High Level Tranforms to reduce energy consumption of signal and image processing operators

H Ye, Lionel Lacassagne, J Falcou, D Etiemble, L. Cabaret, O Florent

HIGH LEVEL TRANFORMS TO REDUCE ENERGY CONSUMPTION OF SIGNAL AND IMAGE PROCESSING OPERATORS

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

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

ABSTRACT

High Level Synthesis for Systems 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. We have evaluated recursive and non recursive filters for signal processing an morphological filters for image processing. We show that the impact of HLTs to reduce energy consumption is high : from $\times$3.4 for one 1D filter up to $\times$5.6 for cascaded 1D filters and about $\times$3.5 for morphological 2D filters.

Index Terms— High Level Synthesis, High Level Transforms, algorithm transforms, software optimizations, ASIC, power consumption, energy optimization, signal processing, image processing.

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 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. More and more commercial or academic HLS tools are available like LegUp [4] or Gaut [5]. We have chosen Catapult-C as it is the tool used by ST Microelectronics for its synthesis farm. Finite Impulse Response and Infinite Impulse Response filters have been chosen as being ubiquitous in signal processing, while morphological filters are also largely used in the image processing area. The first section presents Catapult-C and how to explore the design space configurations. The next sections describe a set of optimizations and their impact on FIR filters, IIR filters and morphological filters as well implementation details on the code generation process based on preprocessor meta-programming.

2. HIGH LEVEL SYNTHESIS TOOLS AND OPTIMIZATIONS

The optimizations can be classified according to three categories: HLT, software optimizations and hardware 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.The usual software optimizations like loop unrolling, register rotation and software pipelining are present in Catapult-C: it is able to unroll or pipeline loops. Moreover, it is able to detect a loop that perform register rotation and transform the loop into a set of register-to-register copy within a register file.

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 number of clock cycles between the start of two iterations of a loop. Let us consider the following computation $t = a + b + c + d$ (from [6]) with usual 2-input adders and assume that the duration of one addition is one cycle. In the first case, one wants one output written every 3 cycles (Fig. 1, top with ii=3). In that case, there is no overlap between the operations and only one adder is needed. To get one output every 2 cycles (Fig. 1, middle with ii=2, two adders are needed as two additions occur on cycle 3: the first one computes $t_3 = t_2 + d$ from the first lane, and the second computes $t_1 = a + b$ from the second lane. To get one output every cycle (Fig. 1, bottom with ii=1, three adders are needed 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 $t_1 = a + b$.

Notice that is the electronic instance of *software pipelining*, which is the most important optimization for VLIW processors. Depending on ii and the algorithm structure, one can have a direct impact on the size and performance of the

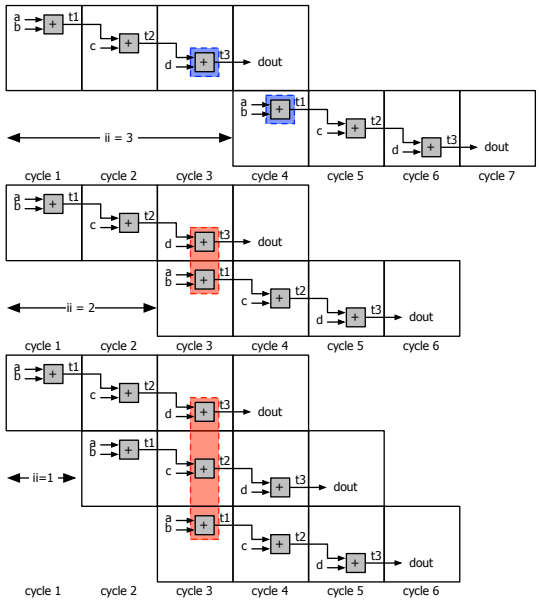

Fig. 1. Software pipelining and *initiation interval*. Top: ii=3 $\Rightarrow$ one adder needed, middle: ii=2 $\Rightarrow$ two adders needed, bottom: ii=1 (fully pipelined) $\Rightarrow$ three adders needed

circuit: with smaller ii, the circuit is faster and bigger; with larger ii, the circuit is slower and smaller.

For benchmarking, the ST 65-nm CMOS library was used with Catapult-C. The evaluation of the power consumption and the area was done with Synopsys Design Compiler – before place and route – 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. In that case, the energy is the product of the execution time by the (static + dynamic) power.

