REAL-TIME VIDEO PIXEL MATCHING

Jean-Baptiste Note, Mark Shand, Jean E. Vuillemin

Département d'informatique
École Normale Supérieure
45, rue d'Ulm, 75005 Paris
email: jean-baptiste.note@ens.fr, mark.shand@ens.fr, jean.vuillemin@inria.fr

ABSTRACT

We present an efficient implementation of a state of the art algorithm *PixelMatch* for matching all pixels within consecutive video frames. The method is of practical interest for tracking movements in video; it is also related to block-matching in standard video compression methods.

From the source specification of *PixelMatch*, a limited number of high-level code transformations are first performed, and analyzed, to produce an intermediate executable software code.

From the intermediate software code, an efficient reconfigurable hardware circuit is synthesized, in a fully automatic manner, to process Standard Definition video streams in real-time on a current mid-size FPGA. The software implementation compiled from the same code runs orders of magnitude faster than the original specification. Despite this, real-time software processing of video streams by *PixelMatch* is still only within reach of the highest-end workstations.

1. INTRODUCTION

PixelMatch estimates the motion of every pixel within two consecutive $W \times H$ images I_1 and I_2 in a video stream. For each position in I_1, a corresponding neighborhood is explored in I_2, to find the pixel in image I_2 with the highest correlation to the original pixel in image I_1. The movement between the original pixel and its best match in I_2 is duly recorded in the output image. The image correlation used is the Sum of Absolute Differences (SAD) between each pixel value in I_1 and in I_2, summed over a square neighborhood around each reference pixel.

So the original specification for *PixelMatch* embeds three finite search loops. An optimal ordering of the search scan through both images is derived by analyzing the internal operations, memory and bandwidth required for each ordering. The final real time hardware circuit is obtained from a minimal logic solution, by appropriately unfolding it in space.

The paper is organized as follows. Section 2 presents the matching problem in mathematical terms and optimally orders the search loops. Based on these premises, section 3 explores the various logic/memory tradeoffs available for simplifying the SAD computation loop. Section 4 describes an optimized tradeoff for the Virtex II hardware and how it can be suitably folded in time/space. Section 5 describes an optimized software implementation.

2. PROBLEM DESCRIPTION

2.1. Basic formulae

Consider two $W \times H$ images I_1 and I_2, usually consecutive images in a video stream. The aim is to estimate the motion of every pixel from I_1 to I_2: given a pixel in I_1 we want to find its *matching pixel* in I_2.

In order to quantify the similarity $\mathcal{E}_{\vec{d}}$ of position p_0 in I_1 and position $p_0 + \vec{d}$ in I_2 we use a classical sum of absolute differences (SAD) of regions around the two positions; this is the first of our nested loops, the integration loop:

$$\mathcal{E}_{\vec{d}}\,(p_0\,) = \sum_{|\vec{\delta}\,| \leqslant \beta} |I_1\,(p_0\,+\vec{\delta}\,) - I_2\,(p_0\,+\vec{d}+\vec{\delta}\,)| \tag{1}$$

The sum is taken over a square region the radius β of which is a parameter for *PixelMatch*.

Given a position p_0 in I_1 we look for the *minimum correlation* for a limited range of displacements. This minimum points to the matching position in I_2 and the associated motion vector; this is the second search loop:

$$\mathcal{E}_{\vec{d}_{\mathrm{opt}}}\,(p_0\,) = \min_{|\vec{d}\,| \leqslant \alpha} \mathcal{E}_{\vec{d}}\,(p_0\,) \tag{2a}$$

$$\vec{d}_{\mathrm{opt}}\,(p_0\,) = \arg\min_{|\vec{d}\,| \leqslant \alpha} \mathcal{E}_{\vec{d}}\,(p_0\,) \tag{2b}$$

Minimization is done over a square region the radius α of which is the second parameter for *PixelMatch*. It is usually chosen so that $\alpha \leqslant \beta$ in order to prevent erroneous matches.

