Introducing image processing and SIMD computations with FPGA soft-cores and customized instructions

Daniel Etiemble and Lionel Lacassagne

***Abstract*—Using an Altera Cyclone kit with NIOS II processor, we show how reconfigurable computing can be used to introduce many optimizing issues of image processing and media applications together with basic notions: replacing software functions by hardware operators, customizing instructions, defining and using SIMD instructions. This approach can be used both for EE and CS students at undergraduate or graduate levels. Teaching can be organized as an introduction with a couple of lectures and labs within a computer organization, reconfigurable computing or image processing course or it can be extended as a semester "self-contained" course on "Image Processing with Reconfigurable Computing".**

***Index Terms*— SIMD, customized instructions, FPGA soft-cores, image processing.**

I. Introduction

This paper is based on lectures and labs in "advanced computer architecture" for fourth year undergraduate students in the Computer Sciences department of Paris Sud University. These students have previously had basic courses in "Logic design" and "Computer architecture and Organization", but have no knowledge of hardware description language and image processing. The "Advanced Computer Architecture" course focuses on program optimizations according to architectural features of the computing systems for the different classes of applications including the embedded systems.

The Altera NIOS Development kit (Cyclone edition) includes the NIOS soft-core for which customized instructions can be defined [1]. Custom logic can be added in parallel to the 32-bit processor ALU to customize different types of instructions, as shown in Fig. 1. Combinational hardware (single cycle instruction) is the simplest approach to introduce first. The customization of instruction is the basic technique that is used, first to replace software C functions by customized hardware instructions, then by defining and introducing SIMD instructions for low level image processing. Simple examples are used to introduce the SIMD concept and the main issues of SIMD programming: data formats and data handling for parallel computation.

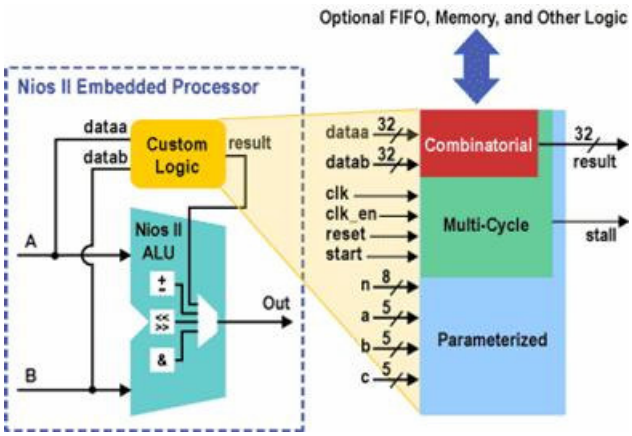

Fig. 1. Customizing instructions for the NIOS II processor

In this paper, we first introduce the hardware and software environments that are used, and the implemented image processing algorithms: two 3x3 image convolutions. Then, we present the different steps that are followed in the optimization procedure using customized instructions and the corresponding results. Possible extensions are then presented.

II. The Teaching Environment

We use the Altera NIOS development kit (Cyclone Edition) which includes the EP1C20F400C7 FPGA device. In the Labs, we use the NIOS II/f version of the processor,

Daniel Etiemble is with the LRI Laboratory, University Paris Sud, 91405 Orsay, France. F. (e-mail: de@ lri.fr).

Lionel Lacassagne is with the IEF Laboratory, University Paris Sud, 91405 Orsay, France (e-mail: lionel.lacassagne@ief. u-psud.fr).

TABLE I
NIOS II/F PROCESSOR FEATURES

Fixed features	Parameterized features
32-bit RISC processor Branch prediction Dynamic branch predictor Barrel shifter	4 KB instruction cache 2 KB data cache

which main features are summarized in Table I. Processor clock frequency is 50-MHz. For parameterized features, we mention the cache sizes that we used in the labs. All information on the device and processor features can be found in [2].

All the programs that we considered have been compiled with the Altera Integrated Development Environment (IDE) [3], which uses the GCC tool chain. –O2 or –O3 option has been used in release mode. Execution times have been measured with the high_res_timer that provides the number of processor clock cycles for the execution time. As we use different image sizes, the results are presented with the Cycle per Pixel metric, which is the total number of clock cycles divided by the number of pixels. For each program, the execution time has been measured at least 5 times and we have taken the average value.