In the following, the small internal arrays are stored into register, while big external arrays are stored into memory. There are one memory per array, except for banked-memory where there are 3 (or 5) memory bank to enable multiple accesses. We assume a streaming behavior: a set of data is transferred into memory before an operator is called to process it. As operators are pipelined, the latency is equal to the value of ii.

3. FINITE IMPULSE RESPONSE FILTER

The Finite Impulse Response (FIR) filters are very common in signal processing. If data are represented by floating-point numbers, there are no special issue to address and the implementation is straightforward. The only issues are *absorption* and *cancellation* when the coefficients magnitude is very high. In our case, we consider integer data - typically - 8-bit and Q_8 computations. That implies three points: 1) the coefficients are multiplied by 2^8 before rounding, 2) the result of the computation is divided by 2^8 – shifted by 8 – and 3) in order to have *rounded* computations instead of *truncated* ones, we must add half the value of the division, that is 2^7. So considering a general 3-tap filter (Eq. 1), the associated implementation is given by the algorithm 1 with the rounding value $r = 128$. Notice that in all our examples, we do not provide the prolog or the epilog of a loop, in order to have small and compact examples (The Duff's device [7] can be used to remove the epilog but leads to a more tricky code).

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) \quad (1)$$

Algorithm 1: fixed-point FIR3 filter – Reg version

```
1 for i = 2 to n − 1 do
2     x0 ← X[i − 0], x1 ← X[i − 1], x2 ← X[i − 2]
3     y ← (b0 × x0 + b1 × x1 + b2 × x2 + r)/256
4     Y[i] ← y
```

3.1. FIR optimization

For FIR filters, there are two points to address. The first one is to compare the impact of the hardware and software optimizations on the synthesis of one filter. The second one is the optimization of *cascaded* FIR filters.

The usual drawback of such a filter is its low complexity: there is one MAC (Multiply-Accumulate) operation for every LOAD. So the FIRs are generally memory bounded (except on VLIW DSPs that are specially designed for signal processing [8] and are able to perform two memory accesses per cycle). So, optimizing means reducing the LOAD duration (the HW optimization) or the number of LOADs (the SW one). We have evaluated four kinds of memory: 1) the Single Port memory with 1 READ or 1 WRITE per cycle, 2) the SP RW memory with 1 READ *and* 1 WRITE per cycle, 3) the Double Port memory: two SP memories, with 2 READs per cycle and 4) the banked memory with 3 (or 5) interleaved SP memory banks, allowing 3 (or 5) accesses per cycle. A small finite state machine automaton selecting the correct memory bank during the filtering has been designed to handle circular adressing. For the software part, we can perform *Register Rotation* or *Loop Unrolling* (Algo. 2 and 3). We omit the prolog and epilog parts to keep the algorithms as small as possible.

Algorithm 2: FIR3 filter – Rot version

```
1 x2 ← X[0], x1 ← X[1]
2 for i = 2 to n − 1 do
3     x0 ← X[i − 0]
4     y ← (b0 × x0 + b1 × x1 + b2 × x2 + r)/256
5     Y[i] ← y
6     x2 ← x1, x1 ← x0 [Registers Rotation]
```

memory	SP	SR RW	DP	3×SP	SP	SP	SP RW	DP
optimization	*Reg*	*Reg*	*Reg*	*Reg*	*Rot*	*LU*	*LU*	*LU*
1× *FIR3*								
area(*best A*)	4237	4266	4562	4949	4458	6120	6240	7027
area(*best E*)	4671	4751	5957	7537	5047	10692	11038	12294
energy(*best A*)	**11.68**	12.08	12.54	13.80	11.92	28.24	28.49	26.18
energy(*best E*)	6.52	6.62	5.47	**3.41**	**1.99**	14.07	14.47	10.61
ii(*best E*)	3	3	2	1	1	3	3	2
two cascaded FIR3								
area(*best A*)	6393	6482	6542	7498	6574	9913	10155	11155
area(*best E*)	8711	9206	10797	10360	7370	15828	16895	16237
energy(*best A*)	12.88	13.11	13.95	12.86	8.79	24.18	25.05	19.93
energy(*best E*)	7.56	5.87	4.34	6.69	**2.82**	15.79	16.45	10.89
ii(*best E*)	3	3	2	3	1	3	3	2
two pipelined FIR3								
area(*best A*)	5888	5943	6207	6543	5715	9625	10039	10400
area(*best E*)	7619	7547	8639	12385	9317	18329	19086	20726
energy(*best A*)	**20.05**	21.78	21.33	23.28	23.11	64.54	65.56	66.98
energy(*best E*)	9.98	10.03	8.11	**5.55**	**3.65**	21.31	22.08	17.42
ii(*best E*)	3	3	2	1	1	3	3	2
memory	SP	SR RW	DP	**5×SP**	SP	SP	SP RW	DP
optimization	Reg	Reg	Reg	Reg	Rot	LU	LU	LU
two fused FIR3 = one FIR5								
area(*best A*)	5563	5619	5683	7121	5990	17968	17563	17330
area(*best E*)	6056	6198	7670	12513	8189	26913	28100	30441
energy(*best A*)	**22.11**	22.72	19.89	27.23	22.01	107.03	107.13	118.48
energy(*best E*)	13.74	14.05	10.52	**5.59**	**3.19**	48.21	50.98	34.14
ii(*best E*)	5	5	3	1	1	5	5	3