Contrary to usual block-matching for video compression [1], the motion vector and associated *minimum correlation* is computed for every position in I_1. Running over all positions is the last of our search loops; for SD images this loop must iterate about $12,000,000$ times per second. Thus a naive implementation of the algorithm is:

input : I_1 and I_2 of size $W \times H$
output: map of motion vectors and correlations

`InitOptimum;`

forall $|\vec{d}| \leqslant \alpha$ **do**
  foreach $p \in I_1$ **do**
    $\mathcal{E}_{\vec{d}} \leftarrow 0;$
    forall $|\delta| \leqslant \beta$ **do**
      $\mathcal{E}_{\vec{d}}$ += $E_{\vec{d}}(p+\delta);$
    `UpdateOptimum(`$p, \vec{d}, \mathcal{E}_{\vec{d}}$`);`

Algorithm 1: Raw block matching

input: $p, \vec{d}, \mathcal{E}_{\vec{d}}(p)$
data : $\vec{d}_{\text{opt}}$ map of current best motion vectors
data : $\mathcal{E}_{\vec{d}_{\text{opt}}}$ map of current best correlations

if $\mathcal{E}_{\vec{d}}(p) \leqslant \mathcal{E}_{\vec{d}_{\text{opt}}}(p)$ **then**
  $\vec{d}_{\text{opt}}(p) \leftarrow \vec{d};$
  $\mathcal{E}_{\vec{d}_{\text{opt}}}(p) \leftarrow \mathcal{E}_{\vec{d}}(p);$

Algorithm 2: `UpdateOptimum` routine

2.2. Ancillary definitions

The translation $T_{\vec{d}}$ of the image I_2 by the motion vector $\vec{d}$ is:

$$I_2 \circ T_{\vec{d}}(p) = I_2(p + \vec{d}) \tag{3}$$

The *energy function* $E_{\vec{d}}$ at position p, for displacement $\vec{d}$ is:

$$E_{\vec{d}}(p) = |I_1(p) - I_2 \circ T_{\vec{d}}(p)| \tag{4}$$

We wish to decompose the computation into vertical strips. The *vertical correlation* $\mathcal{E}^v_{\vec{d}}$ at position p, for displacement $\vec{d}$ is:

$$\mathcal{E}^v_{\vec{d}}(p) = \sum_{\delta \in \{0\} \times [-\beta, \beta]} E_{\vec{d}}(p + \delta) \tag{5}$$

2.3. Efficient SAD computation

There are many common subexpressions between instances of formula (1) when looping over positions p for a fixed displacement $\vec{d}$.

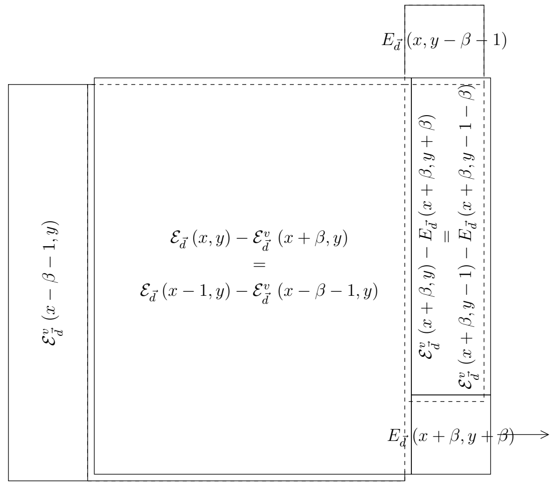

Fig. 1. Subexpression sharing in correlation computation

Consider figure 1; the SAD on the region surrounding position (x, y) can be computed from the sum around position $(x-1, y)$ by subtracting the sum over the vertical strip leaving the integration square and adding the sum over the vertical strip entering it. This is written as formula 6.

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x,y) = \mathcal{E}_{\vec{d}}(x-1,y) - \mathcal{E}^v_{\vec{d}}(x-\beta-1,y) \\ + \mathcal{E}^v_{\vec{d}}(x+\beta,y)\end{aligned} \tag{6}$$

The vertical strip correlation $\mathcal{E}^v_{\vec{d}}(x+\beta, y)$ itself can be computed with minimal overhead from the vertical correlation value of the position above, by adding the energy incoming into the strip, and subtracting the energy at the position leaving the strip:

$$\begin{aligned}\mathcal{E}^v_{\vec{d}}(x+\beta,y) = \mathcal{E}^v_{\vec{d}}(x+\beta,y-1) \\ - E_{\vec{d}}(x+\beta,y-\beta-1) + E_{\vec{d}}(x+\beta,y+\beta)\end{aligned} \tag{7}$$

We use these equations to avoid redundant computation across the image by storing some intermediate results $\mathcal{E}_{\vec{d}}$, $\mathcal{E}^v_{\vec{d}}$, $E_{\vec{d}}$: we trade computation for memory. Thus the bulk of the computation of formula (1) is eliminated.

