part of this paper, this 16-bit FP format will be called "half" or F_{16}, the IEEE-754 32-bit single-precision FP format will be called F_{32}, and the 8-bit, 16-bit and 32-bit integers will be called I_8, I_{16} and I_{32} formats. The format is presented in figure 1. A number is interpreted exactly as in the other IEEE FP formats. The range of the format extends from 6×10^{-5} to $2^{16} - 2^5 = 65504$.

To balance the embedded hardware and the algorithms accuracy, we also had a look at an even lighter FP coding: F_{13}. It has a 5-bit exponent and a 7-bit mantissa (figure 1). It is stored into 16-bit word but is called F_{13} because, the 3 msb of the exponent are *useless* (as we know the computation dynamic range for these algorithms. F_{13} could be interpreted as the two "Most Significant Bytes" of F_{32}.

III. Summary of previous results

In [3][8], we have considered the speedup between versions using SIMD 16-bit FP instructions (called F_{16}) and versions using 32-bit FP instructions (called F_{32}). Speedups have been measured on general purpose processors: Pentium 4 and PowerPC G4. The following benchmarks have been used: Deriche filters), spline zoom [15], JPEG [9] and wavelet transform [14].

For all benchmarks, SIMD F_{16} provides at least a speedup of 2 compared to SIMD F_{32}. Speedup can even reach $\times 4$ because of the smaller F_{16} cache footprints: when F_{16} data fit in the cache, but not F_{32} ones. From a qualitative point of view, F_{16} versions of the algorithms are quite as good as F_{32} and better than the usual I_{32} version. For example JPEG F_{16} results in a 5 dB higher PSNR than I_{32}, and matches the F_{32} PSNR.

For all benchmarks, we also tested several rounding modes: truncation or standard rounding mode, with or without denormals. Analysis shows that truncating without denormals is very

close to rounding , with or without denormals. This is actually the ideal situation for embedded hardware. In the article, we only consider F_{16} with truncation and without denormals.

IV. Algorithms presentation

A. Point of Interest

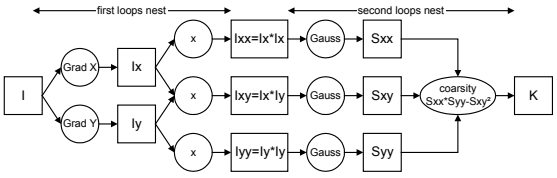

Fig. 2. Harris PoI algorithm

We used the Harris' corner edge detector (figure 2): $K = S_{xx} \times S_{yy} - S_{xy} \times S_{xy}$, where S_{xx}, S_{xy} and S_{yy} are the smoothed squared first derivatives of the image I.

B. Optical Flow

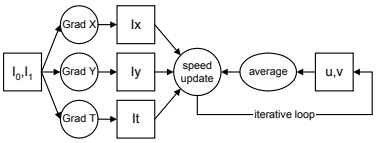

Fig. 3. Horn & Shunk algorithm

The Horn and Shunk optical flow computation is based on an iterative framework (fig 3), whose steps are: spatio-temporal first derivatives I_x, I_y and I_t average speed $(\bar{u}, \bar{v})$, and speed update (u, v).

Here, the main problem is the accuracy through all computations and especially during the division in the speed update stage. Because this division is different in every point, it cannot be easily tabulated and put into a LUT.

$$u = \bar{u} - I_x \times \frac{I_x \times \bar{u} + I_y \times \bar{v} + I_t}{\alpha^2 + I_x^2 + I_y^2} \quad (1)$$

$$v = \bar{v} - I_y \times \frac{I_x \times \bar{u} + I_y \times \bar{v} + I_t}{\alpha^2 + I_x^2 + I_y^2} \quad (2)$$

V. Architectures presentation

A. PowerPC G4 Altivec

To allow a *fair* comparison between the two hardware targets, we also implemented algorithms on a PowerPC G4, to get a reference in terms of speed and power consumption. The PowerPC G4 was the first processor to implement the 128-bit SIMD multimedia extension called Altivec [2] . It is a competitive RISC processor: at 1 GHz, the PowerPC release 7457 has a power consumption of about 10 watts.

B. Nios II

The algorithms have been implemented into an Altera Stratix 2 FPGA running at 150 MHz (when configured with a stand-alone NIOS 2 processor), with 60000 cells.

