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 $E_{\bar{\delta}}(p_0)$ of position $p_0$ in $I_1$ and position $p_0 + \bar{\delta}$ 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:

$$E_{\bar{\delta}}(p_0) = \sum_{|\bar{\delta}| \leq \beta} |I_1(p_0 + \bar{\delta}) - I_2(p_0 + \bar{\delta} + \bar{\delta})|$$ (1)

The sum is taken over a square region the radius $\beta$ 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:

$$E_{\delta_{\text{opt}}}(p_0) = \min_{|\bar{\delta}| \leq \alpha} E_{\bar{\delta}}(p_0) \quad (2a)$$

$$\bar{\delta}_{\text{opt}}(p_0) = \arg \min_{|\bar{\delta}| \leq \alpha} E_{\bar{\delta}}(p_0) \quad (2b)$$

Minimization is done over a square region the radius $\alpha$ of which is the second parameter for PixelMatch. It is usually chosen so that $\alpha \leq \beta$ 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:

\[
\text{input} : I_1 \text{ and } I_2 \text{ of size } W \times H \\
\text{output} : \text{map of motion vectors and correlations}
\]

\[
\text{InitOptimum}:
\]

1. forall \( |\tilde{d}| \leq \alpha \) do.
2. \quad foreach \( p \in I_1 \) do.
3. \quad \quad forall \( |\delta| \leq \beta \) do.
4. \quad \quad \quad \quad \text{\( E_{\tilde{d}}(p + \delta) \)}.
5. \quad \quad UpdateOptimum(\( p, \tilde{d}, E_{\tilde{d}} \));

Algorithm 1: Raw block matching

\[
\text{input}: p, \tilde{d}, E_{\tilde{d}}(p) \\
\text{data}: \tilde{d}_{\text{opt}} \text{ map of current best motion vectors} \\
\text{data}: E_{\text{opt}} \text{ map of current best correlations}
\]

if \( E_{\tilde{d}}(p) \leq E_{\text{opt}}(p) \) then

\[
\tilde{d}_{\text{opt}}(p) \leftarrow \tilde{d}; \\
E_{\text{opt}}(p) \leftarrow E_{\tilde{d}}(p);
\]

Algorithm 2: UpdateOptimum routine

2.2. Ancillary definitions

The translation \( T_{\tilde{d}} \) of the image \( I_2 \) by the motion vector \( \tilde{d} \) is:

\[
I_2 \circ T_{\tilde{d}}(p) = I_2(p + \tilde{d}) \tag{3}
\]

The energy function \( E_{\tilde{d}} \) at position \( p \), for displacement \( \tilde{d} \) is:

\[
E_{\tilde{d}}(p) = |I_1(p) - I_2 \circ T_{\tilde{d}}(p)| \tag{4}
\]

We wish to decompose the computation into vertical strips. The vertical correlation \( E_{\tilde{d}}^v \) at position \( p \), for displacement \( \tilde{d} \) is:

\[
E_{\tilde{d}}^v(p) = \sum_{\delta \in \{0\} \times [-\beta, \beta]} E_{\tilde{d}}(p + \delta) \tag{5}
\]

2.3. Efficient SAD computation

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

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.

\[
E_{\tilde{d}}(x, y) = E_{\tilde{d}}(x - 1, y) - E_{\tild{d}}^v(x - \beta - 1, y) + E_{\tild{d}}(x + \beta, y) \tag{6}
\]

The vertical strip correlation \( E_{\tild{d}}^v(x + \beta, y) \) 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:

\[
E_{\tild{d}}^v(x + \beta, y) = E_{\tild{d}}^v(x + \beta, y - 1) - E_{\tild{d}}(x + \beta, y - \beta - 1) + E_{\tild{d}}(x + \beta, y + \beta) \tag{7}
\]

We use these equations to avoid redundant computation across the image by storing some intermediate results \( E_{\tild{d}} \), \( E_{\tild{d}}^v \), \( E_{\tild{d}}^v \): 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:

![Fig. 1. Subexpression sharing in correlation computation](image)
input: $I_1$ and $I_2$ of size $W \times H$
output: map of motion vectors and correlations

InitOptimum;

