IMPACT OF HIGH LEVEL TRANSFORMS ON HIGH LEVEL SYNTHESIS FOR MOTION DETECTION ALGORITHM

H. Ye[1], *L. Lacassagne*[2], *D. Etiemble*[3], *L. Cabaret* [4], *J. Falcou*[3], *A. Romero*[2] *and O. Florent*[1]

[1] ST Microelectronics, F-38019 Grenoble, France
[2] IEF: Institut d'Electronique Fondamentale, Univ. Paris-Sud, F-91405 Orsay, France
[3] LRI: Laboratoire de Recherche en Informatique, Univ. Paris-Sud, F-91405 Orsay, France
[4] ECP/LISA: Ecole Centrale de Paris, F-92295 Châtenay-Malabry, France

ABSTRACT

High Level Synthesis for System on Chip is a challenging way to cut off development time, while assuming a good level of performance. But the HLS tools are limited by the abstraction level of the description to perform some high level transforms. This paper evaluates the impact of such high level transforms for ASICs and softcores on FPGA. On the representative example of motion detection, we show that we have a speedup of $\times 1.5$ for a softcore on FPGA and $\times 2.5$ for an ASIC while the energy is divided by a factor $\times 2.90$ for the ASIC.

***Index Terms*—** High Level Synthesis, High Level Transforms, algorithm transforms, software optimizations, softcore, FPGA, ASIC, power consumption, energy optimization, motion detection.

1. INTRODUCTION

High Level Synthesis (HLS) for Systems on Chip is a challenging way to cut off development time while assuming a good level of performance. The latest version of HLS tools integrates software optimizations that are coming from the optimizing compiler area [1] like *loop-unrolling*, *software pipelining* and using the *polyhedral model* to improve loop scheduling.To further improve current performance, tools should integrate the semantic of an application domain [2] and the related algorithm transforms [3].

This paper evaluates the impact of such algorithm transforms or high level transforms (HLT) for ASIC and softcore on FPGA. More and more commercial or academic HLS tools are available [4] like LegUp [5] or Gaut [6]. We have chosen Catapult-C as it is the tool used by ST Microelectronics for its synthesis farm. The Sigma-Delta algorithm ($\Sigma\Delta$) has been selected as it was specially designed for embedded systems and it is one of the best mono-modal algorithms [7] for motion detection. A comparison of other methods can be found in [8]. It has been ported on GPP, parallel artificial retina [9], embedded ARM processor [10] and FPGA [11]. So it assets as a reference algorithm for robust motion detection with low complexity. The impact of HLT has been evaluated on a softcore with instruction customization and on an ASIC with ST 65-nm CMOS technology. To improve efficiency, HLT are combined on each target with software optimizations.

2. MOTION DETECTION ALGORITHM

2.1. Sigma-Delta algorithm

The basic principle of the $\Sigma\Delta$ algorithm is to estimate the parameters of the background using $\Sigma\Delta$ modulation. Considering a time-varying signal f_t (continuous or discrete), we estimate a discrete signal d_t by quantizing the time indexes $\{t_i\}_{i\in\mathbb{N}}$, and then performing at every time index i the following update formulas:

If $d_{t_{i-1}} < f_{t_i}$ then $d_{t_i} = d_{t_{i-1}} - \varepsilon$ else $d_{t_i} = d_{t_{i-1}} + \varepsilon$ where ε is the discretization step (least significant bit) of d_t. In $\Sigma\Delta$ background subtraction, the input signal is the value of every pixel over time I_t, from which we compute the first $\Sigma\Delta$ background estimator M_t. Then the values of the absolute differences $|M_t - I_t|$ are used to compute the second $\Sigma\Delta$ background estimator V_t, which is a parameter of dispersion.

Algorithm 1: $\Sigma\Delta$ algorithm

```
1  foreach pixel $x$ do    [step #1: $M_t$ estimation]
2      if $M_{t-1}(x) < I_t(x)$ then $M_t(x) \leftarrow M_{t-1}(x) + 1$
3      if $M_{t-1}(x) > I_t(x)$ then $M_t(x) \leftarrow M_{t-1}(x) - 1$
4      otherwise $M_t(x) \leftarrow M_{t-1}(x)$
5  foreach pixel $x$ do    [step #2: $O_t$ computation]
6      $O_t(x) = |M_t(x) - I_t(x)|$
7  foreach pixel $x$ do    [step #3: $V_t$ update]
8      if $V_{t-1}(x) < N \times O_t(x)$ then $V_t(x) \leftarrow V_{t-1}(x) + 1$
9      if $V_{t-1}(x) > N \times O_t(x)$ then $V_t(x) \leftarrow V_{t-1}(x) - 1$
10     otherwise $V_t(x) \leftarrow V_{t-1}(x)$
11 foreach pixel $x$ do    [step #4: $\hat{E}_t$ estimation]
12     if $O_t(x) < V_t(x)$ then $\hat{E}_t(x) \leftarrow 0$ else $\hat{E}_t(x) \leftarrow 1$
```

In the basic version (Algo. 1), the $\Sigma\Delta$ background M_t and $\Sigma\Delta$ variance V_t are updated every frame, according to the comparison with the current image I_t and the current absolute difference O_t respectively. N is an amplification factor for V_t, allowing to compute the motion label $\hat{E}_t$ by simply comparing O_t and V_t (typical values of N are between 1 and 4).

