Performance evaluation of the Altera C2H compiler on image processing benchmarks

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

Abstract

This paper dela with the use of the C2H Altera compiler for the automatic VHDL synthesis of image processing function. The C2H compiler has been used to accelerate low-level and medium level image processing benchmarks. After code transformation, speedups between 6 and 10 have been obtained. For loops with a recurrence, a speedup greater than 2 has been obtained;

1. Introduction

FPGAs with soft-core processors offer the opportunity for testing various trade-offs between hardware and software implementations of the functions to implement. With the Altera NIOS II the processor can be customized through the addition of new instructions [1]. Custom specific functions can be implemented as coprocessors. Some months ago, Altera has provided a C to Hardware compiler that can be used to speed-up some C functions within a C program [2]. In this paper, we present a preliminary performance evaluation of the C2H compiler on image processing benchmarks. These benchmarks are representative of low-level and intermediate level image processing and include a lot a computation and memory accesses. We compare the compiler results with results delivered by customized implementation of SIMD instructions on the NIOS processor. We show the basic C transformations that provide the best C2H results.

2. Methodology

2.1 The Altera C2H compiler

As previously mentioned, the Altera C2H compiler is described for the user in [2]. It is integrated in the NIOS II integrated development environment (IDE) and generates hardware accelerator for performance-critical sections of code. As noticed in [2], the hardware accelerators generated by the C2H compiler have the following characteristics:

- Parallel scheduling: the C2H compiler recognized events that can occur in parallel. Independent statements are performed simultaneously in hardware.
- Direct memory access: accelerators access the same memory that the NIOS II processor does during execution
- Loop pipelining: the C2H compiler pipelines the logic implemented for loops, based on memory access latency and the amount of code that operates in parallel.
- Memory access pipelining: the C2H compiler pipelines memory accesses to reduce the effect of memory latency.

Basically, the code to accelerate must be expressed as an individual C function. For image processing kernels, the corresponding function is the loop nest that operates on every image pixel.

2.2 The different benchmarks

We consider a set of benchmarks that are representative of low-level and intermediate level of image processing. The scalar C code for each used benchmark is available at [3].

The first benchmark is derived from the description of the EEMBC Grayscale benchmark [3]. It is a high pass filter, which apply a 3 x 3 kernel (figure 1) to each byte pixel (level of gray) of an N x N image. No multiplications or divisions are needed, as they can be replaced by combinations of shifts and add/sub operations ($255 = 2^8-1$; $28 = 2^5-2^2$). However, this filter is interested as accesses to all neighbor pixels are needed and the nine pixel values are needed to compute the filtered value.

$$\frac{1}{256}\begin{bmatrix} -28 & -28 & -28 \\ -28 & 255 & -28 \\ -28 & -28 & -28 \end{bmatrix}$$

Figure 1: "Grayscale" filter derived from the description of EEMBC Grayscale benchmark.

The second sets of benchmarks are the Deriche filters. The horizontal version has two inner loops with a recursion that prevents parallel executions of successive iterations. The horizontal-vertical version allows parallel execution of successive iterations of the inner loops. The Deriche gradient has no multiplications, but includes the abs function.

The third type of benchmarks corresponds to Achard and Harris algorithms to detect points of interest within an image. Figure 2 shows these algorithms. They 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 levels of gray 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. We will only provide the results for the Harris algorithm.

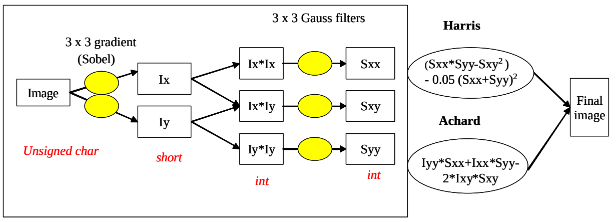

Figure 2: Achard and Harris algorithms to detect points of interest (PoI).

