Efficient Altivec Customization for Embedded Systems

Tarik Saidani, Stéphane Piskorski, Lionel Lacassagne and Samir Bouaziz

***Abstract*— General purpose processors have SIMD units that are dedicated to multimedia and vision applications. This paper focuses on optimization techniques like instruction specialization and a new one called *RISCization*, allowing an efficient implementation of these same operators on high performance vision SOCs. Execution time and power consumption can be both reduced by 50% when compared to a standard implementation.**

***Index Terms*— SIMD, Altivec, vectorization, embedded systems, FPGA, processor customization, high performance image processing, power consumption reduction.**

I. INTRODUCTION

SIMD multimedia instruction set extensions are present in all general purpose processors (GPP): SEE and SSE2+ for floating-point and integer computing inside Intel and AMD processors, Altivec inside PowerPC architectures (G4 and G5) [3]. Because of the nature of the involved algorithms, these SIMD instructions are very efficient for vision and multimedia applications [2]. Adding an existing SIMD instruction set architecture (ISA) inside an FPGA presents several advantages. First, code porting time is very short since the same code can be used both on the considered GPP and the so-modified FPGA. Second, not modifying the code prevents extra bugs, thus reducing debugging time. Third, sophisticated IDEs available on PC can be used to make the code evolve. Using the same software and hardware architecture will provide an efficient way to get an embedded version of an application. However, the main drawback of this kind of approach is that SIMD ISAs are quite big to fit within an FPGA. When power consumption is an issue, PowerPC processors have asserted themselves for embedded applications and especially PowerPC G4 with Altivec instructions for image and signal processing. One way to improve power consumption is to integrate only the most useful instructions into an FPGA. A second way is to specialize these selected instructions since the Altivec ISA is very versatile. This article is a first try to determine what the useful SIMD instructions for a SoC are, and how to efficiently implement them. The answer comes in three steps:

- How efficient are SIMD instructions ?
- Should complex instructions be decomposed into more simple ones ?
- What is the gain provided by restricting capabilities and removing useless features ?

The first paragraph quickly presents basic algorithms that are the foundation of a lot of signal and image processing applications: *dot product* and Finite Impulse Response (*FIR*) filter. The second paragraph deals with *RISCization*, that is, applying the *CISC-to-RISC* concept to SIMD instructions. Complex SIMD instructions are replaced by a set of more simple ones that should be efficiently implemented inside an FPGA. Finally, the last paragraph details the implementation of such instructions inside a Virtex4 FPGA. Estimation of maximum operating frequency and power consumption are provided.

II. SOFTWARE SIMD COMPUTATION

A. Architecture presentation

Altivec is a SIMD arithmetic unit featured with 128-bit vector registers. It is a more complete ISA than Intel SSE2 and SSE3, and provides some instructions like **vec_msum** which is very useful for *dot product* and *FIR* computations. The synopsis of the instruction **d=vec_msum(a,b,c)** [7] is given in Figure 1. Eight bit multiplications are performed to provide intermediate products that are accumulated through a 4-block reduction inside 32-bit blocks: $P = reduc_4(A.B)$. The second stage of the instruction performs an accumulation of P with a third register $D = reduc_4(reduc_4(A.B) + C)$.

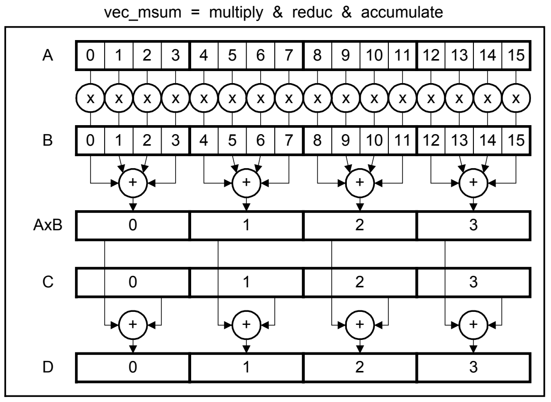

Fig. 1. Altivec **vec_msum** instruction

From a scalar point of view, the instruction performs sixteen 8-bit multiplications, eight 16-bit and four 32-bit sums, which gives a total of 36 scalar instructions inside a unique SIMD instruction. The 4-block reduction step makes this instruction well suited for filters whose size is a multiple of 4. One can note that for *dot product*, a second instruction **vec_sums** should be used to reduce the four 32-bit blocks $D0, D1, D2, D3$ inside one block (Figure 2).

