Published at ICSPAT '97, International Conference on Signal Processing Applications and Technology, September 14-17, 1997, San Diego, CA, USA. pp 1375 - 1380

High speed video de-interlacing with a programmable TriMedia VLIW core

A.K. Riemens, R.J. Schutten, K.A. Vissers

Philips Research, Prof. Holstlaan 4, 5656 AA Eindhoven, Netherlands
email: riemens@natlab.research.philips.com

Abstract

The display of a regular TV picture on a PC-related monitor requires scan rate conversion. We perform this conversion by a visually attractive solution based on a median filter, requiring in the order of 10 operations per pixel and one field memory. This algorithm is implemented as an optimized C program that executes on a Philips TriMedia TM1000 VLIW processor in real time.

It is shown that through programming in C, all architectural features of the TM1000 can be efficiently exploited. A structured approach, including custom operations, function inlining, loop unrolling and software pipelining, is used to come to a very efficient program. The performance is boosted by a factor of 10, and the instruction-level parallelism (ILP) is increased from 1.3 to 4.2. Furthermore, the introduction of one additional special instruction would reduce the computational load by an additional factor of 2.

Keywords: scan rate conversion, median filter, VLIW, high level programming, video signal processing

1 Introduction

This paper focuses on the programming and optimization method of a state-of-the-art media processor for a real-time applications. This subject is discussed based on a case study. The application we selected was scan rate conversion. In this case study, the goal was to write a program that performs at least in real time on a TriMedia TM1000 processor which runs at 100MHz. The target system is a PC with a TM1000. In this paper we first present the processor. Next we discuss algorithms for scan rate conversion and then select one of them. The remainder of the paper focuses on our programming and optimization method.

1.1 The processor

TriMedia [1] is a family of VLIW (very long instruction word) processors of Philips Semiconductors in Sunnyvale. The first product, the TM1000 processor, is targeted towards multimedia applications but it also has good general purpose computing capabilities. This media processor can be programmed to execute a number of functions in real-time, e.g. MPEG2 decoding, video conferencing, audio processing and 3-D graphics. A block diagram is shown in fig. 1.

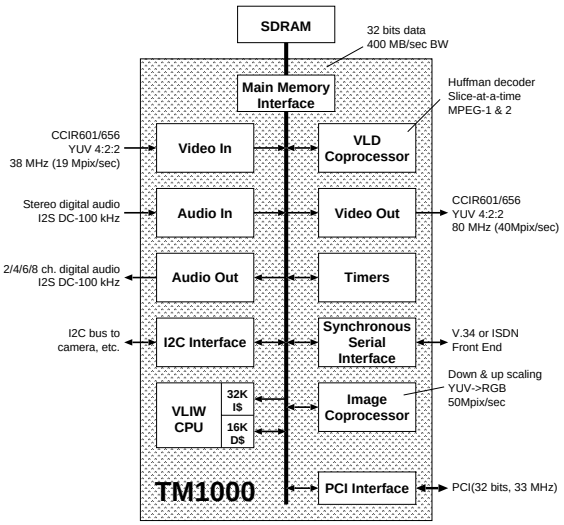

fig. 1 TM1000 block diagram

The TM1000 processor consists of a central, internal data bus that connects all internal blocks and provides access to a PCI bus and SDRAM. Examples of internal blocks of the TM1000 are a 32 bit VLIW CPU core with 16 Kbyte data cache and 32 Kbyte instruction cache, an image coprocessor, and input and output units for audio and video. The CPU contains 128 general purpose registers.

In this case study, a video signal is connected to the "video in" peripheral of the TM1000. This peripheral performs interrupt-driven DMA. The

"image coprocessor", which also performs interrupt driven-DMA, handles the output and sends the data immediately through the PCI bus.

Apart from being a powerful general purpose processor, the TriMedia TM1000 core supports dedicated video processing through a number of features. 5 issue slots can execute in parallel, supporting instruction level parallelism (ILP). Within an issue slot, single instruction multiple data (SIMD) parallelism is supported on dedicated types for video, e.g. 8 bit and 16 bit numbers. Furthermore, dedicated instructions are present to support median filtering, e.g. special minimum and maximum instructions that operate in a SIMD fashion.