Optical flow algorithms compute the difference between successive moving images. In this category, the Horn and Shunk algorithm is the last benchmark that we used. Opposed to other benchmarks, it involves divisions, which makes it particularly interesting for performance evaluation.

2.3 Measures

We used the Stratix II Altera kit. The Stratix II device has up to 60k logic elements, 2 Mb SRAM and 36 DSP blocks that can implement 144 18bx18b multipliers. We used a 50-MHz NIOS II processor (fast version) with hardware integer multiplication for all benchmarks and hardware division for the Optical Flow benchmark.

All the benchmarks have been compiled with the Altera Integrated Development Environment (IDE), which uses the GCC tool chain. –O2 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. The results use the Cycle per Pixel metrics (CPP), which is the total number of clock cycles divided by the number of pixels. For each benchmark, the execution time has been measured at least 5 times and we have taken the averaged value. The measures have been done for N x N images, with N=128, 132, 256, 260, 512 and 516. All the results can be found in [3]. In this paper, we will only provide the results for N=256 and N=260. When N is a power of two (256 for instance), there are a lot of cache conflicts that results from the direct mapping policy that is used for the NIOS II data cache. This is why it is significant to provide the results when N is not a power of two.

3. Experimental results.

3.1 "Grayscale" results.

It is worthy giving some details on the C code transformations to increase the C2H efficiency. As the accelerator should access the data memory through the Avalon Switch Fabric, it is quite evident that reducing the number of memory accesses will generally increase the C2H efficiency. For 3 x 3 kernels such as shown in Figure 1, this can be done according to several techniques. The first one consists in replacing byte accesses by 32-bit word accesses. The pixel values are "unsigned char", but 4 successive pixels in a row can be accessed as a 432-bit word. This is equivalent to unroll 4 times the inner loop. Unrolling 4 times the inner loop multiplies by 4 the number of hardware operators that are needed to implement the inner loop. The second technique consists in balancing the memory accesses between the inner and the outer loop. The outer loop accesses to X[i-1][0], X[i][0], X[i+1][0], X[i-1][1], X[i][1], X[i+1][1] while the inner loop accesses to X[i-1][j+1], X[i][j+1] and X[i+1][j+1]. At the end of the inner loop, the values for columns j-1 and j are updated. Obviously, these accesses can be byte or 32-bit word accesses, which mean that the two techniques can be combined. When combining the two techniques, the loop latency (LL) and cycles per loop iteration (CPLI) of the two loops are given in Table 1. The CPP performance are given in Table 2, in which the measures for the initial C version and the software and C2H versions combining the two techniques (3 word accesses per inner loop). The speedup is computed between the software and C2H optimized versions to only consider the actual acceleration provided par the C2H compilation when $N = 2^k$. Otherwise, the speed-up versus the initial version would be artificially increased due to the data cache direct mapping. The speed-up is dramatic for this benchmark for which all computations are parallel.

Loop	LL	CPLI
Outer loop	12	10
Inner loop	13	7

Table 1: Performance of the "grayscale" accelerated loops

N	Initial	JU4	JU4(C2H)	Speed-up
256	54.43	35.52	2.52	14.10
260	31.01	31.15	2.52	12.36

Table 2: CPP performance for "grayscale"

3.2 Deriche benchmarks.

The Deriche gradient has also parallel computations. The abs function has been in-lined in the code. The results, presented in Table 3, are similar to "Grayscale results".

The Horizontal-Vertical version of Deriche filter involves multiplications and additions, but has no obstacle to parallel executions. The results are presented in Table 4.

The Horizontal version of Deriche filter, which is the most useful, has inner loops with a recurrence. For this benchmark, we have tested two versions: one unrolls 4 times the outer loop with byte accesses, while the second unrolls 4 times the inner loop with word accesses. The corresponding results are presented in Table 5 and 6. Unrolling the outer loop gives slightly better results (about 12.5 CPP versus 14.1 CPP) as parallel computations are possible in the inner loops. However, it cannot benefit from word accesses. On the other hand, unrolling the inner loops benefit from word accesses, but the recurrence "serializes" the inner loop execution.