During the different lab experiments, students must go through and learn every hardware and software steps that are needed before being able to execute their program on the FPGA device. They must design the custom logic to implement every customized instruction using VHDL (or Verilog) description, compile and test it using the Quartus software. Using the SOPC builder, they must choose and configure one NIOS version, add the customized instructions and generate the system (Verilog/VHDL files for the overall system). They must finally compile this system. Similarly, they learn how to use the Integrated Development Environment to write the different C/C++ versions of their programs and they learn how to measure execution times on the hardware board by using the "high_resolution_timer". All these steps are typical on the hardware and software issues when using soft-cores in FPGA boards. We now focus on the original aspects: using customized instructions to teach SIMD computing and image and media processing.

III. A FIRST CONVOLUTION KERNEL WITHOUT ARITHMETIC COMPUTATION: THE MIN FUNCTION

We consider NxN images for each pixel corresponds to a byte (8-bit gray levels) to which we apply 3x3 filters. The first one is the Min3x3() function that replaces the center pixel in a neighborhood with the minimum value in that neighborhood for each 3x3 neighborhood in the image.

The initial C program uses a C Min function shown in Fig. 2. In the second version, we replace the C Min function by a customized MIN32 instruction that delivers the min value of two 32-bit unsigned numbers. As a 32-bit RISC, the processor zero-extends the pixel "unsigned char" values. The VHDL code for the MIN32 function is trivial, even for novice students having no (or a minimal) knowledge of VHDL language.

```
unsigned char min (unsigned char a, unsigned char b)
{if (a <b) return a; else return b;}
```

Fig. 2. C Min function

In the third version, we customize a MIN_SIMD instruction that computes the MIN of each corresponding byte of two 4-byte words, as shown in Fig 3. Each 4-byte word contains 4 consecutive pixel values. This simple example raises one significant SIMD issue: some supplementary data handling instructions to correctly align the data before SIMD operations are introduced. The processor loads (and stores) 32-bit words, which means 4 pixel values with memory accesses aligned on word boundaries. But 3x3 filters mean that for any pixel (byte) on a line, the two neighbor pixels are needed. As unaligned word accesses are prohibited, shift byte right and left instructions are needed that access to two consecutive 4-byte words and generate the words with pixel values (j+2, j+1, j, j-1) and (j+4, j+3, j+2, j+1) when the aligned accessed word contains (j+3, j+2, j+1 and j) pixel values. These instructions are shown in Fig. 4.

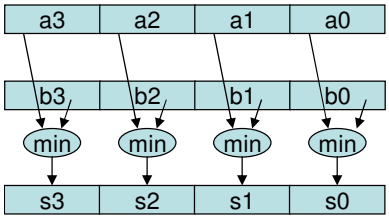

Fig. 3. SIMD Min customized instruction

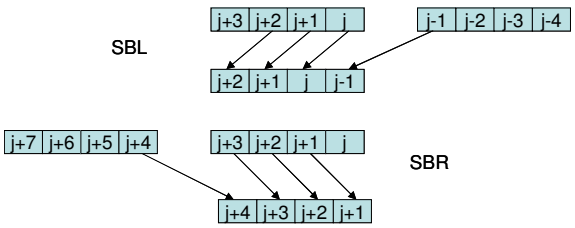

Fig. 4. Shift byte left and Shift byte right SIMD instructions

Table II presents the execution times for the different versions according to the NxN image size. The 32-bit Min

customized instruction is more than two times faster than the corresponding C function. The SIMD version, which uses both the 4x8 Min SIMD instructions and the different alignment instructions (SBL and SBR) are 2.9 times faster than the corresponding scalar version while the ideal speed-up is 4. It outlines the cost of data handling that is necessary for SIMD computation.

TABLE II
MIN FILTER EXECUTION TIMES (CPP)

N*N image	64	128	256
C Min function	126.4	126.1	126.1
32-bit Min function	53.9	53.6	53.6
4x8-bit SIMD Min instruction	18.7	18.3	18.3
HW Instruction/C function speed-up	*2.3*	*2.4*	*2.4*
SIMD/Scalar HW instruction speed-up	*2.9*	*2.9*	*2.9*

.

This simple example raises another issue. The pixels that lie on the sides of the image have no "outside" neighbor. With scalar computation, we can easily neglect applying the filter to the pixels on the first and last line and column of the image. The situation is more complicated with SIMD 4-byte accesses: without considering the word accesses that are respectively before the first word and after the last word of each line, 8 pixels per line are not processed instead of only 2 in the scalar case. In table II, we computed CPP as the execution time divided by (N-2)*(N-2) for the scalar cases and (N-2)*(N-8) for the SIMD case. However, improving the SIMD version to process (N-2)*(N-2) pixels would be necessary for actual applications.