As shown in [7], one can improve the robustness of the algorithm while keeping the complexity low with a two level processing algorithm combined with a conditional update and a Zipfian law for the update frequency. But the major improvement is done by a morphological post-processing. Figure 2 focuses on the impact of a 3×3 morphological opening. The next step to improve the segmentation robustness is to include a colorimetric model [12] to have a better segmentation between the objects and their projected shadow (around the man's feet in the figure).

2.2. Morphological post-processing

The 3×3 opening is the combination of a 3×3 erosion with a 3×3 dilation. As these two operators have the same complexity and the same mathematical property (idempotence), we focus on the implementation of only one operator. Let us define $\oplus$ the operator min used for the erosion and max for the dilation. Note that for binary images, these operators are respectively replaced by the Boolean operators AND and OR (our case here).

Algorithm 2: 1-pass implementation of the 3×3 morphological filter with the 2D-filter corresponding to equation (1), with explicit use of registers (Reg version)

```
Input: image $X$ of size $(n+2) \times (n+2)$
Output: image $Y$ of size $n \times n$
1 for $i = 1$ to $n - 1$ do
2     for $j = 1$ to $n - 1$ do
3         $a_0 \leftarrow X(i-1, j-1), b_0 \leftarrow X(i-1, j), c_0 \leftarrow X(i-1, j+1)$
4         $a_1 \leftarrow X(i, j-1), b_1 \leftarrow X(i, j), c_1 \leftarrow X(i, j+1)$
5         $a_2 \leftarrow X(i+1, j-1), b_2 \leftarrow X(i+1, j), c_2 \leftarrow X(i+1, j+1)$
6         $r \leftarrow a_0 \oplus b_0 \oplus c_0 \oplus a_1 \oplus b_1 \oplus c_1 \oplus a_2 \oplus b_2 \oplus c_2$
7         $Y(i, j) \leftarrow r$
```

The classical problem of the borders processing is addressed by the use of Iliffe arrays [13] based on offset addressing that allows the programmer to allocate images with negative indexes like$[0 - r : (n-1) + r] \times [0 - r : (n-1) + r]$, with r the *radius* of the kernel: for a $k \times k$ kernel, $k = 2r + 1$.

3. HIGH LEVEL TRANSFORMS AND SOFTWARE OPTIMIZATIONS FOR SOFTCORE AND ASIC

Optimizations can be classified according to three categories:

– High Level Transforms,
– Software optimizations, usually done by a compiler, but that can also be applied manually,
– Hardware and architectural optimizations.

HLT are algorithmic transforms based on optimizations belonging to an application domain – like image and signal processing – and are related to operator properties to reduce the algorithm complexity. Examples of such transforms are filter separation and factorization. In the next section, we will focus on the separation and reduction of morphological operators.

The usual software optimizations are loop unrolling, and more generally loop transforms, register rotation, register scalarization and software pipelining. Depending on the architecture, these optimizations will have an important or a small impact on performance.

Finally, the architectural optimizations depend on the architecture – here the RISC model. They take into account the architectures properties to improve the execution of the algorithm. That is to improve the pipeline efficiency (by increasing the amount of independent computation and reducing the pipeline stalls) and to improve cache performance (by reducing cache misses but more globally memory accesses: the cache cannot miss a memory access that does not exist!).

Another improvement is to use SIMD instructions [14] for a software programmable processor, or to generate a SIMD accelerator for a softcore processor, as it is known to be very efficient for low level image processing (with regular computations) [9].

Concerning customization (modifying an architecture), there are three levels of customization: 1) instruction customization, 2) function customization and 3) full-custom ASIC.

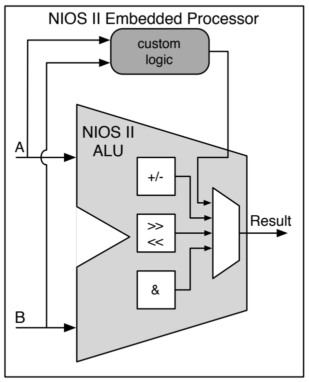

Fig. 1. Instruction customization for NIOS II

Customizing instructions consists in designing instructions that are missing in the current processor instruction set. Softcore processors offer the opportunity to add useful and efficient instructions for an application domain, while keeping the processor complexity low. Two commonly missing instructions for general purpose processors are `min` and