N	Initial	U4J	U4J(C2H)	Speedup
256	45.09	26.98	2.05	13.16
257	21.09	22.31	2.05	10.88

Table 3: CPP performance for the Deriche gradient

N	Initial	U4J	U4J(C2H)	Speedup
256	83.88	27.22	9.01	3.02
260	34.83	16	9.01	1.78

Table 4: CPP performance for the HV version of Deriche filter

N	Initial	U4i	U4i(C2H)	Speedup
256	65.45	50.87	12.53	4.06
260	38.8	29.31	12.53	2.34

Table 5: CPP performance for the horizontal version of the Deriche filter (outer loop unrolled and byte accesses)

N	Initial	U4j	U4j(C2H)	Speedup
256	65.45	30.36	14.08	2.16
260	38.8	29.55	14.08	2.10

Table 6: CPP performance for the horizontal version of the Deriche filter (inner loops unrolled and word accesses).

3.3 Harris benchmark.

The Harris benchmark (Figure 2) is more challenging. It includes successively a Sobel filter and a Gauss gradient before the final computation and should be decomposed into several functions. Again, we must trade-off computations and memory accesses. We have tried two different decompositions. In the first one, the first function includes the Sobel filters and the multiplications (Ix*Ix, Ix*Iy and Iy*Iy) and the second function includes the Gauss gradient and the final computation. This decomposition disadvantage is that both functions have input and output arrays of different sizes (byte arrays and word arrays). In the second decomposition, the first function only includes the Sobel filters and the second one the remaining part of the computation. The second decomposition is more efficient. In this decomposition, the F1 function (Sobel filter) inner loop is 4-times unrolled with word accesses. The F2 function has byte accesses and balanced memory accesses and computation between outer and inner loops to reduce both.

The CPP performances are given in Table 7 and the accelerated loop features in Table 8. The overall speedup is close to 10 and even greater when $N = 2^k$. However, to obtain such a speedup, the original C code should be significantly transformed by choosing the best decompositions into different functions to minimize both the memory accesses and the amount of computations.

N	256	260
Initial	517.76	172.04
F1	38.58	38.66
F2	146.76	120.95
F1+F2	185.35	159.61
F1(C2H)	2.5	2.5
F2(C2H)	14.05	14.05
Overall (C2H)	16.55	16.55
Speedup	11.20	9.64

Table 7: CPP performance for Harris benchmark.

	OL-LL	OL-CPLI	IL-LL	IL-CPLI
F1	12	10	14	7
F2	21	19	22	12

Table 8: Loop latency and Cycles per loop iteration for Harris accelerated outer and inner loops.

3.4 Optical flow benchmark.

The optical flow benchmark corresponds to only one function. We used int arrays for inputs and outputs. As with previous benchmarks, we access the elements of column 0 and 1 in the outer loop and we access of column j+1 in the inner loop. For the C2H accelerator, the loop latencies/cycles per loop iteration are respectively 21/19 and 31/22 for the outer and inner loops. CPP results are given in Table 9.

N	Initial	F1	F1(C2H)	Speedup
128		205.1	31.85	6.44
132		207.5	31.88	6.51

Table 9: CPP performance for the optical flow.

4. Hardware costs

For several benchmarks, we have looked at the hardware implementation of the different accelerators. We should mention that we tried to get significant speedups on the execution times without trying to minimize the hardware cost.

In this section, we present the hardware cost of the accelerator as the percentage of available resources on the FPGA device that are used for the accelerator. This can be compared to the percentage of resources that are used to implement the NIOS II processor. We focus on the number Adaptative Logic Module (ALM) and the number of DSP elements (that are equivalent to 9 bit x 9 bit multipliers). The Stratix II device has 24,176 ALMs and 288 DSP elements.

Table 10 presents the percentage of available resources used by the different accelerators. As previously mentioned, the horizontal version of the Deriche filter uses either one or the other accelerators. For Harris, both F1 and F2 are used. The Harris row is the sum of F1 and F2 rows.