IV. CONVOLUTION KERNELS WITH ARITHMETIC COMPUTATIONS: LAPLACIAN AND GAUSSIAN KERNELS

Typical low-pass spatial filters as Gauss filter (Fig. 5 left) or high-pass filter as Laplacian filter (Fig 5 right) raise another issue for SIMD computation.

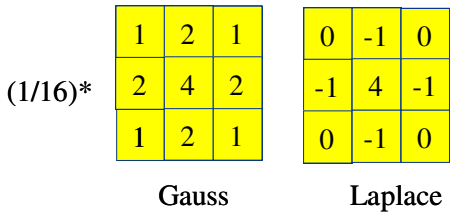

Fig. 5. Gauss and Laplace kernels

These filters use multiplication or division by powers of 2, that can be implemented by shift and add/sub arithmetic operations. Starting with byte pixel values, the intermediate computations need more than byte values. This is not an issue for scalar computation, as 32-bit processors zero or sign-extend the byte values to 32-bit integer values and all computations use 32-bit integers. For SIMD computations, the only solution that keeps the computing accuracy consists in zero-extending each 8-bit value to a 16-bit value before using SIMD arithmetic operations. In our case, ADD and SUB SIMD instructions implement 2x16 arithmetic operations. Now, the optimal speed-up versus scalar case is 2. Table III lists the customized instructions that are used for image digital filtering with integer computation.

Table IV presents the results for the Gauss kernel and Table V presents the corresponding results for the Gaussian kernel. In both cases, the SIMD speed-up is close

TABLE III
SIMD INSTRUCTIONS

Instruction	Action	Clocks
B2HL (A)	Two lower bytes of word A are zero-extended to a 2x16-bit word	1
B2HL (A)	Two higher bytes of word A are zero-extended to a 2x16-bit word	1
H2BL (A)	The 2x16-bit of word A are truncated to 2 lower bytes of a word. The two higher bytes of the result are zeroed.	1
H2BH (A,B)	The 2x16-bit of word A are truncated and placed in the two higher bytes of b. The lower bytes of b are unchanged.	1
SBL (A,B)	The three lower bytes of word A and the higher byte of B are shifted one byte left.	1
SBR (A,B)	The lower byte of A and the higher byte of B are shifted one byte right	1
ADDH (A,B)	2x16 bit addition	1
SUBH (A,B)	2x16 bit subtraction	1
SL1 (A)	Each 16-bit sub-word is shifted one position left	1
SL2 (A)	Each 16-bit sub-word is shifted two positions left	1
SR4 (A)	Each 16-bit sub-word is shifted four position right (logical shift)	1
MINB (A,B)	4x8 unsigned Min operation	1
MINW (A,B)	32-bit unsigned Min operation	1

or equal to 2, which means that the data handling overhead is insignificant.

Other low-pass or high-pass filters could be used. For low-pass filters, some divisions by constants that are not power of 2 raises other complications for SIMD computations.

TABLE IV
GAUSSIAN EXECUTION TIMES (CPP)

	64	128	256
C version	58.3	57.9	57.9
C version with SIMD instructions	31.1	28.6	29.6
Speed-up	*1.9*	*2.0*	*2.0*

TABLE V
LAPLACIAN EXECUTION TIMES (CPP)

	64	128	256
C version	35.1	34.8	34.7
C version with SIMD instructions	19.2	17.4	17.4
Speed-up	*1.8*	*2.0*	*2.0*

V. INTEGER OR FLOATING POINT COMPUTATIONS?

For more complex image processes, a careful examination of the data dynamic range during each processing step is needed. Research of "points of interest" within an image is such a typical image processing application that is used for image stabilization in robotics: the objective is to reduce the image to a limited set of points considered as the most representative of the whole set to be used as an index for this image. Fig. 6 shows the Achard and Harris algorithms.

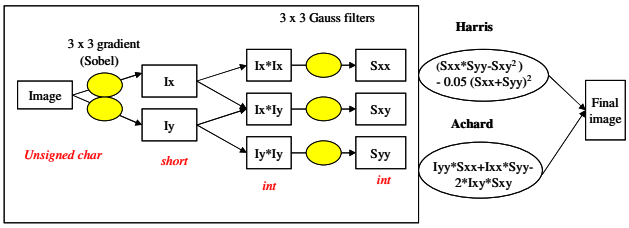

Fig. 6. Achard and Harris algorithms for detection of Points of Interest (POI).