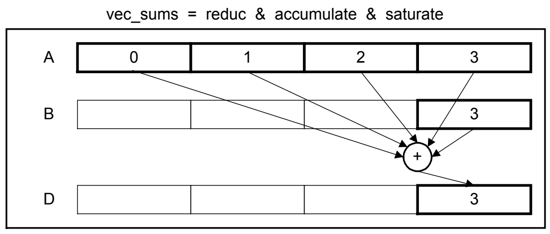

Fig. 2. 4-block reduction with **vec_sums** instruction

B. Software benchmark results

In this section, we provide a benchmark on PowerPC G4 to compare the execution times of the scalar and SIMD versions of *dot product* and *FIR*. The metric used is the number of clock cycles per pixel (*cpp*).

$$cpp = \frac{t.F}{N^2} \tag{1}$$

where N is the width of the input square image, t the execution time and F the frequency of the processor. In the case of PowerPC G4, F equals to 1 GHz. We believe that *cpp* is more representative than *cpi* (cycles per instruction) to compare the complexity per point of different algorithms. Moreover, *cpp* is also useful to detect cache misses from one image size to another. For the benchmarks, data size varies from 128×128 up to 1024×1024.

The first benchmarked operator is *dot product*, which operates on two vectors of size N^2, and computes a scalar according to the equation:

$$\mathbf{a}.\mathbf{b} = \sum_{i=0}^{N^2-1} a_i.b_i \tag{2}$$

Figure 3 is a comparison between the scalar and SIMD versions of *dot product*, using the dedicated instruction **vec_msum**. While the offset between the scalar and SIMD versions appears to be constant($\Delta \simeq 12$ *cpp*), a big gap of performances shows up: the speedup is ×4.1 for big sizes ($N = 1024$), and increases up to ×45 for small sizes of vectors ($N = 128$), when the whole data can fit in cache.

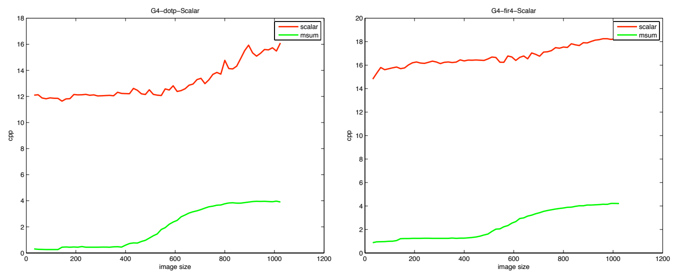

Fig. 3. *cpp* of *dot product* and *FIR4* on PowerPC G4

The second operator is a 4-tap *FIR* filter:

$$Y(i,j) = \sum_{k=0}^{3} f(k).X(i,j+k) \tag{3}$$

We use a 4-tap filter to make the dedicated instruction **vec_msum** the most efficient. Figure 3 provides the same kind of results for *FIR*. In that case too, there is a constant offset between the scalar and the SIMD version: $\Delta \simeq 14$ *cpp*. The speedup is ×4.4 for big sizes and increases up to ×14.9 for small sizes. One can note that, the maximum speedup (×45) is greater than the degree of parallelism ($p = 16$), since it does not only come from the CPU architecture, but also from the cache behavior: SIMD versions perform efficient and aligned accesses to the cache.

III. RISCIZATION

Before switching to FPGAs and embedded systems, an intermediate software step addresses the problem of instruction complexity. This is what we call *RISCization*. Is it efficient to replace complex (*CISC*) SIMD instructions like **vec_msum** by a set of simpler (*RISC*) SIMD instructions of the same ISA ?