The innermost loop of algorithm 1 is replaced with a call to an *incremental correlation computation function*. The `IncrCorrelation` function has internal static memory for storing the intermediate results needed to compute the correlation function incrementally from one pixel to the next, in raster-scan order.

The algorithm may now be re-expressed as:

input : I_1 and I_2 of size $W \times H$
output: map of motion vectors and correlations

`InitOptimum`;

1 **forall** $|\vec{d}| \leqslant \alpha$ **do**
  $\mathcal{E}_{\vec{d}} \leftarrow 0$;
2 **foreach** $p \in I_1$ **do**
    $\mathcal{E}_{\vec{d}} \leftarrow$ `IncrCorrelation` $(\mathcal{E}_{\vec{d}}, I_1, I_2)$;
    `UpdateOptimum` $(p, \vec{d}, \mathcal{E}_{\vec{d}})$;

Algorithm 3: Computationally-efficient block matching

2.4. Efficient minimum correlation computation

Algorithm 3 makes one pass over the images for each value of $\vec{d}$, updating in the `UpdateOptimum` routine the $\vec{d}_{\text{opt}}$ and $\mathcal{E}_{\vec{d}_{\text{opt}}}$ maps.

Storage space is needed for maintaining these maps. Bandwidth is consumed as we must read and possibly update them for each newly-computed correlation value.

Data locality is better exploited if we compute the $\mathcal{E}_{\vec{d}}$ values at the same position across all $\vec{d}$ in a tight loop. In a single visit to the position, all the data needed to make the optimum decision is computed. This is just a matter of inverting loop 1 and 2 in algorithm 3 so that the computation for formula (2a) is now innermost:

input : I_1 and I_2 of size $W \times H$
output: map of motion vectors and correlations

1 **foreach** $p \in I_1$ **do**
  $\mathcal{E}_{\vec{d}_{\text{opt}}} \leftarrow maxint$;
2 **forall** $|\vec{d}| \leqslant \alpha$ **do**
    $\mathcal{E}_{\vec{d}} \leftarrow$ `IncrCorrelation` $_{\vec{d}}\,(\mathcal{E}_{\vec{d}}, I_1, I_2)$;
    if $\mathcal{E}_{\vec{d}} \leqslant \mathcal{E}_{\vec{d}_{\text{opt}}}$ **then**
      $\vec{d}_{\text{opt}} \leftarrow \vec{d}$;
      $\mathcal{E}_{\vec{d}_{\text{opt}}} \leftarrow \mathcal{E}_{\vec{d}}$;
  `Write` $(p, \mathcal{E}_{\vec{d}_{\text{opt}}}, \vec{d}_{\text{opt}})$;

Algorithm 4: Bandwidth-efficient block matching

Now $\mathcal{E}_{\vec{d}_{\text{opt}}}$ and $\vec{d}_{\text{opt}}$ are merely stack variables. However `IncrCorrelation`'s internal memory must be duplicated $(2\alpha + 1)^2$ times to accommodate the parallel computations of correlations.

3. INCREMENTAL CORRELATION COMPUTATION

Function `IncrCorrelation`, the incremental correlation computation function, is at the heart of algorithm 4. In this section we explore the various memory/logic or memory/instruction count tradeoffs available for this function.

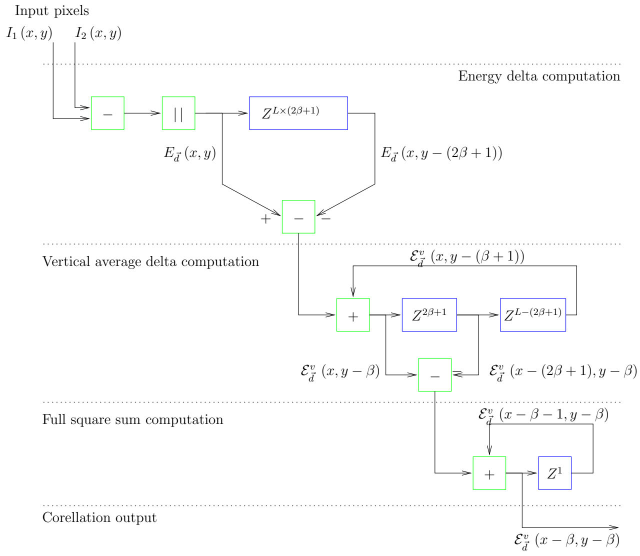

Fig. 2. Correlation computation with minimal logic

3.1. Minimal logic

We use (6) and (7) above that yield:

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x-\beta, y-\beta) &= \mathcal{E}_{\vec{d}}(x-\beta-1, y-\beta) \\ &- \mathcal{E}^v_{\vec{d}}(x-2\beta-1, y-\beta) + \mathcal{E}^v_{\vec{d}}(x, y-\beta-1) \\ &- E_{\vec{d}}(x, y-2\beta-1) \\ &+ |I_1(x,y) - I_2 \circ T_{\vec{d}}(x,y)|\end{aligned} \tag{8}$$

$\mathcal{E}_{\vec{d}}$ can be computed incrementally with as little logic as one adder, two subtracters and one absolute value. The cost for this computation reduction is that the values for $\mathcal{E}_{\vec{d}}$, $\mathcal{E}^v_{\vec{d}}$ and $E_{\vec{d}}$ have to be memoized [1] in sliding windows.

The circuit of figure 2 embodies this computation. It takes pixels in raster-scan order and outputs the $\mathcal{E}_{\vec{d}}$ values in-order. The memory requirements are shown in table 1. The $E_{\vec{d}}$ shift-register is huge: this is one of the main drawbacks of this circuit.

3.2. Duplication of the energy computation

The amount of static memory in the `IncrCorrelation` block can be reduced by not memoizing the $E_{\vec{d}}$ term, recomputing it instead with data fetched from I_1 and I_2, according to the following formula:

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x-\beta, y-\beta) &= \mathcal{E}_{\vec{d}}(x-\beta-1, y-\beta) \\ &- \mathcal{E}^v_{\vec{d}}(x-2\beta-1, y-\beta) + \mathcal{E}^v_{\vec{d}}(x, y-\beta-1) \\ &- |I_1(x, y-2\beta-1) - I_2 \circ T_{\vec{d}}(x, y-2\beta-1)| \\ &+ |I_1(x,y) - I_2 \circ T_{\vec{d}}(x,y)|\end{aligned} \tag{9}$$