With NIOS II processors, the customized instructions can be implemented as combinational (1-cycle) or multicycle instructions. Multicycle operations are not pipelined, which means that the pipeline is stalled until the end of the operation when the *done* signal becomes active. Finding the optimal trade-off between the operation latency (number of cycles) and the maximal CPU clock frequency is thus important. This is why we have tested 1-cycle and 2-cycle versions of the 16-bit FP arithmetic instructions.

C. Xtensa

Some benchmarks were tested using an Xtensa LX customizable processor from Tensilica. to compare F_{16} and F_{32}. The Xtensa LX core is a 32-bit RISC processor, targeting SoC designs, with a configurable instruction set. A proprietary Verilog-like language called TIE can be used to add custom instructions, register files or I/O ports. FP instructions with custom formats could thus be easily implemented and compared with each other. They were also compared to the native Xtensa LX FPU (manipulating data refered to as *floats* hereafter), which provides IEEE compliant FP computation instructions with a 4-cycle latency.

The Xtensa processor simulated was configured to use a 128 bit-wide memory interface, a 32-KB 4-way associative cache memory, a 5-stage pipeline. A FPU was incorporated, which makes use of a 16-entry register file. All speed and area estimates assume an ASIC implementation with a 90nm technology. A hardware loop counter is also provided, which avoids branch mispredictions since the image size is known at compile time in our algorithms.

Custom FP instructions were implemented using TIE, with mantissa and exponent widths as parameters: a F_{16} version and a F_{32}, both without subnormals and with truncated calculations. Table V-C lists the main arithmetic functions that were implemented along with their estimated area (given by Xtensa development tools). Instructions decoding, load/store instructions and other miscellaneous logic require about 25% additional area. The custom FP instructions use dedicated custom register files, also implemented by using the TIE language.

Block	F_{16}	F_{32}	F_{16} SIMD
`Add`	1384	2666	11470
`Mul`	1695	4430	13480
$*2^n$	158	275	1197
$/2^n$	124	221	931
`Byte->`F_{xx}	273	493	1885
F_{xx}`->Byte`	1469	2068	11476
`Reg. file`	5306	9481	34716
`Total`	10409	19634	75155

TABLE I

Main TIE logic blocks and estimated sizes (gates)

VI. Qualitative approach

For the optical flow computation we have tested the following configurations:

- F_{64}: the reference schema that can be seen as the *high precision* Matlab computation,
- F_{32}: the usual FP implementation,
- F_{16}: our proposition,
- F_{13}: our *highly* embedded proposition,
- I_{16}: the integer competitor of F_{16},
- I_{32}: the integer competitor of F_{32}.

The optical flow algorithm uses division operators (first derivative and speed average). The I_{16} implementation keeps them in place but the I_{32} implementation postpones them at the final step, which requires more bits to hold the temporary results. The I_{16} and I_{32} versions use radix-8 fixed-point computations, to provide an 8-bit fractional part and an 8-bit integer part, for the 16-bit version. The speed update equation remains unchanged in I_{16} but is modified for I_{32} version as follows:

$$u = \frac{1}{12}\left[\bar{u} - I_x \times \frac{I_x\bar{u} + I_y\bar{v} + 12 \times 2^8 \times I_t}{16\alpha^2 + I_x^2 + I_y^2}\right] \quad (3)$$

. To optimize speed on PowerPC, F_{32} are used for accurate computations, but 16-bit storage is performed to reduce cache footprints. Conversions or truncations occur after computations:

- $F_{32}\rightarrow I_{16}$: F_{32} computations are stored into I_{16},
- $F_{32}\rightarrow F_{16}$: F_{32} computations are stored into F_{16},
- $F_{32}\rightarrow F_{13}$: F_{32} computations are stored into F_{13}.

For I_{16} storage, a radix 8 is used. For F_{16} storage, both mantissa and exponent are truncated (while the bias changes from -127 to -15). F_{13} storage simply requires the mantissa truncation. To estimate the impact of number coding, two

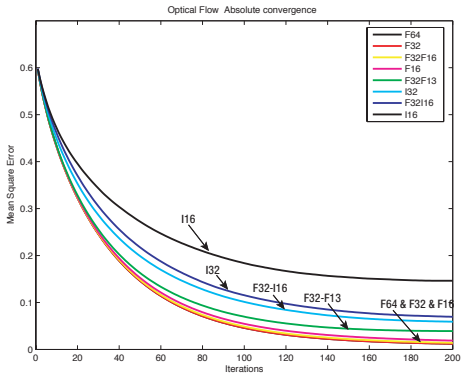

Fig. 4. absolute convergence of optical flow versions

images were created, the second one with an offset of $(3, 1)$ pixel. The images were then downscaled by a factor 4, leading to an apparent motion of $(3/4, 1/4)$ pixel. Figure 4 represents the mean square error between the estimated flow and the real flow.