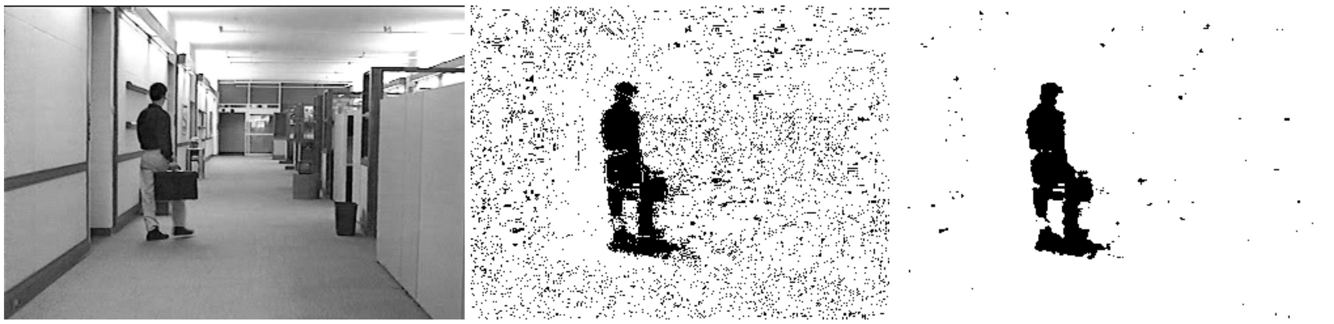

Fig. 2. Example of $\Sigma\Delta$ processing. Left: original image of Hall sequence, middle: $\Sigma\Delta$ without post-processing, right: $\Sigma\Delta$ with 3×3 morphological opening (erosion + dilation).

`max` that avoid using conditional branches and consequently branch mispredictions.

Instruction customization is user-friendly. For the C compiler, a new instruction designed with VHDL or Verilog is declared with a pragma like `#define MIN(a,b) _builtin_custom_inii(0,a,b)` for NIOS II. So it is directly recognized by the compiler as an existing instruction or a reserved keyword of the language.

From an electronic point of view, the piece of custom logic is interfaced with the input and output busses of the ALU: data read from the register bank are rerouted to the custom logic and once the computation is done (in one or more cycles) the result is available at the ALU output (Fig. 1). In this paper, the SOPC Builder from Altera has been used.

If instruction customization can be considered as *fine granularity*, function customization corresponds to an *intermediate granularity*. It consists in creating an external hardware accelerator. The accelerator is connected to the processor through a set of dedicated ports or busses, the data are sent into specialized registers (or FIFO) and the accelerator is activated through an interrupt. Depending on the architecture, the result is sent back to the processor or written into memory. A more general view of function customization is to consider the generated hardware like any external hardware that could read/write data from/to the memory with a DMA or a dedicated bus.

The advantage of such a customization is that the computation is combinatory and no longer controlled by the processor clock: if there are many small operations to perform like additions (in opposition to a multiplication) or binary operations, more than one operation can be achieved during one cycle. In some cases, considering a low frequency processor, and then low frequency synthesis, many computations can be done in one cycle. The main drawback of such customization is the communication delay from and to the processor: the communication duration should be smaller than the time saved by the hardware computation compared to the software computation. As the computation load for one loop iteration is low (for both $\Sigma\Delta$ and morphological operators), hardware accelerators could not be efficient here and will not be used and detailed in this paper.

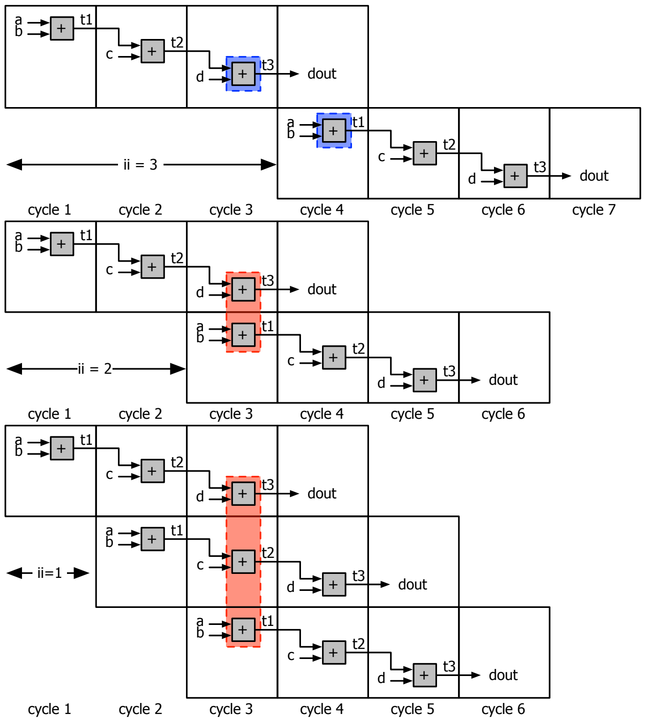

Fig. 3. Software pipelining and *initiation interval*. Top: ii=3 $\Rightarrow$ one adder needed, middle: ii=2 $\Rightarrow$ two adders needed, bottom: ii=1 (fully pipelined) $\Rightarrow$ three adders needed. Example from HLS Blue Book [15]

The most advanced level of customization is the full custom design than can be achieved with HLS tools for Electronic System Level (ESL) methodology. Catapult-C from Calypto Design Systems has been used for the full-custom approach. The typical way of using such a tool is to provide a C or C++ source code and to set the clock frequency to be used. If almost all parameters are automatically explored by the tool, at least one parameter can be set by the user: the *initiation interval* (ii). That is the latency, in cycles, between the start of two iterations of a loop. Let us consider the following computation $t = a+b+c+d$ with classic 2-input adders and assume that the duration of one addition is one cycle. This example comes from HLS Blue Book [15].