[1] http://wikipedia.org/wiki/Memoization

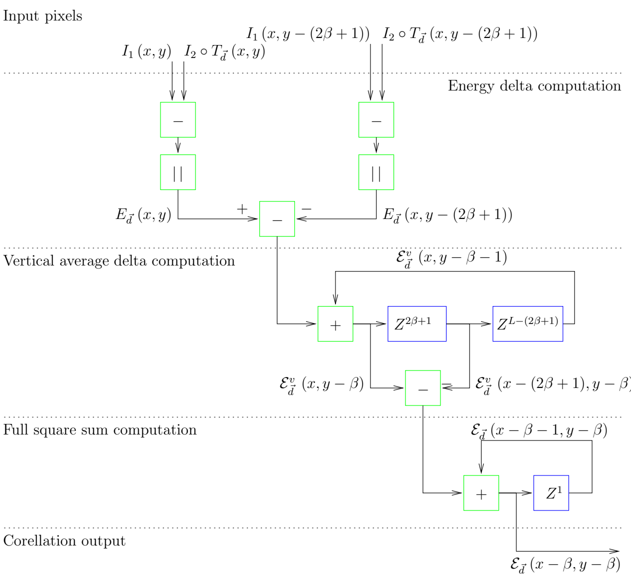

Fig. 3. Correlation computation of subsection 3.2

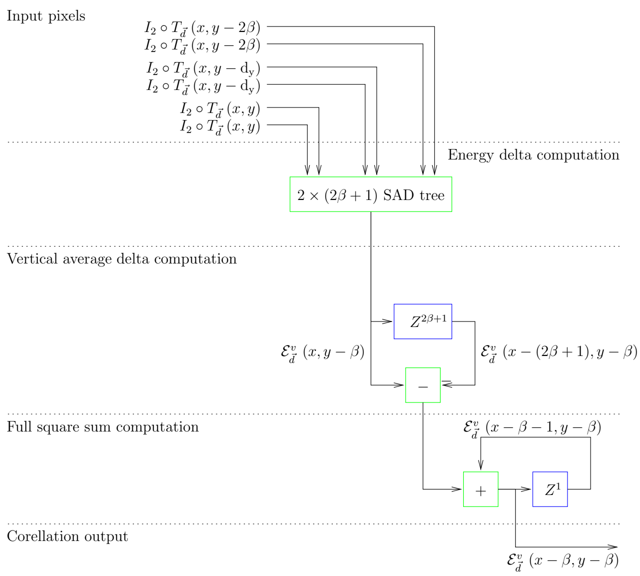

Fig. 4. Correlation computation with minimum memory

This design trades the memory in the $E_{\vec{d}}$ buffer for some redundant computation and an increased bandwidth to I_1 and I_2.

The circuit of figure 3 embodies this computation. The memory requirements for this module are greatly decreased, as shown in table 1 – we dispose of the $E_{\vec{d}}$ shift-register.

3.3. Duplication of the vertical sum computation

We can further lower the memory requirements of the module by not to memoizing the farthest $\mathcal{E}^v_{\vec{d}}$ term, $\mathcal{E}^v_{\vec{d}}(x, y - \beta - 1)$ of formula (9). Therefore we have to recompute it with data fetched from I_1 and I_2, according to the following formula:

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x-\beta, y-\beta) &= \mathcal{E}_{\vec{d}}(x-\beta-1, y-\beta) \\ &- \mathcal{E}^v_{\vec{d}}(x-2\beta-1, y-\beta) \\ &+ \sum_{0 \leqslant \mathrm{d_y} \leqslant 2\beta} \left| I_1(x, y-\mathrm{d_y}) - I_2 \circ T_{\vec{d}}(x, y-\mathrm{d_y}) \right|\end{aligned} \tag{10}$$

This design eliminates a large part of the $\mathcal{E}^v_{\vec{d}}$ shift-register – the internal memory of `IncrCorrelation` has been made independent of the image width W.

This memory was traded for a considerable increase in logic, bandwidth and wiring complexity. Indeed, the circuit now contains a SAD tree over $(2\alpha + 1)$ elements and has to fetch $(2\beta + 1)$ values from each image (figure 4).

4. HARDWARE IMPLEMENTATION

4.1. Implementation parameters

The goal for the circuit is to be able to process in real-time Standard Definition video streams. Standard Definition video streams can be PAL or NTSC. The bandwidth for both standards is practically the same, and we chose to settle for PAL numbers.

We treat standard PAL images in 720×576 format, with a framerate of 25 images/sec. Algorithmic parameters β and α are both set to 5. So the search and integration square area are 121 pixels, the throughput is $10,368,000$ motion vectors per second selected among $1,254,528,000$ correlations.

Our prototyping platform is the Sepia phase 3 board from HP's Sepia project [2]. This PCI board contains a Virtex II XC2V2000 for the realization of application specific circuits.

4.2. Logic/memory tradeoff

In section 3 we considered one instance of the subroutine $\texttt{IncrCorrelation}_{\vec{d}}$ which takes as input data from the images and computes the correlation value at each pixel for a given displacement $\vec{d}$.

Section 2.4 showed that we need to instantiate $(2\alpha+1)^2$ times this module in order to compute the motion vectors with as little logic, memory and bandwidth as possible.

Memory usage puts a hard constraint on our design: we have to duplicate $(2\alpha + 1)^2$ times the internal memory of `IncrCorrelation`. The circuit of section 3.3 is the only circuit which allows this on our Virtex II platform.

	Minimum logic	Intermediate logic	Minimum memory
$\mathcal{E}_{\vec{d}}$ SR	$(n + 2\log_2(2\beta+1)) \times 1$		
$\mathcal{E}^{v}_{\vec{d}}$ SR	$(n + \log_2(2\beta+1)) \times W$		$(n + \log_2(2\beta+1)) \times (2\beta+1)$
$E_{\vec{d}}$ SR	$n \times W.(2\beta+1)$	0	
Memory	76,047 b	8,463 b	1,346 b
Logic	66 LUTs	83 LUTs	148 LUTs