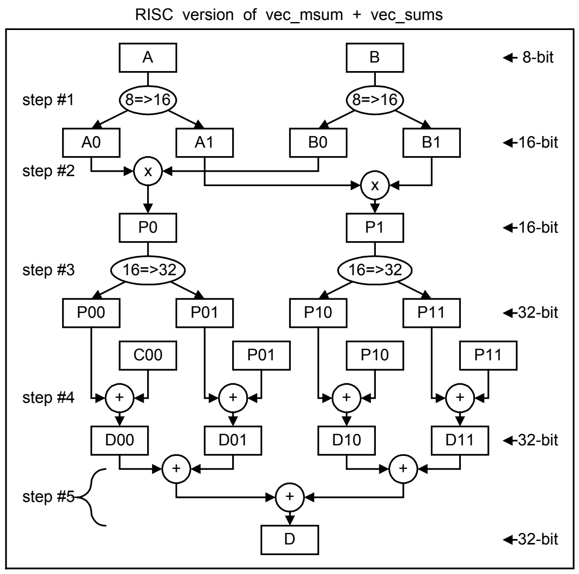

Fig. 4. **vec_msum** *RISCization*

If we have a lool at **vec_msum**, there are several ways to split this instruction. For example, by replacing 8-bit by 16-bit multiplications (16-bit×16-bit→16-bit) or by replacing 4-block reductions by 4 separate accumulators. The most simplified equivalent set of instructions is given in Figure 4 and is described below:

- Step 1: input 8-bit operands A,B are converted into two 16-bit data (low part and high part).
- Step 2: a 16-bit block wise multiplication is performed: no overflow neither truncation since input data are 8-bit wide.
- Step 3: conversion to 32-bit blocks.
- Step 4: 32-bit accumulation.
- Step 5: reduction (using **vec_sums**) to sum the four 32-bit blocks.

Figure 5 and 6 present the best *cpp* for three CISC and RISC versions. A lot of implementations have been done, with various schemes to replace reduction instructions for RISC version. Only three are presented, one for each multiplication instruction of Altivec ISA: **vec_msum**, **vec_mladd** and **vec_mule/vec_mulo**.

- CISC version with the **vec_msum** *reduction* instruction,
- RISC #1 version, with the **vec_mladd** 16-bit multiplication-accumulation instruction: $D = A \times B + C$,
- RISC #2 version, with only the **vec_mule** and **vec_mulo** 8-bit multiplication (8-bit×8-bit→16-bit) instructions.

For RISC versions, reductions are replaced by a set of additions (**vec_add**) and permutations(**vec_perm**). Of course, RISC versions are not faster than the CISC one that implements the **vec_msum** instruction. But when switching to an FPGA implementation, this could be no more true since running frequencies depend on the complexity of the involved logical blocks (worst critical path).

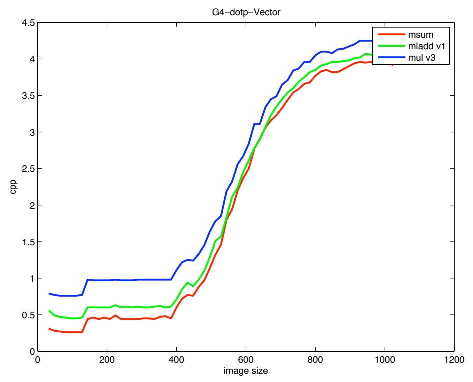

Fig. 5. *cpp* of *dot product* on PowerPC G4

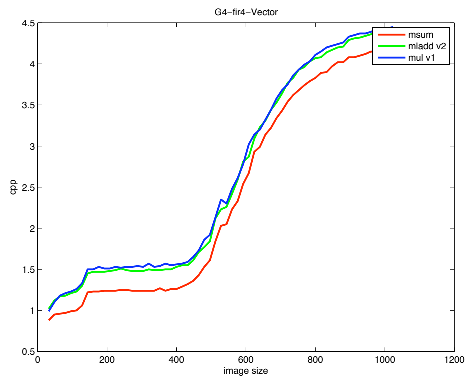

Fig. 6. *cpp* of FIR Filter on PowerPC G4

IV. Embedded Implementation

To estimate the impact of *RISCization*, SIMD instructions have been synthesized into a Virtex4 FPGA. RISC and CISC blocks combinations have been compared through orthogonal criteria like area utilization, running frequency F, power consumption P and execution time t. While *cpp* is meaningful for software benchmarks, t should be preferred in the case of hardware benchmarks. Since the running frequency depends on the implemented operators, the fastest implementation (with the smallest t) is not necessarily the one with the smallest *cpp*. And in image processing, execution time is one of the constraints to enforce.