	ALMs	DSP Elements
CPU	4.62%	2.78%
Deriche_H U4i	5.12%	66.67%
Deriche_H U4j	14.61%	30.56%
Harris F1	3.99%	13.89%
Harris F2	7.25%	30.21%
Harris	11.24%	44.10%
Optical flow	23.25%	16.67%

Table 10: Percentage of available hardware resources used by the NIOS II CPU and the different accelerators.

For all benchmarks that are considered in Table 10, the accelerators use more hardware resources than the 32-bit NIOS II RISC CPU. On the other hand, the hardware resources that are used by the accelerators use less than 25% of the available ALMs and less than 50% of the DSP elements, except for the Deriche_H U4i version that use 2/3 of the DSP elements. Again, we notice that we didn't focus on the minimization of hardware resources.

5. Advantages and issues of C2H compilation.

In other papers, we have considered the customization of 16-bit SIMD integer and floating point instructions and evaluated the SIMD performance with the benchmarks that we used in this paper. Even if 32-bit memory accesses allow loading and storing 4 bytes, 2x16-bit integer arithmetic SIMD instructions are needed to deal with carry propagation. It means that the maximal speedup that can deliver SIMD instructions is 2. Obviously, the maximal speedup provided by 16-bit SIMD floating point instruction is also 2.

With C2H compilation, the maximal speedup that can be obtained with 32-bit accesses when computing byte values is 4 as the four bytes can be simultaneously computed. However, this pseudo-SIMD computation uses more hardware that what would actually be needed. As the C language doesn't provide a simple way to access individual bytes inside a 4-byte word, the parallel computation of the 4 different bytes is done by

- Masking each individual byte with 0xFF, 0xFF00, 0xFF0000 and 0xFF000000 individual masks.
- Shifting by 0, 8, 16 and 24 bit right the results of the previously masked integers.
- Doing the parallel computation required by the function, either as 16-bit computation or 32-bit computation. The computation generally delivers results within an 8-bit range.
- Shifting by 0, 8, 16 and 24 bit left the results of the previous step
- Merging the four different results to get the final 32-bit word including the 4-byte word to store in memory.

These different steps are “simulating” SIMD computation, but use more hardware resources than what would be needed if SIMD computation could be described with the C language.

6 Concluding remarks

We have tested the Altera C2H compiler on image processing benchmarks, from a typical high pass filter corresponding to low level processing up to more significant benchmarks used for image stabilization in robotics.

To get a significant C2H compiler efficiency, a good expertise of program optimizations is needed. The functions that we have accelerated have been transformed from the original versions, using decomposition in different functions, loop unrolling and techniques to reduce the number of memory accesses.

After these transformations, the C2H compiler is rather efficient on these benchmarks, which are not trivial ones. Speedups in the 6 to 10 range have been obtained when there is no significant obstacle to parallelism. Even for the horizontal version of Deriche filter, which has a recurrence in the inner loop, a speedup greater than 2 has been obtained in a situation where it is impossible to use SIMD parallelism.

These tests have been done with a 4-year old 1.6 GHz, 256-MB Pentium 4 laptop. According to the benchmarks, the overall time needed to build software, generate SOPC builder system and run Quartus II compilation ranges from one to several hours. Programs should be carefully prepared to avoid discovering mistakes in the description of an accelerated function… several hours later.

References

[1] Altera, “NIOS Custom Instructions, Tutorial”, June 2002, http://www.altera.com/literature/tt/tt_nios_ci.pdf

[2] Altera, “NIOS II C2H Compiler User Guide”, May 2006, www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf

[3]] Benchmark code : http://www.lri.fr/~de/F16/code-C2H

[4] D. Etiemble, S. Bouaziz and L. Lacassagne, "Customizing 16-bit floating-point instructions on a NIOS II processor for FPGA image and media processing", in IEEE Workshop on Embedded Systems for Real Time Media Processing (Estimedia), Jersey City, September 2005.