In the first case, one wants one output written every 3 cycles (Fig. 3, top with ii=3). In that case there is no overlap of any operation and only one adder is required. With a constraint of one output every 2 cycles (Fig. 3, middle with ii=2) two adders are required as there are two additions on cycle 3: the first one computes $t_3 = t_2 + d$ from the first lane, and the second to compute $t_1 = a + b$ from the second lane. With a hard constraint of one output every cycle (Fig. 3, bottom with ii=1) three adders are required on cycle 3: the first one computes $t_3 = t_2 + d$, the second one computes $t_2 = t_1 + c$ and the third one computes $t_1 = a + b$.

Note that is the electronic instance of *software pipelining*, the most important optimization for VLIW processors. Depending on ii and the algorithm structure, one can have a direct impact on the size and the performance of the circuit: with smaller ii, the circuit is faster and bigger; with larger ii, the circuit is slower and smaller.

For processors (softcore and hardcore), the main problem is the memory bandwidth, as many algorithms (and the considered algorithms belong to that class) are memory bounded. So the first target is to reduce the amount of memory accesses. Then the second target is to reduce the amount of computations and the amount of hazards (comparisons that can stall the pipeline).

In the next section we first present high level transforms, then software and architectural optimizations.

3.1. High Level Transforms

As $\Sigma\Delta$ is a very simple pixel-to-pixel algorithm, there is no HLT to apply. So we focus on the morphological operators.

The classical software optimization is *Loop-Unrolling* but it suffers from one drawback: the code size increases by the order of unrolling. So we prefer the *Register-Rotation* combined with *scalarisation* (to put temporary results into registers). In algorithm 3, the pixel of the left and central column are loaded into registers before the loop (lines 4-6). After the computation registers are rotated (lines 14-16). Compared to the initial algorithm with 9 LOADs, there are only 3 LOADs in that version. The arithmetic complexity remains the same: 8 operations (named OP in the following). *Register-Rotation* leads to two versions: Rot_1 (Algo. 3) in 1 pass, Rot_2 (Algo. 4) in two passes.

Taking into account the *idempotent* property of the morphological operators, the 2D structuring element $SE_{3\times3}$ can be replaced by two 1D elements (Eq. 1). This algorithmic transformation called Rot_2 (Algo. 4) reduces the complexity (4 OP instead of 8 previously) but increases the number of

Algorithm 3: 1-pass implementation of the 3×3 morphological filter with *Register Rotation*, Rot_1 version

```
   Input: image $X$ of size $n \times n$
   Output: image $Y$ of size $n \times n$
1  for $i = 1$ to $n-1$ do
2      [preload the first two columns of each line]
3      $j \leftarrow 1$
4      $a_0 \leftarrow X(i-1,j-1), b_0 \leftarrow X(i-1,j)$
5      $a_1 \leftarrow X(i,j-1), b_1 \leftarrow X(i,j)$
6      $a_2 \leftarrow X(i+1,j-1), b_2 \leftarrow X(i+1,j)$
7      for $j = 1$ to $n-1$ step 3 do
8          $c_0 \leftarrow X(i-1,j+1)$
9          $c_1 \leftarrow X(i,j+1)$
10         $c_2 \leftarrow X(i+1,j+1)$
11         $r \leftarrow a_0 \oplus b_0 \oplus c_0 \oplus a_1 \oplus b_1 \oplus c_1 \oplus a_2 \oplus b_2 \oplus c_2$
12         $Y(i,j) \leftarrow r$
13         $a_0 \leftarrow b_0, b_0 \leftarrow c_0$    [RR of the first line]
14         $a_1 \leftarrow b_1, b_1 \leftarrow c_1$    [RR of the second line]
15         $a_2 \leftarrow b_2, b_2 \leftarrow c_2$    [RR of the third line]
```

LOADs to 6. The main problem is that such an algorithm requires two passes on the image and then, if the image is too large to entirely fit in the cache, it generates cache misses. So Rot_1 version will be prefered to Rot_2.

$$SE_{3\times3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \quad (1)$$

Algorithm 4: 2-pass implementation of the 3×3 binomial filter with two 1-D filters of equation 1, Rot_2 version

```
   Input: image $X$ of size $n \times n$
   Output: image $Y$ of size $n \times n$
1  for $i = 1$ to $n-1$ do
2      for $j = 1$ to $n-1$ step 3 do
3          $x_0 \leftarrow X(i-1,j), x_1 \leftarrow X(i,j), x_2 \leftarrow X(i+1,j)$
4          $r \leftarrow x_0 \oplus x_1 \oplus x_2$
5          $Y(i,j) \leftarrow r$
6  for $i = 1$ to $n-1$ do
7      for $j = 1$ to $n-1$ step 3 do
8          $a \leftarrow X(i,j-1), b \leftarrow X(i,j), c \leftarrow X(i,j+1)$
9          $r \leftarrow a \oplus b \oplus c$
10         $Y(i,j) \leftarrow r$
```

By introducing another optimization, we can both factorize the computations and reduce the number of memory accesses. The two passes of the 1D-filter on the image can be combined within a single pass. First, the result of the first 1D-filter is stored in a register. This transformation is called a *reduction*. In our case, it is a column-wise *reduction*: instead of memorizing 6 pixels (Algo. 3, lines 4-6), we compute the *reduced* value by column (Algo. 5, lines 5 & 6). Then the second operator is directly applied to the *reduced* values (Algo. 5, line 12). In that version there are only 3 LOADs and 4 OPs. All the complexity figures are summarized in table 1 where MV stands for a *move* (to copy one register to another one) and AI represents the arithmetic intensity (ratio between arithmetic operators and memory accesses).