1 forall $|\vec{d}| \leq \alpha$ do
2 \hspace{1em} $E_{\vec{d}} \leftarrow 0$;
3 \hspace{1em} foreach $p \in I_1$ do
4 \hspace{2em} $E_{\vec{d}} \leftarrow \text{IncrCorrelation}(E_{\vec{d}}, I_1, I_2);$ \hspace{1em}
5 \hspace{2em} UpdateOptimum $(p, d, E_{\vec{d}});$ \hspace{1em}
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 $E_{\vec{d}_{\text{opt}}}$ 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 $E_{\vec{d}}$ 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
2 \hspace{1em} $E_{\vec{d}_{\text{opt}}} \leftarrow \text{maxint};$
3 \hspace{1em} forall $|\vec{d}| \leq \alpha$ do
4 \hspace{2em} $E_{\vec{d}} \leftarrow \text{IncrCorrelation}(E_{\vec{d}}, I_1, I_2);$ \hspace{1em}
5 \hspace{2em} if $E_{\vec{d}} \leq E_{\vec{d}_{\text{opt}}}$ then
6 \hspace{3em} $\vec{d}_{\text{opt}} \leftarrow \vec{d};$
7 \hspace{3em} $E_{\vec{d}_{\text{opt}}} \leftarrow E_{\vec{d}};$ \hspace{1em}
8 \hspace{2em} Write $(p, E_{\vec{d}_{\text{opt}}}, \vec{d}_{\text{opt}});$ \hspace{1em}
Algorithm 4: Bandwidth-efficient block matching

Now $E_{\vec{d}_{\text{opt}}}$ 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.

3.1. Minimal logic

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

$$E_{\vec{d}}(x - \beta, y - \beta) = E_{\vec{d}}(x - \beta - 1, y - \beta) - E_{\vec{d}}(x - 2\beta - 1, y - \beta) + E_{\vec{d}}(x, y - \beta - 1) - E_{\vec{d}}(x, y - 2\beta - 1) + |I_1(x, y) - I_2 \circ T_{\vec{d}}(x, y)|$$ (8)

$E_{\vec{d}}$ can be computed incrementally with as little logic as one adder, two subtractors and one absolute value. The cost for this computation reduction is that the values for $E_{\vec{d}}, E_v$ and $E_{\vec{d}}$ have to be memoized in sliding windows.

The circuit of figure 2 embodies this computation. It takes pixels in raster-scan order and outputs the $E_{\vec{d}}$ 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:

$$E_{\vec{d}}(x - \beta, y - \beta) = E_{\vec{d}}(x - \beta - 1, y - \beta) - E_{\vec{d}}(x - 2\beta - 1, y - \beta) + E_{\vec{d}}(x, y - \beta - 1) - |I_1(x, y - 2\beta - 1) - I_2 \circ T_{\vec{d}}(x, y - 2\beta - 1)| + |I_1(x, y) - I_2 \circ T_{\vec{d}}(x, y)|$$ (9)

1http://wikipedia.org/wiki/Memoization
3.3. Duplication of the vertical sum computation

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

$$
E^v_d(x - \beta, y - \beta) = E^v_d(x - \beta - 1, y - \beta) - E^v_d(x - 2\beta - 1, y - \beta) + \sum_{0 \leq d_y \leq 2\beta} |I_1(x, y - d_y) - I_2 \circ T^x_d(x, y - d_y)|
$$

(10)

This design eliminates a large part of the $E^v_d$ 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).
Intermediate logic

<table>
<thead>
<tr>
<th></th>
<th>Minimum logic</th>
<th>Intermediate logic</th>
<th>Minimum memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_J^c$ SR</td>
<td></td>
<td>$(n + 2 \log_2(2\beta + 1)) \times 1$</td>
<td></td>
</tr>
<tr>
<td>$E_J^v$ SR</td>
<td></td>
<td>$(n + \log_2(2\beta + 1)) \times W$</td>
<td>$(n + \log_2(2\beta + 1)) \times (2\beta + 1)$</td>
</tr>
<tr>
<td>$E_J^d$ SR</td>
<td></td>
<td>$n \times W(2\beta + 1)$</td>
<td>0</td>
</tr>
<tr>
<td>Memory</td>
<td>76,047 b</td>
<td>8,463 b</td>
<td>1,346 b</td>
</tr>
<tr>
<td>Logic</td>
<td>66 LUTs</td>
<td>83 LUTs</td>
<td>148 LUTs</td>
</tr>
</tbody>
</table>

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

![Image](image1.png)

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

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 $C_{d_y}$ computes in turn the correlation values $(E_{(d_x, d_y)})_{d_y \in [-\alpha, \alpha]}$, 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
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 trade-off 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 cachebranch [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