All these architectural options can be exploited using the C programming language. The programming of such a VLIW processor should not be done at the assembly level. The compiler can conveniently and very efficiently take care of all details like register allocation, memory organization and instruction scheduling, taking branch delays and pipeline effects into account.

An important characteristic for VLIW architectures is that the compiler determines which operations are issued together. Each VLIW instruction contains 5 instructions. When there is no useful operation, an instruction is empty (*nop*). One new VLIW instruction is issued each clock cycle. The pipeline only stalls at cache misses.

1.2 The algorithm

If one wants to display a standard video signal on a PC monitor, the problem of *scan rate conversion* arises. Video signals from any normal source comply with the broadcast standard (NTSC or PAL). The PC monitor, however, fails in two important ways to comply with this standard. First of all, current broadcast standards are interlaced: each frame consist of two fields, where the first field only contains the odd lines of the image and the second one contains only the even lines. A PC monitor accepts progressive pictures: each frame contains all lines. The second difference is the screen refresh frequency. In the broadcast standard, there are 50 or 60 fields/sec. On the PC, however, the refresh rate depends on the capabilities of the monitor and settings of the graphics card. It is a frequency in the order of 60 to 100 frames/sec.

This paper deals with mapping the de-interlace step onto a TriMedia processor. De-interlacing is the task of calculating the odd lines from an even field and vice versa. Many methods for de-interlacing exist with different levels of performance and complexity [2][3].

On the low-end side of the performance scale are the algorithms that perform line repetition or line averaging (both are intra-field interpolation methods). On non-moving sequences the result of these algorithms still suffers from the original 25 or 30 Hz line flickering.

Another possibility is line insertion. Here the missing lines are copied from the same vertical position from the previous field (this is an inter-field interpolation method). On non-moving sequences this algorithm performs very well. However, even with just slightly moving sequences annoying artifacts become visible.

On the high-end side of the performance scale are the motion compensated methods that use information from the past, shifted according to the appropriate motion vector. However, these methods are highly demanding with respect to required compute power and required memory quantity and bandwidth.

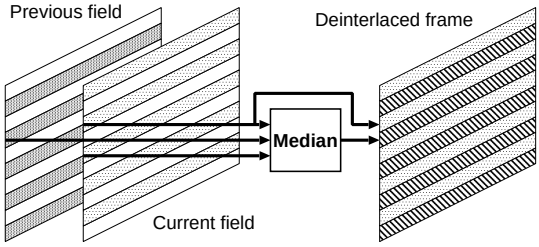

fig. 2 Median filter de-interlacing algorithm

The median filter de-interlacing algorithm combines the benefits of line repetition and line insertion, eliminating the typical artifacts of the individual algorithms [4]. Pixels in the missing lines are calculated by taking the median of two pixels from the neighboring lines in the current field, and of one pixel from the line on the same vertical position in the previous field. This is depicted schematically in fig. 2.

In non-moving sequences the median will usually select a pixel from the line in the previous field (equivalent to line insertion). In moving sequences it is very likely that this pixel has an ex-

treme value compared with the other two pixels in the median, so a pixel from the current field will be selected (equivalent to line repetition). Therefore, this algorithm combines a good perceptual performance with acceptable computational cost [2].

Dedicated hardware solutions for this algorithm exists (e.g. Philips IC SAA4990). In this case study, we focus on a software solution. This illustrative application of the median filter, is also part of more advanced scan rate conversion techniques. The TM1000 however is not only geared to such basic algorithms, but also to MPEG2 decoding, 3-D graphics processing, video conferencing etc.

2 Optimization process

We focus in the remainder of this paper on the program that actually performs the calculations of the median function. We use a host system with an instruction set simulator as a development environment, as shown in fig. 3.