Algorithm 5: 1-pass implementation of the two seperated 1D operators with *reduction*, Red version

```
Input: image $X$ of size $n \times n$
Output: image $Y$ of size $n \times n$
1  for $i = 1$ to $n-1$ do
2      $a_0 \leftarrow X(i-1, j-1), b_0 \leftarrow X(i-1, j)$
3      $a_1 \leftarrow X(i, j-1), b_1 \leftarrow X(i, j)$
4      $a_2 \leftarrow X(i+1, j-1), b_2 \leftarrow X(i+1, j)$
5      $\mathbf{r_a} \leftarrow a_0 \oplus a_1 \oplus a_2$   [reduction of the first column]
6      $\mathbf{r_b} \leftarrow b_0 \oplus b_1 \oplus b_2$   [reduction of the second column]
7      for $j = 1$ to $n-1$ do
8          $c_0 \leftarrow X(i-1, j+1)$
9          $c_1 \leftarrow X(i, j+1)$
10         $c_2 \leftarrow X(i+1, j+1)$
11         $r_c \leftarrow c_0 \oplus c_1 \oplus c_2$   [reduction of the third column]
12         $r \leftarrow r_a \oplus r_b \oplus r_c$   [applying the horizontal operator]
13         $Y(i, j+0) \leftarrow r$
14         $r_a \leftarrow r_b$   [rotation of the reduced registers]
15         $r_b \leftarrow r_c$
```

version	OP	LD + ST	MV	AI
Reg (1-pass of 2D-op)	8	9+1=10	0	0.8
Rot_1 (1-pass of 2D-op)	8	3+1=4	6	2.0
Rot_2 (2-pass 1D-op)	4	2(3+1)=8	0	0.5
Red (1-pass 1D-op)	4	3+1=4	2	1.0

Table 1. Morphological operator complexity and arithmetic intensity

3.2. Low Level Transforms: software and architectural optimizations

3.2.1. Software optimizations

In that section we present two optimizations answering two questions. The first one addresses C code refactoring for Catapult-C: how to write efficient C code ? The second one addresses softcores: what can be done to reduce the execution time when faced to many if-then-else tests ?

Algorithm 6: *Basic* double *if-then-else*

```
1  if $r < x$ then $r \leftarrow r + 1$
2  if $r > x$ then $r \leftarrow r - 1$
```

Let us focus on the double *if-then-else* of algorithm 1, lines 2 & 3 (and also lines 8 & 9). As there are 3 cases (incrementation, decrementation, nothing), it requires two Boolean tests (Algo. 6). What is usually done to save a comparison (in 50 % of the cases) is to do a *nested if-then-else* (Algo. 7). but the drawback of such optimization is the *serialization* of the tests.

Algorithm 7: *Nested* double *if-then-else*

```
1  if $r < x$ then
2      $r \leftarrow r + 1$
3  else
4      if $r > x$ then
5          $r \leftarrow r - 1$
```

Another way to reformulate the double *if-then-else*, is to separate the comparisons from the computations. In algorithm 8, the comparisons (that can be *nested*) only set the value of δ that is added at the end.

Algorithm 8: Double *if-then-else* with δ

```
1  $\delta \leftarrow 0$
2  if $r < x$ then $\delta \leftarrow +1$
3  if $r > x$ then $\delta \leftarrow -1$
4  $r \leftarrow r + \delta$
```

The final reformulation (Algo. 9) is more an *hack* than a software optimization, as it directly mixes arithmetic addition with comparisons. It relies on the fact that the result of a comparison is `0x0` or `0xFFFFFFFF` whether the result is *false* or *true* and that `0xFFFFFFFF` is also equal in 2-complement arithmetic to `-1`. In that case the expression $r \leftarrow r + 1$ becomes $r \leftarrow r - (-1)$ and finally $r \leftarrow r - lt$ (line 3). The case is similar for the other comparison.

The complexity of these 4 versions is summarized in the table 2 where CMP stands for a comparison ($<$ or $>$) and ADD for an addition or a subtraction. As the comparison is done with a subtraction, the *total* column is the sum of CMP and ADD, while the last column is the number of conditional branches associated to the *if-then-else*. As we assume that each comparison is completely random they are unpredictable even with a branch predictor:the number of mispredictions is proportional to the number of branches. As we can see, only the hack version has no conditional branches. The hack version has the largest complexity: four ADDs. Its main advantage is that there is no more *if-then-else*, so we can avoid pipeline stalls associated to branch mispredictions. The benchmark section will present the impact of such a style of coding for Catapult-C.