Table 1. FIR average area and energy

Algorithm 3: fixed-point FIR3 filter – LU version

$x_2 \leftarrow X[0], x_1 \leftarrow X[1]$
for $i = 2$ **to** $n-1$ **do**
  $x_0 \leftarrow X[i-0]$
  $y_0 \leftarrow (b_0 \times x_0 + b_1 \times x_1 + b_2 \times x_2 + r)/256$
  $Y[i+0] \leftarrow y_0$
  $x_2 \leftarrow X[i+1]$
  $y_1 \leftarrow (b_0 \times x_2 + b_1 \times x_0 + b_2 \times x_1 + r)/256$
  $Y[i+1] \leftarrow y_1$
  $x_1 \leftarrow X[i+2]$
  $y_2 \leftarrow (b_0 \times x_1 + b_1 \times x_2 + b_2 \times x_0 + r)/256$
  $Y[i+2] \leftarrow y_2$

3.2. Results and analysis

Let us call *best A*, the configuration that minimizes the areas and *best E* the configuration that minimizes the energy.

The first part of the table 1 provides the average area and energy of all the best configurations for synthesis frequency varying from 200 to 800 MHz by step of 200 MHz.

As area and energy figures are very close for a given ii, they have been replaced by their average in order to provide all the results within only one table. the table also provides the *initiation interval* (ii). The best HW optimization is the $3 \times SP$ version with a banked memory. The energy is divided by $\times 3.43$ versus the basic SP version and the area increases by 78 %. The best SW optimization is SP+Rot: the energy is divided by $\times 5.87$ with only a 19% area increase. The reason why these two configurations are by far the most efficient is that a computation can be launched every cycle (ii=1). Notice that *Loop-Unrolling* is not efficient with Catapult-C: the DP+LU consumes more energy than DP+Reg. That is the reason why we did not evaluate the $3 \times SP$+LU version.

If we now focus on the efficiency of *two* cascaded filters, there are three cases:

1. two independent filters (Algo. 4), with a temporary memory T of the same size than the input,
2. two pipelined filters with a 1-point FIFO for the Rot version and a 3-point FIFO for the Reg version (Algo. 5),
3. one single filter that is the fusion of the previous filters.

From a software point of view, the pipelined version corresponds to a *loop-fusion* but with two separates filters, while the fused version correspond to the filter fusion.

The table 1 summarizes all the results. Notice that in the case of the cascaded filters, the area does not consider the temporary memory area and its power consumption. As a matter of fact, a 65-nm 1024×8-bit SP memory has a power consumption of 14 $pJ/point$ with an area close to 15000 μm^2. So the cascaded filters results are just provided as the

first step of the optimization process, keeping in mind that that a designer will start with the pipelined version with a small FIFO. Compared to SP, the RW (respectively DP) memory has a surface and a power consumption that are $\times 1.5$ and $\times 1.3$ (respectively $\times 1.8$ and $\times 1.6$) bigger than SP ones.

Regarding the pipelined filter configurations, the energy of the Rot version is $\times 5.49$ smaller than the energy of Reg version. The figure is even higher for the fused filters configuration: 3.19 pJ, that is $\times 6.93$ less than the SP+Reg version, while the area is quite the same (0.45% smaller). Again, only $3\times$SP+Reg and SP+Rot versions lead to an *initiation interval* of 1 cycle per point.