Table 1. Memory and logic requirements of various versions of `IncrCorrelation`

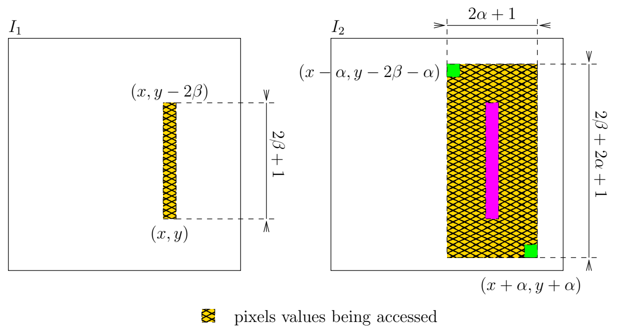

Fig. 5. Access to I_1 and I_2 data for computing the $(2\alpha+1)^2$ correlations at position $(x-\beta, y-\beta)$

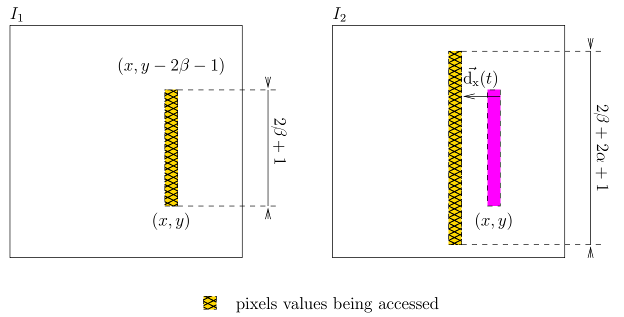

Fig. 6. Access to I_1 and I_2 for the $(C_{d_y})_{d_y \in [-\alpha, \alpha]}$ units

We also put on the FPGA the I_1 and I_2 shift-registers used for buffering the data needed for feeding the module instances. The increase in bandwidth from I_1 and I_2 is limited as many data accesses from the $(2\alpha+1)^2$ modules overlap, yielding the memory access pattern of figure 5.

4.3. Logic folding

The circuit which fully unrolls the module of section 3.3 $(2\alpha+1)^2$ times accepts one pixel from each image and outputs one motion vector per clock cycle.

The design is mainly feed-forward. Automatic retiming [3] yields very high clock speeds, around 200MHz on the Virtex II, far in excess of the throughput required for SD video processing. Can we meet our needs with less logic?

This particular problem involves duplicated logic and is therefore well adapted to *time-space folding*. Time-space folding is described theoretically in [4] and has a long history in engineering practice of circuit design (for example [5]). This technique allows us to trade logic for a longer computing time while memory stays the same.

The logic within the $(2\alpha+1)^2$ module instances can be folded k times so that each logic instance computes in turn the correlation for $(2\alpha+1)^2/k$ values of the displacement. In our case the sweet spot for logic sharing seems to be the balance between time and space unfolding.

Thus the loop over $(2\alpha+1)^2$ correlations at each position is evenly unrolled $2\alpha+1$ times in space and $2\alpha+1$ times in time.

In this design, one computation unit labeled $C_{d_{y_0}}$ computes in turn the correlation values $(\mathcal{E}_{(d_x, d_{y_0})})_{d_x \in [-\beta, \beta]}$, then the whole design moves on to the next position.

Logic sharing comes at the cost of a more complex control as we need to feed the logic with the appropriate values from the shift-registers holding the data for I_1 and I_2.

Every $2\alpha+1$ clock tick the design inputs a new pixel into the I_1 and I_2 shift-registers. Then during the $2\alpha+1$ next clock ticks, the data access window of figure 6 slides to the right according to the value of d_x. This is achieved with moving taps in the I_2 shift-register. The dual port RAMs of the Virtex II are a perfectly suited for this task.

5. SOFTWARE IMPLEMENTATION

5.1. Implementation

The high-level description of algorithm 4 can be directly translated into C once an appropriate software implementation of `IncrCorrelation` has been chosen.