Algorithm 9: Hacked *if-then-else* with 2-complement

```
1  $lt \leftarrow (r < x)$
2  $gt \leftarrow (r > x)$
3  $r \leftarrow r - lt$
4  $r \leftarrow r + gt$
```

version	CMP	ADD	total	conditional BRANCH
basic	1 + 1	0.5 + 0.5	3	1+1=2
nested	1 + 0.5	0.5 + 0.25	2.25	1+0.5=1.5
delta	1 + 1	0.5 + 0.5	3	1+1=2
hack	1 + 1	1 + 1	4	**0**

Table 2. Complexity of *if-then-else* versions

Algorithm 10: *Masked* double *if-then-else*

```
1 $lt \leftarrow (r < x)$
2 $gt \leftarrow (r > x)$
3 $inc \leftarrow (+1)$ AND $lt$
4 $dec \leftarrow (-1)$ AND $gt$
5 $r \leftarrow r + inc + dec$
```

Considering softcores, we evaluate the impact of instruction customization (Fig. 1) for the NIOS II. We create two new instructions: `r=lt_inc(r,x)` and `r=gt_dec(r,x)` that replace the two considered *if-then-else* tests. One can perform the increment or the decrement in only one instruction `r=inc_dec(r,x)` as the VHDL synthesis implements the two comparisons in parallel.

3.2.2. *Architectural optimizations*

Finally one can perform yet another software optimization on the softcore. As the cache of the processor is direct-mapped, miss conflicts are higher than with set-associative caches. Reducing the number of memory accesses reduce the number of instructions and may reduce the number of miss conflicts.

The basic solution is to replace four consecutive 8-bit LOADs by one 32-bit LOAD. That corresponds to a loop unroll of 4. Once a 32-bit data is loaded, it is unpacked (with classical shifts and masks instructions) into four 8-bit data that are processed by the previous algorithm. Then the four results are packed into a 32-bit data and stored by into memory. For a softcore processor 32-bit specialized instructions can be designed to perform Sub-Word Parallelism (SWP). As $\Sigma\Delta$ operator is pixel-wise, such a design is straightforward. For the morphological operator SWP are unnecessary as 32-bit `OR` and `AND` exist (the gray-level version will required four 8-bit `MAX` and `MIN`) but new instructions that merge two 32-bit registers should be designed (see [9] for `vec_left` and `vec_right` macros to extract unaligned vectors).

4. BENCHMARKS: RESULTS AND ANALYSIS

4.1. Softcore processors

For softcore, an Altera board with a NIOS II onto a Stratix2 (EP2S60F672C5ES) was used. . The clock frequency is 50 MHz. The 32-KB data cache is direct-mapped with 32-B lines. To avoid the possibility of systematic eviction when two different arrays are mapped into the same cache lines, padding was used. For softcores, four configurations were benchmarked (Tab. 3 & 4):

- 8-bit × SW: basic 8-bit code, without accelerator
- 8-bit × HW: basic 8-bit code, with 8-bit hardware accelerator
- 32-bit × SW: 32-bit SWP code, without accelerator (8-bit computations with software pack and unpack operations)
- 32-bit × HW: 32-bit SWP code, with SWP hardware accelerator (32-bit computations done in hardware)

The metrics used are the average number of cycles per pixel (*cpp*) and energy per pixel.

Concerning $\Sigma\Delta$ we can observe that 32-bit memory accesses are efficient only when coupled with instruction customization. The reason is the cost to perform pack/unpack steps is very high: 3 `SHIFTs` and 3 `ANDs` per unpack and 3 `SHIFTs` and 3 `ORs` per pack. That is a total of 18 `SHIFTs` and 18 Boolean instructions! When custom instructions are used, there is no more such steps and the 32-bit version is fast. With 8-bit custom instructions, there is no more pipeline stall due to *if-then-else*. That provides a speedup of ×1.23 versus the software version. For the 32-bit version, compared to 8-bit version, custom instructions provide a super-linear speedup of ×7.35 that comes from an efficient usage of both the pipeline and the cache. Combined together the total speedup is ×9.06.

version	SW	HW	gain SW/HW
$\Sigma\Delta$ 8-bit	47.1	38.2	× 1.23
$\Sigma\Delta$ 32-bit	109.0	5.2	× 20.96
32-bit gain	× 0.43	× 7.35	×**9.06**

Table 3. *cpp* of $\Sigma\Delta$ algorithm on softcore NIOS II processor

Concerning the morphological operators (Tab. 4), we can make the same observation about 32-bit accesses: they are efficient if and only if there is no extra instructions like pack/unpack steps to process data. Regarding instruction customization, there is no gain in 8-bit versions, and the gain of 32-bit versions is not pertinent. The reason is that instructions (software or hardware) are scheduled by clock, so only one instruction can be executed per cycle, and that such instruction must enforce a *2-operands-only* form: there is no way to perform two Boolean operations inside one instruction. So Boolean instructions (`AND` or `OR`) go at the same speed, by software or by hardware. If we now focus on HLT, we can see that these optimizations are always efficient whatever the other optimizations/transformations: the gain is approximatively ×1.5 for all versions (we do not consider the 32-bit software version for the previously explained reasons). Combined together we reach a speedup of ×7.85 for 32-bit custom instructions with HLT on NIOS II compared to 8-bit software version without HLT.