Algorithm 4: 2 FIR3 filters, with temporary memory T

for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow X[i],\ y_1 \leftarrow F_1(x),\ T[i] \leftarrow y_1$
for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow T[i],\ y_2 \leftarrow F_2(x),\ Y[i] \leftarrow y_2$

Algorithm 5: two pipelined FIR3 filters

for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow X[i],\ y_1 \leftarrow F_1(x),\ y_2 \leftarrow F_2(y_1),\ Y[i] \leftarrow y_2$

Algorithm 6: two fused FIR3 filters

for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow X[i],\ y \leftarrow F_2(F_1(x)),\ Y[i] \leftarrow y$

3.3. Meta-programming

In order to provide a high level interface for these transforms, we implemented an IDL (*Interface Description Language*) like interface for the FIR function definitions using preprocessor meta-programming. Meta-programming is a set of software techniques that bring the benefits of automation to software developments. Those techniques rely on various languages specific features that enable developers to manipulate programs fragments as data at various stage of the compilation process. If the most famous meta-programming technique is template meta-programming (as defined in [9]), preprocessor or macro-based meta-programming fills a very important niche. Based on the seminal work [10], preprocessor meta-programming allows the developers to manipulate so-called preprocessor data structure and preprocessor control flows to generates arbitrarily complex codes for function interfaces or repetitive, token based iterations. Such structures include arrays, tuples and sequences. Control flows can be defined as preprocessor loops or iterations over structures. The advantages of such techniques over handwritten,

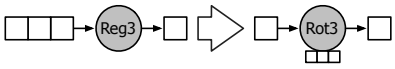

Fig. 2. *producer-consummer* model of FIR3 Reg & Rot versions

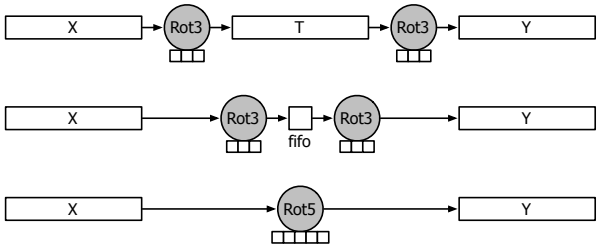

Fig. 3. three versions of two cascaded FIR3: independent filters with temporary T memory (top), pipelined filters with a 1-point FIFO (middle), fused filters (bottom)

direct macro expansion is the higher level of abstractions and the facility to compose macro generators. Libraries like `Boost.ConceptCheck` [11] or `Boost.Local` use such facilities to emulates languages features with a high degree of interface compliance.

In our case, preprocessor meta-programming was used to be able to comply with Catapult-C support of current C and C++ standards. As conformance to the C preprocessor standard is fairly common among existing tools, this strategy is portable across various tools outside of Catapult-C. Template meta-programming has been checked but was not completely supported by the Catapult-C compiler. In the following listings (Lst. 1 & 2), `ac_channel` is a Catapult-C C++ class for the stream I/O and `array` stands for a C99 variable length array.

Listing 1. FIR3 with C99 array

```
FIR(((array(sint8,N),X),(array(sint8,M),H),
     (array(sint8,N),Y)))
```

Listing 2. FIR3 with ac_channel I/O

```
FIR(((ac_channel(uint8),X),(array(sint8,M),H),
     (ac_channel(uint8),Y)))
```

The macros expand in the following stages: a declaration of variables and a static array `RF`, a loop containing the load of a point (from an C99 array of from a stream `ac_channel`), then a *register-rotation* `RF` with the memorization of the last input inside, the convolution and the output. We have observed that Catapult-C exactly generates a Register File with a rotation for the first loop.

Listing 3. macro expansion of listing 1

```
void fir(sint8 X[256], sint8 H[3], sint8 Y
    [256]) {
  int i;
  for(i=0; i<256; i++) {
    sint8 x; sint8 y;  sint8 r=1<<7;static
        sint8 RF[3];
    x = X[i];
    {int k; for(k=0;k<3-1;k++) RF[k] = RF[k+1];
        RF[3-1]= x;};
    {int k; sint16 y16; y16=r; for(k=0;k<3;k++)
        {y16+=RF[k+i]*H[k];} y=(uint8)(y16>>8)
       ;}; Y[i]=y;}
  }
```

Listing 4. macro expansion of listing 2

```
void fir(ac_channel<uint8>& X, sint8 H[3],
    ac_channel<uint8>& Y) {
  {uint8 x; uint8 y; sint8 r=1<<7; static sint8
      RF[3];
  x = X.load();
  {int k; for(k=0;k<3-1;k++) RF[k]=RF[k+1]; RF
      [3-1]= x;}
  {int k; sint16 y16; y16=r; for(k=0;k<3;k++) {
      y16+=RF[k]*H[k];} y=(uint8)(y16>>8);} Y.
      write(y);}
}
```