Modern microprocessors are essentially sequential machines once one has exploited the instruction level parallelism offered by superscalar execution units. In a *sequential machine* the whole algorithm is unrolled in time, as opposed to a *parallel machine* where unrolling can be performed either in time or space as was considered earlier. Naively one

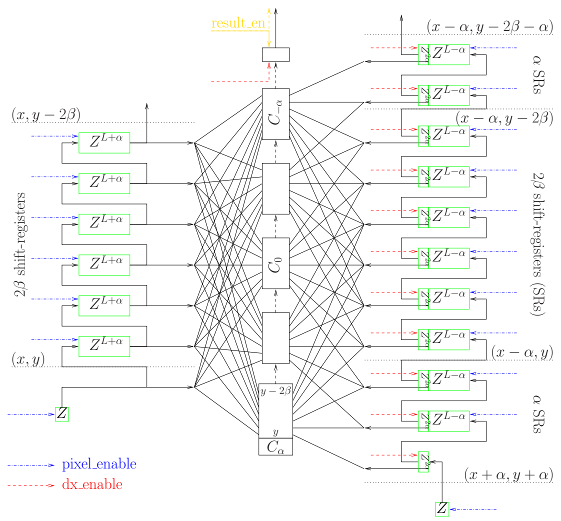

Fig. 7. Instance of the folded circuit for $\alpha = 2, \beta = 3$

may think that the sole parameter to optimize is the number of instructions executed. However, the cache hierarchy has a profound effect on instruction execution times, so much so that it may be more efficient to repeat certain calculations if it leads to improved cache performance. Thus a tradeoff must be made between, on the one hand, the *instruction count* and, on the other, the *memory usage* and *data locality* of the algorithm.

The algorithmic discussion of section 3 showed the various instruction count / memory tradeoffs available for the `IncrCorrelation` function. Our tests clearly point to the architecture of subsection 3.2 as the best candidate for the software implementation. The memoized data of the architecture 3.1 does not fit in cache; conversely the instruction count of architecture 3.3 is too big.

5.2. Performance

Based on the above considerations we focussed on a software implementation of the algorithm of subsection 3.2.

The code was straightforwardly derived from the high-level description and compiled with GCC 4.0. The resulting code achieves speeds of 1.8Mpixels/second on a 2.2GHz Opteron core model number 275. The cachegrind [6] profiles showed less than 1 percent cachemiss in the inner loop. Our reference code is available on the web [7].

Modern processors with multiple cores and SMP abilities allow for coarse-grained unrolling in space – thread-level parallelism. We threaded the code on a quad-core workstation to reach half-realtime processing of PAL streams.

These same processing cores are actually parallel machines with multiple-issue pipelines; SIMD instructions are ubiquitous. These features allow for low-level, processor-dependent optimizations. Such an optimization path is promising and will be investigated in future work.

6. CONCLUSION

We have performed an in-depth analysis of the *PixelMatch* pixel-level motion estimation algorithm. This work yields an intermediate optimized algorithm description based on logic/memory tradeoffs and loop reordering. From this intermediate description we show how to straightforwardly derive efficient hardware and software implementations.

The resulting hardware is tailored for real-time processing of PAL data streams by logic folding. It makes very efficient use of the logic and memory resources found on the Virtex II FPGA. It uses minimum bandwidth, as the images are streamed once into the design while the resulting maps are directly streamed out of it.

The resulting software reduces the instruction count, wisely uses the cache, and optimally orders the software instructions, yielding an efficient use of the multiple pipelines found in modern processors to achieve maximum performance, half-realtime on a four-way Opteron workstation.

The methodology of high-level optimizations applied to both hardware and software can be fruitfully applied to other streaming computations. Often video streaming computations use data local to the current pixel position and can benefit from the same high-level reasoning to achieve efficient implementation.

7. REFERENCES

[1] J.-C. Tuan, T.-S. Chang, and C.-W. Jen, "On the data reuse and memory bandwidth analysis for full-search block-matching vlsi architecture." *IEEE Trans. Circuits Syst. Video Techn.*, vol. 12, no. 1, pp. 61–72, 2002.

[2] L. Moll, M. Shand, and A. Heirich, "Sepia: Scalable 3d compositing using pci pamette." in *FCCM '99*. IEEE Computer Society, 1999, pp. 146–.

[3] F. T. Leighton, *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[4] J. Hopcroft, W. Paul, and L. Valiant, "On time versus space," *J. ACM*, vol. 24, no. 2, pp. 332–337, 1977.

[5] C. D. Thompson, "Area-time complexity for vlsi," in *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1979, pp. 81–88.

[6] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, 2003.

[7] Pixelmatch reference implementation. [Online]. Available: http://jbnote.free.fr/pixelmatcher