While the hardware cost of the NIOS processor is 1,703 LUTs, 1,354 registers, 332-Kb block memory bits and 82 4Kb RAM blocks, the cost of the custom instructions is only 453 LUTs for both the 8-bit and 32-bit versions, which is a quite small overhead.

version	SW	HW	gain SW/HW
8-bit *Reg*	21.9	20.6	$\times$ 1.06
8-bit Rot_1	22.0	21.0	$\times$ 1.05
8-bit *Red*	14.1	14.0	$\times$ 1.01
HLT gain (*Reg*/*Red*)	$\times\mathbf{1.55}$	$\times\mathbf{1.47}$	-
32-bit *Reg*	31.4	5.9	$\times$5.32
32-bit Rot_1	46.6	6.3	$\times$7.40
32-bit *Red*	17.15	4.0	$\times$4.29
HLT gain (*Reg*/*Red*)	$\times\mathbf{1.83}$	$\times\mathbf{1.48}$	-
total gain (*Reg*8 / *Red*32)	$\times$1.28	$\times\mathbf{5.15}$	$\times\mathbf{5.48}$

Table 4. *cpp* of morphologicial operator on NIOS II processor

4.2. Full-custom ASIC

For the full-custom approach, the ST 65 nm library with a dual-port memory was used with Catapult-C. The evaluation of the power consumption and the area was done with Synopsys Design Compiler without activating the capabilities of Catapult-C to reduce the total power consumption generating local/global clock gating glue as we assume that the ASIC is always running. Concerning ASIC, the *cpp* is quite equal to the *initiation interval*. It is in fact a bit smaller because of the *setup time* t_{su}: we have $t_{exe} = ii/F - t_{su}$ and $cpp = t_{exe} \times F/n < ii$.

Table 5 presents the results in term of energy/pixel for refactoring the *if-then-else* tests of $\Sigma\Delta$. For the four versions, the best performance is always reached for ii=1, and the average gain of the delta version (close to the hack) is $\times 1.14$. If this configuration is compared to the basic configuration that leads to the smallest area, the average gain in energy is higher than $\times 4$: we cannot independently optimize for energy or for size.

freq (MHz)	200	400	600	800	average
basic	1.75	1.52	1.47	1.87	1.65
nested	1.63	1.87	2.10	2.48	2.02
hack	1.40	1.41	1.59	1.50	1.48
delta	1.40	1.40	1.49	1.50	1.45
gain	$\times$1.25	$\times$1.09	$\times$0.99	$\times$1.25	$\times\mathbf{1.14}$

Table 5. Energy (pJ/pixel) of $\Sigma\Delta$ algorithm on 65 nm ASIC with ii=1

For the morphological operator, HLT have a major impact on the efficiency. Let us call "best" and "smallest" the configurations associated to the best energy consumption, and the smallest area (Tab. 6 & 7). Let us also call $ii = 0$ the combinatory version. As the basic version (Reg) requires 9 LOADs (Tab. 6) we need 9 cycles to perform all the LOADs with a single-port RAM and $\lceil 9/2 \rceil = 5$ cycles with a dual-port RAM. For the same reason, the minimum number of cycles for Rot and Red versions (3 LOADs) is 2 cycles. That is very important, as for all the explored configurations, the best energy was reached for the smallest ii. We can see that the average gain due to HLT is $\times\mathbf{2.90}$. Moreover, with a single-port RAM, the gap between Reg and Red versions would be even greater, as the energy increases with the ii (Tab. 8). Finally, if we compare the configuration of the smallest area without HLT to the best Red, the average gain reaches $\times\mathbf{3.49}$.

freq (MHz)	200	400	600	800	average
smallest *Reg* (*ii*=0)	6.45	6.67	7.44	7.79	7.09
best *Reg* (*ii*=5)	5.49	5.76	6.44	5.87	5.89
best Rot_1 (*ii*=2)	2.47	2.78	3.14	2.95	2.84
best *Red* (*ii*=2)	1.80	2.02	2.25	2.05	2.03
best *Reg* / best *Red*	$\times$3.05	$\times$2.85	$\times$2.86	$\times$2.86	$\times\mathbf{2.90}$
smallest *Reg* / best *Red*	$\times$3.58	$\times$3.30	$\times$3.31	$\times$3.80	$\times\mathbf{3.49}$

Table 6. Energy (pJ/pixel) of the morphological operator on 65 nm ASIC with best ii for Reg, Rot_1 and Red versions

freq (MHz)	200	400	600	800	average
smallest *Reg* (*ii*=0)	2893	2893	2893	2986	2916
best *Reg* (*ii*=5)	3206	3208	3206	3030	3163
ratio best / smallest	1.11	1.11	1.11	1.01	1.08
smallest Rot_1 (*ii*=4)	2905	2908	2923	2847	2896
best Rot_1 (*ii*=2)	3534	3534	3563	3443	3519
ratio best / smallest	1.22	1.22	1.22	1.21	1.22
smallest *Red* (*ii*=4)	2374	2378	2400	2408	2390
best *Red* (*ii*=2)	2685	2685	2714	2616	2675
ratio best / smallest	1.13	1.13	1.13	1.09	**1.12**
smallest *Reg* / best *Red*	1.08	1.08	1.07	1.14	**1.09**

Table 7. Area (μm^2) of the morphological operator on 65 nm ASIC with best ii for Reg, Rot_1 and Red versions: the smallest area and the area associated to the best energy

We can perform the same analysis for the area (Tab. 7). There is an average area increase of 8%, 22% and 12% for each level of HLT optimization (Reg, Rot and Red). But if we compare the smallest area without HLT to the area associated to the best Red energy, we can see that best Red has a smaller area than the configuration without optimization (smallest Reg). HLT has also a (small) impact on area.