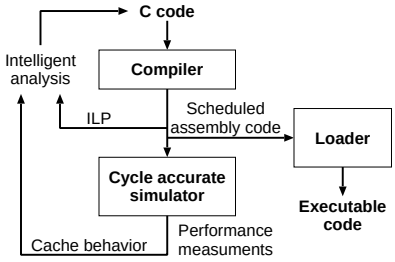

fig. 3 Tool chain

2.1 Methodology

The optimization consists of an iterative loop using this development environment as follows:

- Write / modify the C program.
- Run the C compiler (“tmcc”).
- Analyze the generated assembly file(s) to get insight into the amount of instruction level parallelism, data dependencies, branch delay slots, etc.
- Run the cycle accurate simulator “tmsim”, which also simulates the cache behavior. This gives the number of CPU cycles actually spent in each function, as well as the number of stall cycles caused by cache misses. This simulator also generates output files, making the verification of functional equivalence in each iteration easier.
- Based on the information acquired in the previous steps and on knowledge of the TM1000 architecture, it is possible to understand what’s going on in the machine. This can lead to suggestions of how to improve the program. Consequently, the C code can be adapted to verify these suggestions.

Note that all programming effort is done in the C language. Assembly language is only used to analyze the code generated by the compiler.

When analyzing the program behavior, only three performance indicators are used. These are sufficient for the focus for further improvements. They are:

- The number of cycles for the test run (measured by the simulator). This number is normalized to the real-time requirement, resulting in a percentage of CPU load.
- The instruction level parallelism (ILP). This is simply derived from the assembly files by counting the number of cycles and the number of *nop* instructions. The theoretical maximum of the TM1000 is 5, since it is a 5 issue slot VLIW machine.
- The number of cache stalls (also measured by the simulator). Especially the data cache may be critical. This number in table 1 gives the percentage of cycles of the total program cycles that are spend as stall cycles.

The next sections describe each of the optimization steps, where we focus on the following issues *in a specific order*:

- First: reduce the number of executed instructions (steps 1 and 2).
- Next, utilize SIMD (single instruction, multiple data) parallelism (step 3).
- Then, increase the ILP (instruction level parallelism) (steps 4 and 5).
- And finally, tune the data cache behavior (step 6).

2.2 Reference code

The optimization process starts with reference code as shown below[1]. The “median” operation is implemented as a function, while the loops over all pixels of all lines are clearly illustrated.

1. This code is valid for the odd field only; processing the even field is slightly different. The computational load in both cases is equal, so this issue is not mentioned again.

```
pixel_t Median (pixel_t a, pixel_t b, pixel_t c)
{
  pixel_t  result, max, min;

  max = a>b ? (a>c ? a : c) : (b>c ? b : c);
  min = a<b ? (a<c ? a : c) : (b<c ? b : c);
  result = a + b + c - min - max;
  return(result);
} /* end of Median */

/* inputs: cur_fld, prv_fld;       outputs: out_frm */
for (lin = 0; lin < lpf - 1; lin++) {        /* lines per field */
  for (pix = 0; pix < ppl; pix++) {       /* pixels per line */
    out_frm[2*lin*ppl+pix] = cur_fld[lin*ppl+pix];
    out_frm[(2*lin+1)*ppl+pix] = Median(cur_fld[lin*ppl+pix],
                cur_fld[(lin+1)*ppl+pix], prv_fld[lin*ppl+pix]);
  }
}
```

The results of the reference code and successive optimization steps are in table 1. The reference code runs 5.94 times slower than real time on a TM1000 running at 100MHz. We denote this in the table as 594%.

In general, it is extremely important to focus on the data flow first. Unnecessary storage of intermediate data in the background memory should be avoided. There is no intermediate data storage in the reference code.