These algorithms share most computations and differ by the final step. They include a 3x3 Sobel gradient followed by 3x3 Gauss filters. The common part is typical of low level image processing. For integer computations, initial images with 8-bit gray levels have unsigned char format to code the pixels. Sobel gradient computations lead to short format to avoid overflow and the following multiplications lead to int. format.

The SIMD instruction defined for Laplacian and Gauss kernels (Table III) can be used to speed-up the computation of the Sobel gradient. Then, the remaining part of the computations uses 32-bit integers and there is no way to define and use SIMD instructions because of the typical drawback of SIMD integer instructions: the output format of arithmetic operations is different from the input format (multiplying two 16-bit words provides a 32-bit word).

One interesting alternative consists in using floating-point numbers. Obviously, simple precision FP number is not a solution for many reasons: first, we need to implement the 32-bit FP addition and multiply instructions, which is rather complicated even with simplifying assumptions (no denormals, truncation instead of rounding); second, there is no way to implement SIMD simple precision FP instructions within 32-bit words. Third, using multi-cycles FP 32-bit instructions instead of 32-bit integer instructions will slow down the computation versus the scalar integer version.

Using 16-bit FP instructions that we have studied both for general purpose processors and embedded processors in previous papers [4,5] is an interesting alternative. The "half" format, introduced in the OpenEXP format [6] and in the Cg language [7] defined by NVIDIA is presented in Fig. 7. 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(Exponent field-15). The range of the format extends from 2-24 = 6 x 10-8 and (216-25) = 65504. In the remaining part of this paper, the 16-bit floating point format will be called half or F16.

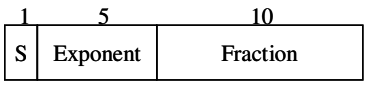

Fig. 7. "half" format (F16 format)

From teaching point of view, introducing FP computing instructions is interesting. The students must implement VHDL pipelined versions of FP operators, which is a significant VHDL lab. Considering 16-bit FP numbers, without denormals and with the simplest rounding mode (truncation) makes things easier. It turns out that using 16-bit FP numbers instead of 32-bit FP number has no impact on the image processing for the considered benchmarks, as we have shown in [4]. The student can also consider this

issue by using PSNR metrics.

The F16 SIMD instructions presented in Table VI are the FP versions of the integer SIMD instructions defined in Table III. A specificity is that the shift instructions that are used for data alignment can be combined with byte to F16 conversions (Fig. 8) to minimize the number of instructions and the execution time.

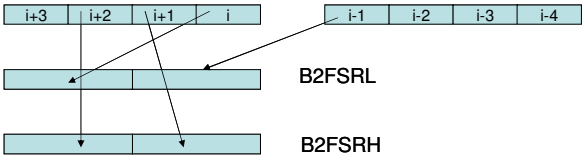

Fig. 8. Byte to F16 and shift conversion instructions

Tables VII and VIII give the execution time (CPP) for Achard and Harris algorithms. As the algorithms have a large common part, the results are close and significant of algorithms including a lot of low-level image processing. We have also added the results for simple precision 32-bit FP computations. The F32 arithmetic instructions are similar to the F16 versions (no denormals, truncation as rounding mode). The F32 add and mul latencies are respectively 4 and 3 cycles. As expected, the F32 version is slower than the original integer C version. The F16 version is more than 50% faster than the original C version. The speed-up of the F16 version versus F32 is more than 2: it comes from the SIMD implementation of F16 instructions and the lower latencies of the F16 arithmetic instructions versus the F32 corresponding arithmetic instructions. Obviously, many other different image processing benchmarks could be used.