We can see that (legend indicates curves in reverse order) I_{16} and I_{32} converge to a bigger error than F_{16}. We can notice there is no difference between F_{32} and F_{64}, and that F_{16} is very close to F_{32}. This is an evidence of F_{16} interest in image processing algorithms.

We can also notice that, obviously, $F_{32}\rightarrow F_{16}$ are better than F_{16}, since computations are done with a greater accuracy. But what is also very important is that, first, $F_{32}\rightarrow I_{16}$ is close to I_{32} with a twice smaller memory footprint, but leads to a bigger error than $F_{32}\rightarrow F_{13}$ which is between F_{16} and I_{32}. That enforces the use of F_{13}, for both PowerPC and the hardware implementation. Now focusing on the ten to thirty

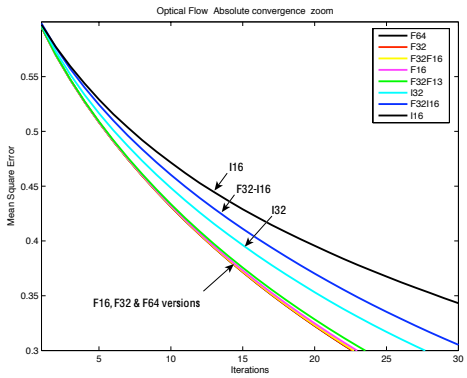

Fig. 5. first iterations of optical flow convergences

first iterations of the algorithm (5) instead of hundreds of iterations which is a bit irrelevant for real-time execution on embedded systems, we can see figure 4, that all the FP versions overlap: F_{16} and F_{13} versions are as accurate as F_{32} and F_{64} versions.

For PoI, the dynamic range is the main issue. Since the input data is an 8-bit image, the output K could be as large as 32 bits. Since IEEE-754 F_{16} maximum value is approximately 2^{16}, there are two possibilities to fix the problem:

- by normalizing input interval to $[0 : 1]$ with a division by 256 that could take place during the conversion from I_8 to F_{16},
- by replacing -15 bias by 0 or by a positive value.

Since computations are performed with truncation we have tested two configurations: division by 16 and by 256. Results are very close. The accuracy was estimated through the extraction of PoI on the whole Movi *house* sequence [12]. We provide for *variable width mantissa*, ranging from 4 bits to 10 bits, the minimum, the average and the maximum PSNR between custom F_{16} and F_{32} over the 120 images of the sequence.

float	mantissa size	min	avg	max
F_{16}	10	78.9	82.0	85.5
F_{15}	9	72.8	76.0	79.9
F_{14}	8	66.0	68.8	72.2
F_{13}	7	58.9	61.8	64.6
F_{12}	6	52.3	55.4	58.3
F_{11}	5	46.9	50.0	53.0
F_{10}	4	42.3	44.5	45.4

TABLE II
PSNR between variable width F_{16} and F_{32} for PoI

Again, F_{16} are very close to F_{32}, and the F_{13} version with a 7-bit mantissa performs well. We can also note that, because

of the nature of the algorithm - "multiplication of horizontal gradient by the vertical gradient" - the algorithm is very robust to a short number coding.

VII. Quantitative approach

In this section, we provide the execution time results for the three architectures. The metric used is *cpp* ($cpp = \frac{t \times Freq}{n^2}$), which is the number of clock cycles per pixel. We believe that *cpp* is more useful than *cpi* to compare different architectures, since it can also help to compare the complexity-per-point of different algorithms. Moreover *cpp* is also useful to detect cache misses from one size of image to another.

A. Quantitative approach with PowerPC

For the PowerPC implementation, we started from a full F_{32} version where the image is coded with F_{32} pixel instead of I_8: such a version provides a rough idea of performance and is also easy to develop. Then we switched to I_8: the internal loop had to be unrolled four times, since in I_8 vector pixels are packed by 16 and in F_{32} vector, pixels are packed by 4. Such a scheme provides additional ways to optimize the code:

- software pipelining: to reduce the data dependency stress within a loop iteration (for PoI, it prevents from storing first derivatives into memory) and to reduce the total number of cache accesses,
- data interlacing: to reduce the number of active references in the cache,
- data packing: F_{32} points are stored into F_{13} to divide by 2 the cache footprint.