2.3 Step 1: use custom operations

When analyzing the compilation result (assembly files) of the reference version, it appears that the function “Median” causes a lot of overhead because of the function call and the if-then-else branches. This function is simplified significantly by rewriting it as a macro using *imin* and *imax* instructions. These instructions return the lowest and highest values of the two operands, without the need of a conditional test and jump instruction. These special instructions are put in the C code as functions. They are supported by the TriMedia compiler and will compile to a single assembly instruction, taking only one issue slot. They are examples of the instruction set of the TM1000 containing special instructions for DSP and multimedia operations. The macro looks like:

```
#define Median(a,b,c)\
        (imin(imax( imin((a),(b)), (c)), imax((a),(b))))
```

Here the inner loop body contains only one piece of linear code. As a result of this step, the algorithm runs at 319% of a TM1000@100MHz.

2.4 Step 2: improve loop

At this point, there is one inner loop body that still has to be optimized. Two changes are done in this step.

- Move as many calculations as possible from the loop body to the loop preamble. This reduces the number of executed instructions.
- Remove false data dependencies. In this case study, all pointers reference distinct memory locations (no aliasing), so the compiler does not need to assume any data dependency when dereferencing these pointers. *Restricted pointers* are available for this purpose. Removing the data dependency will increase the instruction level parallelism.

  Now, we have achieved 220%.

2.5 Step 3: reduce load/store activity

Up till now, we mainly focused on reduction of the number of executed instructions. Now we start to focus on SIMD parallelism as well.

```
#define Median(a,b,c)\
        (imin(imax( imin((a),(b)), (c)), imax((a),(b))))

#define Median_Q(a,b,c) (pack16lsb( packbytes( \
  Median(ubytesel((a),3), ubytesel((b),3), ubytesel((c),3)), \
  Median(ubytesel((a),2), ubytesel((b),2), ubytesel((c),2))), \
packbytes( \
  Median(ubytesel((a),1), ubytesel((b),1), ubytesel((c),1)), \
  Median(ubytesel((a),0), ubytesel((b),0), ubytesel((c),0))) \
))
```

In the result of step 2, load and store operations are performed on each pixel, i.e. on each byte. When they are performed on each machine word (32 bits), one operation loads or stores four pixels. This is SIMD parallelism. Furthermore, the loop overhead is reduced, since one loop iteration is performed on every four pixels now. The penalty for this approach is that the data has to be unpacked into separate registers before calculating the median and the results have to be packed into a word again after the calculation as shown below. The result of this step is significant: 93% of TM1000@100MHz.

2.6 Step 4 & 5: unroll loop

In these steps, the main focus is on improvement of the instruction level parallelism (ILP).

Loop unrolling is done step by step to achieve a sufficiently high ILP. The loop is unrolled twice in each of the steps 4 and 5. This has two advantages: first of all the loop overhead is reduced to

25%. Furthermore, the total number of instructions in the loop body is increased, which results in an increased ILP. After these steps, the function runs at 73% of TM1000@100MHz.

2.7 Step 6: improve cache behavior

In this final step, we focus on data cache behavior.

In the result achieved so far, 27% of the CPU cycles are spilled on data cache stalls. Both the two input buffers and the output buffer are cache block aligned[1]. Since the median function reads and writes the data sequentially, four pointers (two input fields, two lines in the output frame) will hit a cache block boundary at the same time. This causes a high temporal locality of the data bus traffic (the program requests bursts of data bus traffic). The data cache is optimized for a uniform temporal distribution of cache misses. Therefore, a good temporal spreading of the cache misses will reduce the number of data cache stalls. The performance can be improved by adding "prefetch" instructions with different address offsets.

The resulting code runs at 68% of TM1000@100MHz.