5. CONCLUSION AND FUTURE WORK

We have shown that high level transforms (HLT) are very efficient for both softcore on FPGA and ASIC. For softcore,

freq	200	400	600	800	average
combinatory logic	2.95	3.05	3.60	3.46	3.27
ii **= 2**	**1.80**	**2.02**	**2.25**	**2.05**	**2.03**
ii = 3	2.56	2.81	3.17	2.89	2.86
ii = 4	2.90	3.01	3.57	3.46	3.24
ii = 5	3.58	3.70	4.08	4.23	3.90
ii = 6	4.25	4.38	5.13	5.00	4.69
ii = 7	4.94	5.11	5.90	5.85	5.45
ii = 8	5.61	5.79	6.76	6.62	6.20

Table 8. Impact of ii on the energy (pJ/pixel) for Red version of the morphological operator on ASIC: the smaller the better

HLT provide a speedup of $\times 1.5$ with instruction customization. For ASIC, the software optimizations are no more required as they are done by Catapult-C. Then, by reducing the number of memory accesses, HLT allow synthesis for a smaller value of the *initiation interval*: HLT provide a speedup of $\times 2.5$. And as energy is related to the *initiation interval*, we have an average gain of $\times 2.90$. Usually one has to choose between speed and low power consumption. With the combination of HLT and HLS, we do not have to choose: the ASIC is both faster and greener!

In future works, we will implement HLT through algorithmic skeletons [16] to make the whole process (algorithm transformation and synthesis) fully automatic. For ASICs, we will implement banked single-port RAM to try to reach a 1-cycle throughput. For softcores on FPGA we will evaluate other specialized instructions and the replacement of a direct-mapped cache by a set-associative cache like those available at OpenCores. Finally, for external accelerators we will implement a color-version of the algorithm to increase algorithm robustness and to provide more numbers to crunch for the accelerator.

6. REFERENCES

[1] R. Allen and K. Kennedy, Eds., *Optimizing compilers for modern architectures: a dependence-based approach*, chapter 8,9,11, Morgan Kaufmann, 2002.

[2] S. Le Beux, L. Moss, P. Marquet, and J.L. Dekeyser, "A high level synthesis flow using model driven engineering," in *Algorithm-Architecture Matching for Signal and Image Processing*. Springer, 2012, pp. 253–274.

[3] Markus Pueschel, Jos M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE: special issue on program generation, optimization and adaptation*, vol. 93,2, pp. 232–275, 2005.

[4] Wikipedia, "`http://en.wikipedia.org/wiki/High-level_synthesis`,".

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *FPGA11: international symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 33–36.

[6] B. Le Gal, E. Casseau, P. Bomel, C. Jego, N. Le Heno, and E. Martin, "High-level synthesis assisted rapid prototyping for digital signal processing," in *International Conference on Microelectronics*, 2004, pp. 746–749.

[7] L. Lacassagne, A. Manzanera, and A. Dupret, "Motion detection: fast and robust algorithms for embedded systems," in *International Conference on Image Analysis and Processing*, 2009, pp. 3265–3268.

[8] M. Piccardi, "Background subtraction techniques: a review," in *Conference on Systems, Man and Cybernetics*. IEEE, 2004, vol. 4, pp. 3099–3104.

[9] L. Lacassagne, A. Manzanera, J. Denoulet, and A. Mérigot, "High performance motion detection: Some trends toward new embedded architectures for vision systems," *Journal of Real Time Image Processing*, pp. 127–148, october 2008.

[10] S. L. Toral, M. Vargas, F. Barrero, and M. G. Ortega, "Improved sigma ? delta background estimation for vehicle detection," *Electronics letters*, vol. 45-1, pp. 851–863, 2009.

[11] M.M. Abutaleb, A. Hamdy, M.E. Abuelwafa, and E.M. Saad, "Fpga-based object extraction based on multimodal sigma-delta background estimation," in *International Conference on Computer Control*, 2008, pp. 1–7.

[12] M. Gouiffes, C. Collewet, C. Fernandez-Maloigne, and A. Trémeau, "A photometric model for specular highlights and lighting changes. application to feature points tracking.," in *International Conference on Image Processing*. IEEE, 2006, pp. 2117–2120.

[13] J.K. Iliffe, "The use of the genie system in numerical calculation," *Annual Review in Automatic Programming*, vol. 2, pp. 1 – 28, 1961.

[14] K. Diefendorff, P.K. Dubeyn, R. Hochsprungand, and H. Scales, "Altivec extension to powerpc accelerates media processing," *Micro*, vol. 20, 2, pp. 85–95, March-April 2000.

[15] M. Fingeroff and T. Bollaert, Eds., *High-Level Synthesis - Blue Book*, chapter 4, pp. 41–44, Mentor Graphic, 2010.

[16] T. Saidani, J. Falcou, C. Tadonki, L. Lacassagne, and Daniel Etiemble, "Algorithmic skeletons within an embedded domain specific language for the cell processor," in *PACT*, 2009, pp. 67–76.