TABLE VI
F16 SIMD INSTRUCTIONS

Instruction	Action	Clocks
B2F16L (A)	Converts the two lower bytes of A into two F16	1
B2F16H (A)	Converts the two higher bytes of A into two F16	1
F2BL (A)	Converts two F16 of A into two unsigned bytes in the lower part of a 32-bit word and zero in the higher part.	1
F2BH (A,B)	Converts two F16 in A into two unsigned bytes in the higher part of B (lower part is unchanged)	1
S2F16 (A)	Converts two 16-bit integers in A into two F16	1
F2S (A)	Converts two F16 in A into two 16-bit integers	1
B2FSRL (A,B)	Converts the high order byte of A and the low order byte of B into two F16	1
B2FSRH (A)	Converts the two middle bytes A into two F16	1
FSR (B,A)	Put the low order F16 of word B into high order F16 of result. Put the high order byte of word A into low order F16 of result.	1
ADDF (A,B)	F16 SIMD addition	2
SUBF (A,B)	F16 SIMD subtraction (A-B)	2
MULF (A,B)	F16 SIMD multiplication	2

TABLE VII
ACHARD EXECUTION TIMES (CPP)

N*N image	64	128	256
C integer original program	349	360	366
F16	217	235	245
F32	560.5	578.5	NA
F16/Int speed-up	*1.60*	*1.53*	*1.49*
F16/F32 speed-up	*2.7*	*2.5*	*NA*

.

TABLE VIII
HARRIS EXECUTION TIMES (CPP)

N*N image	64	128	256
C integer original program	348	359	365
F16	212	230	240
F32	568	586	NA
F16/Int speed-up	*1.6*	*1.6*	*1.5*
F16/F32 speed-up	*2.7*	*2.5*	*NA*

.

VI. FURTHER DEVELOPMENTS

So far, we have concentrated the experiments on the image processing optimization issues by using customized SIMD instructions with the NIOS II processor. Applying a kernel or a more complex algorithm to an actual image is not necessary to measure the execution time of the different versions. In our experiment, we have considered N*N array filled up with an arbitrary initialization instead of actual images. Using actual image is a significant improvement, at least to check the correctness of the different versions and verify that the "optimized" versions produce the same image than the original C version of the program. Introducing the communication mechanism between the PC server and the FPGA board to download images to the board and upload the resulting image is another more "computer science oriented" step in a set of labs. The connection of a frame grabber to the FPGA board can also be introduced to present bus performances and DMA transfers.

Examining the hardware costs of the customized

instructions is also an interesting aspect. For instance, the hardware cost for all the SIMD integer instructions of Table III is 176 logic cells (LC) to compare with the 2,062 LCs for the CPU and 280 LCs to implement the custom instructions interface. One reason is that many instructions only shuffle input and output ports! In [5], we have shown that the hardware cost for all the F16 instructions in Table VI is 1,188 LCs, which is an acceptable overhead to the CPU cost.

To study more ambitious customization techniques, we can modify the currently used 16-bit floating point format by tuning the number of bits dedicated to fraction and exponent parts according to the dynamic range and accuracy that are needed by a specific application.

So far, the concepts that are presented in this paper have been experimented with the "light" version for computer science students having a minimal knowledge in advanced logic design and hardware description languages. The labs have mainly focused on the optimization issues. For such students, it is wise to prepare and provide (if needed) the VHDL files of the customized instructions. One potential drawback is that compiling the whole system including the NIOS processor needs around 20 minutes on a 1.6-GHz Pentium-4 PC, which means that everything must be carefully prepared between launching the compilation. The student feedback was good.

VII. Concluding Remarks

In this paper, we have shown that FPGA devices with a soft-core CPU are powerful devices to introduce most of the optimization issues of image processing, from elementary low-level image processing to more sophisticated image treatment. Customizing instructions with the hardware resources of the FPGA gives significant insight on most of the hardware/software trade-offs. At the same time, the approach can be used at different teaching levels. For students with limited knowledge in electrical engineering (and hardware description languages), most of the VHDL operators that are needed to customize the instructions are so simple that they can easily implement them. For students having passed basic courses in logic design (including hardware description language) and computer architecture), far more complicated pipelined operators can be used and the course can evolve from an introduction to image processing and SIMD computation to an advanced course.

References

[1] Altera, "NIOS II Custom Instructions User Guide", Jan 2005, http://www.altera.com.cn/literature/ug/ug_nios2_custom_instruction.pdf

[2] Altera: www.altera.com

[3] Altera, NIOS II Integrated Development Tool Tutorial, http://www.altera.com.cn/support/software/embedded/ide/tutorial/soft-ide_tutorial.html

[4] L. Lacassagne and D. Etiemble, "16-bit floating point operations for low-end and high-end embedded processors", in Digests of ODES-3, March 2005, San Jose, Available at http://www.ece.vill.edu/~deepu/odes/odes-3_digest.pdf

[5] L. Lacassagne and D. Etiemble, "Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing", Proc. Estimedia 2005, September 2005, Jersey City, USA

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

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