Even though the program length has increased by adding the prefetch instructions, the number of cycles for the loop body has remained the same, because *nop* instructions are replaced by useful ones. The generated code occupies 2.2 Kbyte, so it fits easily in the 32 Kbyte instruction cache.

```
#define Median(a,b,c)\
        (imin(imax( imin((a),(b)), (c)), imax((a),(b))))

#define Median_Q(a,b,c) (pack16lsb( packbytes( \
   Median(ubytesel((a),3), ubytesel((b),3), ubytesel((c),3)), \
   Median(ubytesel((a),2), ubytesel((b),2), ubytesel((c),2))), \
packbytes( \
   Median(ubytesel((a),1), ubytesel((b),1), ubytesel((c),1)), \
   Median(ubytesel((a),0), ubytesel((b),0), ubytesel((c),0))) \
))

i_cf_cur_lin = cur_fld;          /* current field, current line */
i_cf_alt_lin  = cur_fld + ppl;   /* current field, next line */
i_pf_cur_lin = prv_fld;          /* previous field, current line */
o_cur_lin    = out_frm;          /* out frame, current line */
o_nxt_lin    = out_frm + ppl;    /* out frame, next line */
for (lin = 0; lin < lpf - 1; lin++) {          /* lines per field */
   cf_cl_pix = i_cf_cur_lin[pix];
   cf_al_pix = i_cf_alt_lin[pix];
   pf_cl_pix = i_pf_cur_lin[pix];
   for (pix = 0; pix < wpl; pix += 4) {        /* words per line */
      PREFETCH((char *)o_nxt_lin, 1);
      med_pix = Median_Q(cf_cl_pix, pf_cl_pix, cf_al_pix);
      o_cur_lin[0] = cf_cl_pix; o_nxt_lin[0] = med_pix;
      cf_cl_pix = i_cf_cur_lin[pix+1];
      cf_al_pix = i_cf_alt_lin[pix+1];
      pf_cl_pix = i_pf_cur_lin[pix+1];
      med_pix = Median_Q(cf_cl_pix, pf_cl_pix, cf_al_pix);
      o_cur_lin[1] = cf_cl_pix; o_nxt_lin[1] = med_pix;
      PREFETCH((char *)(&i_pf_cur_lin[pix+8]), 1);
      cf_cl_pix = i_cf_cur_lin[pix+2];
      cf_al_pix = i_cf_alt_lin[pix+2];
      pf_cl_pix = i_pf_cur_lin[pix+2];
      med_pix = Median_Q(cf_cl_pix, pf_cl_pix, cf_al_pix);
      o_cur_lin[2] = cf_cl_pix; o_nxt_lin[2] = med_pix;
      cf_cl_pix = i_cf_cur_lin[pix+3];
      cf_al_pix = i_cf_alt_lin[pix+3];
      pf_cl_pix = i_pf_cur_lin[pix+3];
      med_pix = Median_Q(cf_cl_pix, pf_cl_pix, cf_al_pix);
      o_cur_lin[3] = cf_cl_pix; o_nxt_lin[3] = med_pix;
      PREFETCH((char *)(&i_cf_alt_lin[pix+10]), 1);
      o_cur_lin += 4; o_nxt_lin += 4;
      cf_cl_pix = i_cf_cur_lin[pix+4];
      cf_al_pix = i_cf_alt_lin[pix+4];
      pf_cl_pix = i_pf_cur_lin[pix+4];
   }
   i_cf_cur_lin += wpl; i_cf_alt_lin += wpl; i_pf_cur_lin += wpl;
   o_cur_lin += wpl; o_nxt_lin += wpl;
}
```

2.8 Suggestions

It is clear that the performance of the processor could be improved significantly if the median calculation were to be simplified. Therefore we propose *quadmin* and *quadmax* instructions, performing a *min* and a *max* operation on each of the bytes in the 32 bit word. When we use these instructions, the median macro becomes:

```
#define Median_Q(a,b,c)\
    (quadmin(quadmax( quadmin(a,b), c), quadmax(a,b)))
```

The original macro contains 31 min/max operations; the proposed one only 4! An experiment was conducted to emulate these instructions in the program. The complete program with the new instructions runs at only 30% of such a TM1000 at 100MHz!

If the computation was reduced this far, the data cache would take at least 30% of the machine cycles. The current implementation of the data cache in the TM1000 is such that prefetch instructions still cause some stall cycles. Small extensions of the current cache design can increase the effectiveness of prefetches even further.

3 Results

The results after each optimization step are

1. Caused by constraints of the video input unit.

shown together in table 1. Two columns of this table are also shown in fig. 4 and fig. 5. They clearly show the progress of each step.