4. INFINITE IMPULSE RESPONSE FILTER

The recursive filters (Eq. 2) are difficult to optimize by a compiler. The reason is the dependency between the current output $y(n)$ and the previous outputs that are also the current inputs $y(n-1)$ and $y(n-2)$. Let us focus on the IIR12 filters that are used for edge detection. This filter (Eq. 2) is a smoother filter obtained after a simplification [12, 8] of the well-known Canny-Deriche filter [13, 14]

$$y(n) = (1-\gamma)^2 x(n) + 2\gamma y(n-1) - \gamma^2 y(n-2) \quad (2)$$

Algorithm 7: IIR12 filter – *Normal* form

for $i = 2$ **to** $n-1$ **do**
  $x_0 \leftarrow X[i-0],\ y_1 \leftarrow Y[i-1],\ y_2 \leftarrow Y[i-2]$
  $y_0 \leftarrow (b_0 \times x_0 + a_1 \times y_1 + a_2 \times y_2 + r)/256$
  $Y[i] \leftarrow y_0$

4.1. IIR optimization

Thanks to the coefficient expression, the *Factor* form (Eq. 3) is the *factorization* of the previous filter with powers of γ. It replaces one multiplication by two additions. Depending on their respective size / power consumption and latency, it could lead to smaller / lower power or faster designs.

$$y(n) = x(n) + 2\gamma\left[y(n-1) - x(n)\right] - \gamma^2\left[y(n-2) - x(n)\right] \quad (3)$$

Algorithm 8: IIR12 filter – *Factor* form

for $i = 2$ **to** $n-1$ **do**
  $x_0 \leftarrow X[i-0],\ y_1 \leftarrow Y[i-1],\ y_2 \leftarrow Y[i-2]$
  $y_0 \leftarrow \big(256 \times x_0 + a_1(y_1 - x_0) + a_2(y_2 - x_0) + r\big)/256$
  $Y[i] \leftarrow y_0$

Concerning the dependency problem, the shortest and strongest dependency $y(n-1)$ can be removed by replacing the expression of $y(n-1)$ into $y(n-2)$ (Eq. 4). This form is called *Delay* as the dependency is *delayed* to $y(n-2)$ and $y(n-3)$.

$$y(n) = (1-\gamma)^2 x(n) + 2\gamma(1-\gamma)^2 x(n-1) + 3\gamma^2 y(n-2) - 2\gamma^3 y(n-3) \quad (4)$$

As for FIR filters, some software optimizations like *Register-rotation* and *Loop-Unrolling* can be applied to the three forms. We have chosen to only apply the *Register-rotation* as it produces more efficient designs. The complexity of these three versions is given in table 2. We assume that the multiplication and division by 256 are replaced by left and right logical shifts.

Algorithm 9: IIR23 filter – *Delay* form

for $i = 2$ **to** $n-1$ **do**
  $x_0 \leftarrow X[i-0],\ x_1 \leftarrow X[i-1]$
  $y_2 \leftarrow Y[i-2],\ y_3 \leftarrow Y[i-3]$
  $y_0 \leftarrow (b_0 \times x_0 + b_1 \times x_1 + a_2 \times y_2 + a_3 \times y_3 + r)/256$
  $Y[i] \leftarrow y_0$

version	MUL	ADD	SHIFT	LOAD	STORE
Normal form	3	2+1	1	3	1
Factor form	2	4+1	2	3	1
Delay form	4	3+1	1	4	1

Table 2. IIR filters complexity

4.2. IIR Results and analysis

Table 3 presents the results of the three forms in term of area, power and energy. We can observe that energy consumption increases with ii, while area and power decrease. The reason is that Catapult-C optimizes for area and power, so the smallest area (*best A* configuration) and lowest power is obtained when ii is not constrained (noted auto ii). In that case, Catapult-C chooses $ii \in \{5, 6, 7\}$.