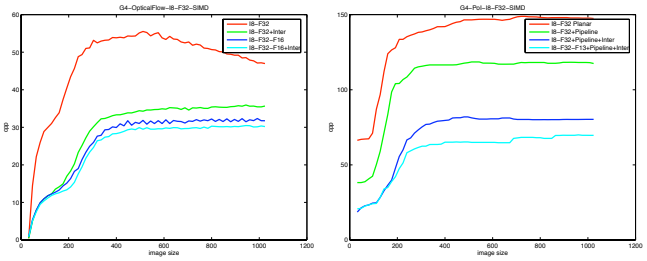

Fig. 6. Optical flow & PoI on powerPC Altivec

We provide 4 versions: $I_8 \rightarrow F_{32}$ and $I_8 \rightarrow F_{32} \rightarrow F_{13}$, with and without data interlacing. For F_{13}, we get a very dense code where data interlacing has a small impact, compared to $I_8 \rightarrow F_{32}$ version. At the end, we achieve a $\times 2$ speedup from the basic SIMD version.

B. Quantitative approach with Xtensa

For the Xtensa LX implementation, the image processing benchmarks (PoI and Optical Flow) were written in C. A first version used integer values (referred to as *int* hereafter). A second one used native *floats*; data had the C language *float* type, and calculations were made using the Xtensa LX's FPU and the associated 16-entry-wide register file. Two other versions, using F_{16} and F_{32} were written with C intrinsics to call our custom FP instructions. Since the number of cycles needed to perform computations using our custom instructions depends on the desired operating frequency, simulations were made with several assumptions: an ideal 1 or 2-cycle latency case and a worst case with custom instructions having the same 4-cycle latencies as the native ones.

The results also depend on the latency of the memory connected to the system. Simulations were first run with the hypothesis of a latency-free memory, which is totally unrealistic but shows the speedup directly brought by the faster-computed instructions. They were then run with memory accesses including the cache behaviour. Eventually, a 16-bit SIMD version (8 points per data vector, because of the 128-bit wide memory interface) was tested to get an idea of what performance could be achieved when using F_{16} at their full potential.

Figures 7 and 8 present the number of cycles needed to compute each algorithm. Table VII-B shows the number of cycles spent processing data cache misses. Finally table VII-B shows number of cycles spent waiting because of data dependencies between instructions in the pipeline.

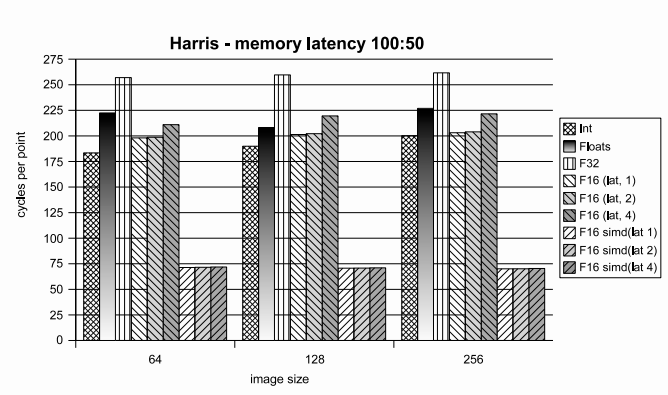

Fig. 7. PoI's cpp. - data cache and memory latencies simulated

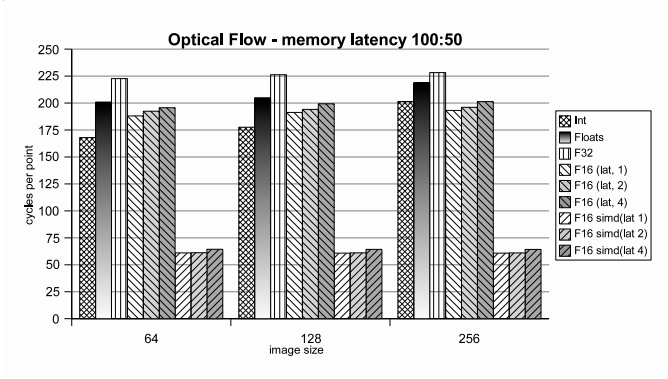

Fig. 8. Optical Flow algorithm - data cache and memory latencies simulated

The simulations run with a zero latency memory shows that F_{16} computation does not bring a significant speedup by itself, compared to its 32-bit equivalent (results are even contrary to what expected when compared to native *floats* since the compiler is able to optimize them in a better way).