Step	TM1000@100MHz	ILP	D$ stall
Ref.	594%	1.9	4.8%
1	319%	1.3	6.2%
2	220%	1.7	9.0%
3	93%	2.8	18%
4	78%	3.7	26%
5	73%	4.0	27%
6	68%	4.2	20%

table 1 Performance indicators after each step

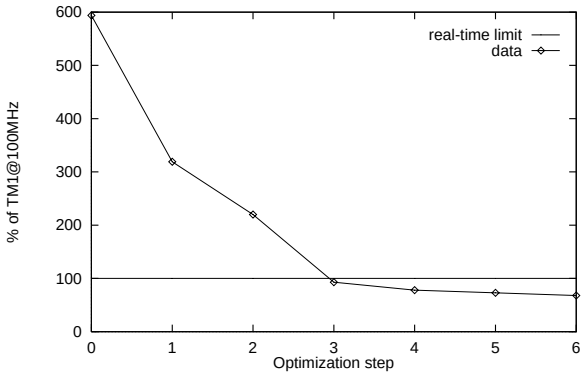

fig. 4 Percentage of TM1000 after each step

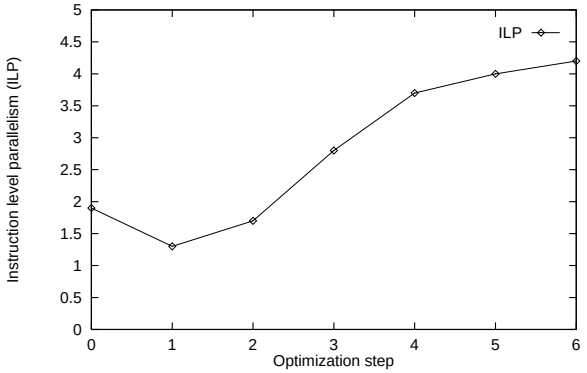

fig. 5 ILP after each step

The method here is first to reduce the number of executed instructions and then focus on ILP and finally on cache behavior. It would not make sense to focus on optimizing ILP before reducing the number of instructions, since the drastic changes in the structure of the program and number of instructions that results from this reduction will have a large impact on the ILP. The cache behavior can only be analyzed in the end, since this is dependent on the sequence of actual data accesses.

4 Conclusions

This experiment shows the importance of programming at a high level while still being able to exploit dedicated features in a VLIW architecture. Furthermore, the gap between applications and architectures is bridged in a novel and productive manner for industrially relevant problems.

The experiments with the new instructions and the suggestions for modified caches show that significant performance improvements can be achieved when developing in applications, compilers and architectures together.

The application of the outlined method will allow application programmers to write efficient programs for the TM1000, using only the high level language C. This is a significant improvement over current practice in programming DSPs or other SIMD processors, which is usually done in assembly language, for various reasons [5][6].

References

[1] S. Rathnam and G. Slavenburg, *"An architectural overview of the programmable multimedia processor TM1"*, Proc. Compcon, IEEE CS Press, 1996.

[2] T. Doyle and M. Looymans, *"Median filtering of television images"*, Proc. ICCE, June 1986, pp. 186-187.

[3] G. de Haan, P.W.A.C. Biezen, O.A. Ojo, *"An Evolutionary Architecture for Motion-Compensated 100 Hz Television"*, IEEE Transaction on Circuits and Systems for Video Technology, Vol. 5, No.3, June 1995.

[4] M.J.J.C. Annegarn, T. Doyle, P.H. Frencken, D.A. Van Hees, *"Video signal processing circuit for processing an interlaced video signal"*, US patent no. 4740842, 26 April 1988.

[5] Vojin Zivojnovic et. al. *"DSPs, GPPs And Multimedia Applications - An evaluation Using Dspstone"*, ICSPAT 95, pp. 1779 -1783.

[6] *"Using MMX Instructions to Implement Median Filter"*, http://www.intel.com/drg/mmx/appnotes/ap552.htm