The smallest energy consumption is again obtained for $ii = 1$. If we compare *Normal* and *Factor* forms, $ii = 1$ and/or *Factor* provide energy reductions of $753/179 = \times 4.2$ at 200 MHz and $1171/211 = \times 5.6$ at 400 MHz. If we

	Normal form		*Factor* form		*Delay* form			
Freq (MHz)	200	400	200	400	200	400	**600**	**800**
Area (μm^2)								
auto ii	**3780**	**3762**	**3635**	**3931**	**4469**	**4830**	**5517**	9963
ii = 1	5274	5227	3984	4789	6769	7817	8239	**9872**
ii = 2	4746	4739	3555	4048	5425	6012	6285	8024
ii = 3	4163	4557	3492	3824	5204	5585	6496	10354
ii = 4	4019	3944	3475	3924	4925	5211	5648	10750
best E / best A	$\times$1.40	$\times$1.39	$\times$1.10	$\times$1.22	$\times$1.51	$\times$1.62	$\times$1.49	$\times$1.00
Power (μW)								
auto ii	**352**	**613**	325	**583**	**443**	**798**	**1248**	**2027**
ii = 1	445	692	**320**	710	551	1186	1668	2432
ii = 2	469	797	348	707	539	1085	1571	2516
ii = 3	433	698	329	612	488	939	1511	2274
ii = 4	405	686	317	634	518	927	1383	2374
best E / best A	$\times$1.26	$\times$1.13	$\times$1.00	$\times$1.22	$\times$1.24	$\times$1.49	$\times$1.34	$\times$1.20
Energy ($pJ/point$)								
auto ii	5.88	9.20	5.66	7.34	9.73	16.02	14.63	20.43
ii = 1	**2.12**	**1.65**	**1.40**	**1.77**	**2.02**	**2.97**	**2.73**	**3.02**
ii = 2	3.15	3.98	3.18	3.53	3.54	5.17	5.28	6.29
ii = 3	4.88	5.23	4.50	4.49	5.41	6.97	7.55	8.53
ii = 4	5.82	6.86	5.74	6.20	7.41	9.27	9.19	11.87
best A / best E	$\times$2.78	$\times$5.58	$\times$4.05	$\times$4.14	$\times$4.81	$\times$5.40	$\times$5.35	$\times$6.77

Table 3. IIR area, power and energy for synthesis frequency in [200..800] with 200 MHz step

compare *Normal* and *Delay*, the energy consumption of *Delay*+ii=1 is also smaller than for *Normal*+$ii = 0$. If we compare with *Factor* and *Normal* forms for $ii = 1$, *Delay* has a higher energy consumption, but it allows synthesis at twice the max frequency of the other forms.

One conclusion can be derived: if the peak power consumption is not too high, setting the *initiation interval* to the smallest possible value always leads to the best configuration for energy consumption, that is the key for longer embedded system autonomy. Moreover, the *Delay* transformation allow the use of higher frequencies, postponing the need to switch to a more recent and more expensive CMOS technology to get higher frequencies.

5. MORPHOLOGICAL FILTERING

The morphological operators [15] exist in two flavors: the binary ones and the grey level ones. In the following, we assume binary ones with a 3×3 structuring element. The filtering operators like opening, closing and alternate sequential filters are all based on two basic operators: the erosion and the dilation. These two morphological operators rely on the use of min and max computations over the structuring element for gray images. These operators are respectively replaced by the boolean operators AND and OR for binary images. As they are all idempotent, let us define the $\oplus$ operator that is one of the four basic operators. The basic implementation of such an operator is given in algorithm 10.

The usual problem to process the array edges is addressed by the use of Iliffe arrays [16] 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

Algorithm 10: 3×3 morphological filter (Eq.5) - Reg version

```
1 for i = 1 to n − 1 do
2     for j = 1 to n − 1 do
3         a0 ← X[i-1][j-1], b0 ← X[i-1][j], c0 ← X[i-1][j+1],
          a1 ← X[i][j-1], b1 ← X[i][j], c1 ← X[i][j+1],
          a2 ← X[i+1][j-1], b2 ← X[i+1][j], c2 ← X[i+1][j+1]
4         r ← a0 ⊕ b0 ⊕ c0 ⊕ a1 ⊕ b1 ⊕ c1 ⊕ a2 ⊕ b2 ⊕ c2
5         Y[i][j] ← r
```

r the *radius* of the kernel: for a $k \times k$ kernel, $k = 2r + 1$.

5.1. 2D Morphological operator optimization

As previously observed for FIR, *Loop-Unrolling* is not very efficient for HLS. So we prefer the *Register-Rotation* combined with *scalarization* (to put temporary results into registers). In algorithm 11, the pixel of the left and central column are loaded into registers before the loop (lines 4-6). After the computation,the registers are rotated (lines 14-16). Compared to the initial algorithm using 9 LOADs, that version has only 3 LOADs. The arithmetic complexity remains the same: 8 operations (named OP in the following). *Register-Rotation* leads to the Rot version (Algo. 11).