image size	64	128	256
Optical Flow			
`int,floats,`F_{32}	0.62	0.63	0.63
F_{16},F_{16SIMD}	0.38	0.38	0.38
PoI			
`int,floats`	0.81	0.81	0.81
F_{32}	0.82	0.82	0.82
F_{16}	0.44	0.44	0.44
F_{16SIMD}	0.45	0.45	0.44

TABLE III

XTENSA - DATA CACHE MISSES PER POINT

image size	64	128	256
PoI			
`floats`	19.71	20.35	20.67
F_{32}	14.17	14.58	14.79
`int`,F_{16},F_{16SIMD}	1.00	1.00	1.00

TABLE IV

XTENSA - INSTRUCTION INTERLOCKS PER POINT

However, simulating the cache behaviour highlights the benefit of F_{16}: their twice smaller size leads to twice less cache misses, which is a real advantage as the processor/memory speed gap increases. Furthermore, this also doubles the number of operations per instruction for constant SIMD vector size and hardware cost, since HDL operators reduce in size in the same proportions.

Besides table VII-B shows the reduction of the number of pipeline stalls due to data dependencies. The reduced number of cycles needed to compute F_{16} helps the compiler to reorder the instructions for a better pipeline efficiency.

C. Quantitative approach with NIOS2

Two kinds of synthesis have been performed: structural version (the architecture is only described with structural elements) and functional version (structural description but with functional comparators). The implementation of F_{13} instead of F_{16} provides a higher clock rate with the same area (number of cells) in the case of retiming for the structural version. The speedup due to retiming for F_{16} and F_{13} is $[\times 2.3 : \times 2.8]$ while the area increase is only $[\times 1.5 : \times 1.6]$. Finally F_{13} combined with retiming generates a overall speedup of $\times 3$ with only an area increase of $\times 1.5$. For the NIOS2, 3 versions of the operators were designed:

- F_{16}-2: 2-cycle latency multiplication and addition.
- F_{16}-1.5: a 1-cycle multiplication and a 2-cycle addition.
- F_{16}-1: 1-cycle multiplication and addition.

For these three versions we provide the maximum frequency and *cpp* and compare the SIMD2 F_{16} to scalar I_{32} version.

We can notice that the *cpp* are better for sizes which are not powers of two (200 and 260). This comes from the NIOS2 direct-mapped data cache. A smart implementation should use images with padding at the end of each line to get "unaligned" starts of lines.

		image size				
design	Freq	64	128	200	256	260
scalar	150	300.5	310.2	219.3	315.6	227.4
F_{16}-2	130	211.3	229.7	204.0	239.4	208.0
F_{16}-1.5	92	179.0	194.5	167.8	202.8	171.4
F_{16}-1	80	136.5	148.2	120.0	154.6	123.0

TABLE V

cpp POI ON NIOS2

VIII. EMBEDDED APPROACH

To estimate the embedded capability of each architecture, we compute the energy consumption which is based on a rough estimation of the power consumption multiplied by the execution time. We do not try to get accurate results but trends to compare the three architectures.

image size	64	128	256	512
Optical Flow				
cpp	7.5	11.6	19.6	28
t(ms)	0.03	0.19	1.24	7.34
energy (mJ)	0.30	1.9	12.8	73
PoI				
cpp	22.6	29	40	53.2
t(ms)	0.09	0.48	2.66	13.9
energy (mJ)	0.92	4.75	26.6	139

TABLE VI

EMBEDDED PERFORMANCE OF POWERPC G4

A rough estimate of the Xtensa LX processor power consumption was made by assuming it proportional to the chip area computed by Xtensa tools, since our academic license did not permit a better estimation. This leads to about 250mW when adding scalar F_{16} instructions and 350mW when adding SIMD F_{16} instructions, with the same basic assumptions (core, caches, process geometry, etc.) at 500MHz. This gives an idea of the reachable electric consumption as given in table VII, where results are given for a realistic average case (10:5:5:5 memory latencies, 2-cycle F_{16} instructions) instead of the worst case previously described in figures 7 and 8.

To estimate the power consumption of the three F_{16} versions, we used PowerPlay from Quartus, with the pessimistic assumption of 25% of active signals at each cycle. We can see that all F_{16} version have a smaller or equal consumption than I_{32}, with also a smaller frequency and a smaller *cpp*.

Due to small parallelism (2 operations per instruction), the speedup of SIMD F_{16} compared to scalar I_{32} is small, but more importantly, the 1-cycle F_{16} version frequency is half the scalar version, making easier to interface the FPGA with the memory. We can also notice that with such 1-cycle operators, there is no more data dependencies in the code, decreasing also the impact of the compiler quality.