Version	*cpp*	Area %	F (MHz)	P (Watt)	t (ms)	E (μJ)
256 × 256 data						
msum	**0.44**	22.5	151	0.202	0.191	39
mladd	0.61	24.2	256	0.165	**0.156**	**26**
mul	0.97	23.5	209	0.110	0.303	33
512 × 512 data						
msum	**1.32**	22.5	151	0.202	2.292	463
mladd	1.51	24.2	256	0.165	**1.545**	**255**
mul	1.78	23.5	209	0.110	2.227	245
1024 × 1024 data						
msum	**3.91**	22.5	151	0.202	27.154	5485
mladd	3.97	24.2	256	0.165	**16.251**	2681
mul	4.19	23.5	209	0.110	20.969	**2307**

TABLE I

dot product SYNTHESIS RESULTS

Version	*cpp*	Area %	F (Mhz)	P (Watt)	t (ms)	E (μJ)
256 × 256 data						
msum	**1.25**	20.5	164	0.213	0.499	106
mladd	1.49	21.2	337	0.258	**0.290**	**75**
mul	1.53	21.5	337	0.264	0.298	79
512 × 512 data						
msum	**1.84**	20.5	164	0.213	2.940	626
mladd	2.12	21.2	337	0.258	**1.650**	**426**
mul	2.13	21.5	337	0.264	1.658	438
1024 × 1024 data						
msum	**1.84**	20.5	164	0.213	26.908	5731
mladd	2.12	21.2	337	0.258	**13.700**	**3535**
mul	2.13	21.5	337	0.264	13.856	3658

TABLE II

FIR SYNTHESIS RESULTS

Two assumptions have been made for the comparison: FPGA and PowerPC G4 have the same Altivec instructions latencies and the FPGA could be interfaced with a memory hierarchy whose specifications (cache size, associativity, and latencies) would be equivalent to those of PowerPC G4 ones. Using *Xilinx ISE* and *XPower Estimator* tools, power consumptions have been estimated for each standalone SIMD Altivec instruction (these measures do not take into account the consumption of the I/O blocks). Tables IV and II present the results for 256×256, 512×512 and 1024×1024 data sizes.

In order to estimate the efficiency of the embedded system, we measure the amount of energy $E = t \times P$ required to compute the algorithm. Once the processing speed is enforced (typically $t <$ 33ms for 30 images/s), the total energy E is the constraint to optimize, not only the power P. One can note also, that synthesis tools provide the *maximum* running frequency. If the overall algorithm implementation is too fast, implying a too high power consumption, the system can be *downclocked*.

First, if we look at the execution time, we can see that the CISC version is never the fastest one, but **vec_maldd** which is a complexity golden mean between **vec_msum** and **vec_mule/vec_mulo**. Gains vary from ×1.2 up to ×1.6 for *dot product* and from ×1.7 up to ×2.0 for *FIR*. These results are very important: from the execution time point of view, a direct implementation into an FPGA of the classic dedicated and "optimized-for" CISC instruction will lead to a non optimal implementation .

Second, if we focus on energy, the gains between CISC and RISC version range from $\times 1.6$ up to $\times 2.0$ for *dot product* and from $\times 1.4$ up to $\times 1.6$ for *FIR*. That validates our *RISCization* approach: usually the fastest implementation is also the most power hungry one, but it the case of mapping SIMD instruction from a general purpose processor to an FPGA, this is not the case: RISC implementations are quick and energy efficient !