As the morphological operators are *idempotent*, the 2D structuring element $SE_{3\times3}$ can be decomposed into two 1D elements (Eq. 5).

$$SE_{3\times3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \quad (5)$$

Algorithm 11: 1-pass implementation of the 3×3 morphological filter with *Register Rotation*, Rot version

```
1  for i = 1 to n − 1 do
2      j ← 1   [preload the first two columns of each line]
3      a0 ← X(i − 1, j − 1), b0 ← X(i − 1, j)
4      a1 ← X(i, j − 1), b1 ← X(i, j)
5      a2 ← X(i + 1, j − 1), b2 ← X(i + 1, j)
6      for j = 1 to n − 1 step 3 do
7          c0 ← X(i − 1, j + 1)
8          c1 ← X(i, j + 1)
9          c2 ← X(i + 1, j + 1)
10         r ← a0 ⊕ b0 ⊕ c0 ⊕ a1 ⊕ b1 ⊕ c1 ⊕ a2 ⊕ b2 ⊕ c2
11         Y(i, j) ← r
12         a0 ← b0, b0 ← c0   [RR of the first line]
13         a1 ← b1, b1 ← c1   [RR of the second line]
14         a2 ← b2, b2 ← c2   [RR of the third line]
```

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. 11, lines 4-6), we compute the *reduced* value by column (Algo. 12, lines 5 & 6). Then the second operator is directly applied to the *reduced* values (Algo. 12, line 12). In that version, there are only 3 LOADs and 4 OPs. All the complexity figures are summarized in table 4 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 12: 1-pass implementation of the two seperated 1D operators with *reduction*, Red version

```
1  for i = 1 to n − 1 do
2      a0 ← X(i − 1, j − 1), b0 ← X(i − 1, j)
3      a1 ← X(i, j − 1), b1 ← X(i, j)
4      a2 ← X(i + 1, j − 1), b2 ← X(i + 1, j)
5      r_a ← a0 ⊕ a1 ⊕ a2   [reduction of the first column]
6      r_b ← b0 ⊕ b1 ⊕ b2   [reduction of the second column]
7      for j = 1 to n − 1 do
8          c0 ← X(i − 1, j + 1)
9          c1 ← X(i, j + 1)
10         c2 ← X(i + 1, j + 1)
11         r_c ← c0 ⊕ c1 ⊕ c2   [reduction of the third column]
12         r ← r_a ⊕ r_b ⊕ r_c   [applying the horizontal operator]
13         Y(i, j + 0) ← r
14         r_a ← r_b   [rotation of the reduced registers]
15         r_b ← r_c
```

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

Table 4. Morphological operator complexity and arithmetic intensity

5.2. Results and analysis

For the morphological operators, HLTs have a major impact on the efficiency. Let us call "bestE" and "bestA" the configurations associated to the smallest energy consumption, and the smallest area (Tab. 5 & 6). Let us also call auto ii the combinatory version. As the basic version (Reg) requires 9 LOADs (Tab. 5,) 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 bestE was reached for the smallest ii. The gain due to HLTs ranges from $\times\mathbf{2.85}$ to $\times\mathbf{3.05}$. Moreover, with a single-port RAM, the gap between Reg and Red versions would be even greater, as the energy increases with the ii. Finally, if we compare the configuration of the smallest area without HLT to the best Red, the gain ranges from $\times\mathbf{3.3}$ to $\times\mathbf{3.8}$.

freq (MHz)	200	400	600	800
bestA Reg (auto ii)	6.45	6.67	7.44	7.79
bestE Reg (ii=5)	5.49	5.76	6.44	5.87
bestE Rot (ii=2)	2.47	2.78	3.14	2.95
bestE Red (ii=2)	1.80	2.02	2.25	2.05
bestE Reg / bestE Red	×3.05	×2.85	×2.86	×2.86
bestS Reg / bestE Red	×3.58	×3.30	×3.31	×3.80

Table 5. Energy (pJ/pixel) of the morphological operator on a 65-nm ASIC with best ii for Reg, Rot and Red versions

freq (MHz)	200	400	600	800
bestA Reg (auto ii)	2893	2893	2893	2986
bestE Reg (ii=5)	3206	3208	3206	3030
ratio bestE / bestA	1.11	1.11	1.11	1.01
bestA Rot_1 (ii=4)	2905	2908	2923	2847
bestE Rot (ii=2)	3534	3534	3563	3443
ratio bestE / bestA	1.22	1.22	1.22	1.21
bestA Red (ii=4)	2374	2378	2400	2408
bestE Red (ii=2)	2685	2685	2714	2616
ratio bestE / bestA	1.13	1.13	1.13	1.09
bestA Reg / bestE Red	1.08	1.08	1.07	1.14