IX. CONCLUDING REMARKS

We have continued the evaluation of 16-bit FP operators and instructions. While previous results focused on the impact of 16-bit SIMD on general purpose processors and automatic

image size	64	128	256	512
Optical Flow F_{16}				
cpp	157.8	160.0	162.1	n.a.
t(ms)	1.3	5.2	21.2	n.a.
energy (mJ)	0.3	1.3	5.3	n.a.
PoI F_{16}				
cpp	158.2	162.4	164.3	n.a.
t(ms)	1.3	5.3	21.5	n.a.
energy (mJ)	0.4	1.4	5.4	n.a.
Optical Flow F_{16} SIMD				
cpp	26.6	26.8	27	n.a.
t(ms)	0.2	0.9	3.5	n.a.
energy (mJ)	0.1	0.3	1.2	n.a.
PoI F_{16} SIMD				
cpp	30.3	30.2	30.1	n.a.
t(ms)	0.25	1.0	4.0	n.a.
energy (mJ)	0.1	0.4	1.4	n.a.

TABLE VII

ROUGH ESTIMATES OF XTENSA LX PERFORMANCES

design	freq. (MHz)	Power (mW)
scalar	150	1303
F_{16}-2	130	1307
F_{16}-1.5	92	1145
F_{16}-1	80	1123

TABLE VIII

MAX FREQUENCY AND POWER CONSUMPTION OF NIOS2

code vectorization, we have extended the evaluation of short and customizable FP formats to embedded vision systems and SoCs, through the implementation of image stabilizing algorithms where accuracy and dynamic range problems make F_{16} assert themselves. We got promising results on each customized architecture.

Xtensa achieves the same level of performance, but with a $\times 10$ smaller consumption than Altivec. NIOS 2 efficiency is not as high as PowerPC or Xtensa: but this soft core implementation of F_{16} can provide low-end embedded systems with FP capabilities and flexibility. Its main advantage is to cut development time compared to classical FPGA design, and also to be ready to integrate into embedded vision systems. We can also note that, once optimized, the widely spread PowerPC G4 remains a good challenger for embedded applications.

We are currently developing an Altivec compatible ISA for Xtensa, to accelerate the port of existing applications and to facilitate comparisons between different architectures without re-developing high level code and data transformations.

		image size				
design	freq.	64	128	200	256	260
scalar	150	2.61	2.69	1.90	2.74	1.98
F_{16}-2	130	2.12	2.31	2.05	2.41	2.09
F_{16}-1.5	92	2.23	2.42	2.09	2.53	2.14
F_{16}-1	80	1.93	2.08	1.68	2.17	1.73

TABLE IX

NIOS2 ENERGY CONSUMPTION FOR POI ($\mu J/pixel$)

		image size		
archi.	freq.	64	128	256
NIOS II	80	7.848	34.085	113.16
PowerPC G4	1000	0.920	4.750	26.60
Xtensa	500	0.087	0.346	1.38

TABLE X

POI ENERGY CONSUMPTION (MJ) ARCHITECTURE COMPARISON

Future works will consider more robust and sophisticated embedded motion compensation algorithms and techniques for the automatic exploration of configurations according to speed, power consumption and hardware resources metrics.

REFERENCES

[1] R.P. Brent, H.T. Kung, "A Regular Layout for Parallel Adders", IEEE Transacton of computer, vol 31,3, pp 260-264, 1982.

[2] K. Diefendorff, P.K. Dubeyn R. Hochsprung, H. Scales, "Altivec extension to PowerPC accelerates media processing", IEEE Micro, March-April 2000.

[3] 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.

[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] J. Harris, M. Stephens, "A combined corner and edge detector", 4th ALVEY Vision Conference, pages 147-151,1998.

[6] P. M. Kogge, H. Stone, "A Parallel Algorithm for the efficient solution of a general class of recurrence equations", IEEE Transactions on computers, Vol 22, 8, pp786-793, 1973.

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

[8] L. Lacassagne, D. Etiemble, "16-bit floating-point instructions for embedded multimedia applications", in IEEE CAMP 2005, Palermo, Italy.

[9] 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.

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

[11] 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).

[12] IRISA Movi group: http://www.irisa.fr/texmex/base_images/index.html

[13] OpenEXR, http://www.openexr.org/details.html

[14] A. Said and 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.

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

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