V. Instructions Customization / Specialization

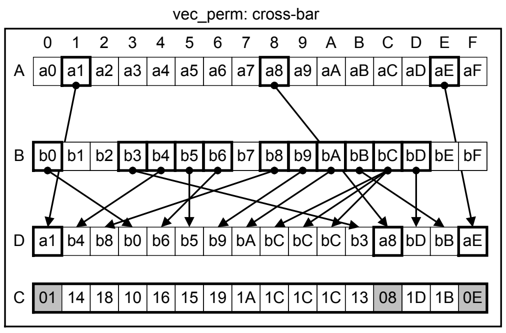
vec_perm: cross-bar
0 1 2 3 4 5 6 7 8 9 A B C D E F
A a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aA aB aC aD aE aF
B b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 bA bB bC bD bE bF
D a1 b4 b8 b0 b6 b5 b9 bA bC bC bC b3 a8 bD bB aE
C 01 14 18 10 16 15 19 1A 1C 1C 1C 13 08 1D 1B 0E

Fig. 7. Altivec permutation instruction

The last step to improve FPGA implementations is to reduce the useless capabilities of versatile instructions. The Altivec instruction **vec_perm** (synopsis Figure 7) is an interesting example. This instruction behaves like a complete crossbar and can perform permutations with regular patterns (as SSE2+ instructions can do), but also irregular patterns. For an algorithm or an application, only a few set of these capabilities are useful. For *dot product* and *FIR*, the only need is to construct unaligned vector registers. By specializing, on demand, such an instruction, area size and power consumption can be drastically reduced. Specialization has been applied to the *multiply-add* operation **vec_mladd**. The instruction is split into two instructions: **vec_mul** that does not exist in Altivec ISA in 16-bit version, and **vec_add** (**vec_mladd**(A,B,C)=**vec_add**(**vec_mul**(A,B),C)).

instruction	area reduction	power reduction
vec_sums	32 %	1 %
vec_msum	32 %	3 %
vec_mladd	22 %	3 %
vec_sum4s	30 %	3 %
vec_perm (v1)	56 %	5 %
vec_perm (v2)	35 %	5 %

TABLE III

GAIN PROVIDED BY CUSTOMIZATION

Table V presents the area and power consumption reduction provided by instruction specialization. One can note that power gain is not important because the difference between versions resides in the logical blocks; these blocks are not great power consumers. However, customization can significantly reduce area occupation. As SIMD instructions are quite big they can prevent from completing a synthesis. Area reduction is the only solution to make all the required SIMD instruction fit into an FPGA.

VI. Conclusion

We have presented the design of an Altivec SIMD instruction unit for FPGAs. The major advantages of designing an Altivec compatible unit is to be able to directly reuse PowerPC-aimed C code, without modifying it, into an FPGA, nor adding bugs, and thus reducing development time. We have used optimization techniques like instruction specialization and presented a new one called *RISCization* applying the *CISC-to-RISC* concept to hardware instruction implementation. Impacts of *RISCization* on basic signal processing algorithms are very interesting: the energy consumption has been reduced by a factor ranging from $\times 1.4$ up to $\times 2.0$ and area has been also reduced by a factor ranging from $\times 1.3$ up to $\times 1.5$. Current and future works are to develop high-level tools to perform an automatic design space exploration of the configurations to efficiently implement Altivec coded applications into an FPGA.

References

[1] S. Patel, *Altivec Programming: A Hands-On Tutorial*, Architecture and Performance Group, Apple Computer Inc.

[2] K. Diefendorff and al, *Altivec Extension to PowerPC Accelerates Media Processing*, 23 IEEE Micro, vol. 20, no. 2, Mar. 2000, pp. 85-95.

[3] L. Gwennap, *Altivec Vectorizes PowerPC*, Microprocessor Report, vol. 12, no. 6, Mar. 1999.

[4] D. Talla and L.K. John, *Cost-effective Hardware Acceleration of Multimedia Applications*, in Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors, 2001.

[5] L. Han, J. Chen, C. Zhou, Y. Li, X. Zhang, Z. Liu, X. Wei and B. Li, *An Embedded Reconfigurable SIMD DSP with Capability of Dimension-Controllable Vector Processing*, in Proceedings of the IEEE International Conference on Computer Design (ICCD'04).

[6] John L. Hennesy, *Computer Architecture: a quantitative approach*, third edition, Morgan Kaufmann Series in Computer Architecture and Design.

[7] *Altivec Technology Interface Manual*, Motorola, Freescale Semiconductor.

[8] *Apple Developer Connection: Velocity Engine*, `http://developer.apple.com/hardwaredrivers/ve/` .