Table 6. Area (μm^2) of the morphological operator on 65 nm ASIC with best ii for Reg, Rot and Red versions: the smallest area and the area associated to the smallest energy

We can perform the same analysis for the area (Tab. 6). For each level of HLT optimization (Reg, Rot and Red), there is an area increase close to 11%, 22% and 13%. But if we compare the smallest area without HLT to the area associated of the smallest Red energy version, we can see that smallest Red energy version has a smaller area than the configuration without optimization (smallest Reg area). The impact of HLTs on chip area is limited.

6. CONCLUSION AND FUTURE WORKS

We have shown that we can enhance the performance of a HLS tool like Catapult-C by performing a software optimization like *register rotation* (by hand or by macro meta-programming). Moreover this software optimization is more efficient than hardware optimizations such as switching to dual-port or interleaved memories for filtering.

We have shown that high level transforms (HLT) are very efficient. The principle is to transform the code – and the associated hardware – in order to get the lowest possible latency represented by the value of the *initiation interval* ii. Smaller ii lead to reduced power and energy consumptions and faster execution times.

For FIR with fusion of cascaded filters, the energy consumption is divided by $\times 5.87$. For IIR, the *Delay* form allows to reach higher frequencies for synthesizing ASICs without needing the more recent technology. For 2D morphological filters, *reduction* is the key to reduce complexity and memory accesses, and thus decrease ii. We get a faster and smaller design with an energy consumption divided by $\times 3.5$

Usually one must choose between speed and low power consumption. With the combination of HLT and HLS, no choice is needed: the ASIC is both faster and greener!

In future works, we will implement the high level transforms through algorithmic skeletons [17, 18, 19] and C++ Meta-programming [9] to make the whole process (algorithm transformation, operator fusion and synthesis) fully automatic within th reach of the underlying tools as it will require a strict conformance to C++ standard. We will target color algorithms as they are more computation intensive [20] than usual black & white algorithms.

7. 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] M. Pueschel, J.M.F. Moura, J. Johnson, D. Padua, M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. 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] 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.

[5] 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.

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

[7] T. Duff, "`http://en.wikipedia.org/wiki/Duff's_device`," .

[8] L. Lacassagne, F. Lohier, and P. Garda, "Real time execution of optimal edge detectors on risc and dsp processors," in *ICASSP*. IEEE, 1998.

[9] D. Abrahams and A. Gurtovoy, Eds., *C++ Template Metaprogramming: concepts, tools and techniques from Boost and Beyond*, chapter 5, Addison-Wesley, 2005.

[10] P. Mensonides, "The chaos preprocessor library `https://github.com/ldionne/chaos-pp`," 2005.

[11] A. Lumsdaine J. Siek, "Concept checking: Binding parametric polymorphism in c++," in *Workshop on C++ Template Programming*, 2000, pp. 1–12.

[12] F. Garcia Lorca, L. Kessal, and D. demigny, "Efficient asic and fpga implementations of iir filters for real time edge detection," in *International Conference on Image Processing*. IEEE, 1997, pp. 406–409.

[13] J. F. Canny, "A computationa approach to edge detection," *Pattern Analysis Machine Intelligence*, vol. 8,6, pp. 679–698, 1986.

[14] R. Deriche, "Fast algorithms for low-level vision," *Transaction on Pattern Analysis*, vol. 12,1, pp. 78–87, 1990.

[15] P. Soille, *Morphological Image Analysis Principles and applications*, Springer, ISBN 3-540-42988-3, 1999.

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

[17] 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.

[18] J. Serot and . Falcou, "Functional meta-programming for parallel skeletons," in *Computational Science–ICCS 2008*, pp. 154–163. Springer Berlin Heidelberg, 2008.

[19] K. Hammond and G. Michaelson, Eds., *Algorithmic skeletons: structured management of parallel computation*, chapter 13, pp. 289–303, Springer, 1999.

[20] M. Gouiffès and B. Zavidovique, "Body color sets: A compact and reliable representation of images," *Journal of Visual Communication and Image Representation*, vol. 22,1, pp. 48–60, 2011.