Implementation of the AVS Video Decoder on a Heterogeneous Dual-Core SIMD Processor

Maria Koziri, *Member, IEEE*, Dimitrios Zacharis, Ioannis Katsavounidis, *Member, IEEE*, and Nikos Bellas

Abstract — *Multi-core Application Specific Instruction Processors (ASIPs) are increasingly used in multimedia applications due to their high performance and programmability. Nonetheless, their efficient use requires extensive modifications to the initial code in order to exploit the features of the underlying architecture. In this paper, through the example of implementing Advance Video Coding (AVS) to a heterogeneous dual-core SIMD processor, we present a guide to developers who wish to perform task-level decomposition of any video decoder in a multi-core SIMD system. Through the process of mapping AVS video decoder to a dual-core SIMD processor we aim to explore the different forms of parallelism inherent in a video application and exploit to speed-up AVS decoding in order to achieve real time functionality. Simulation results showed that the extraction of parallelism at all levels of granularity, especially at the higher levels, can give a total speed-up of more than 195× compared to a software x86-based implementation, which enables real-time, 25fps decoding of D1 video*[1] .

Index Terms — AVS, video decoder, SIMD processor, multi core processor.

I. INTRODUCTION

In the last few years, we have seen the emergence of a number of video standards for applications spanning from wireless low-rate, to high definition broadcast video. These systems have been implemented with a variety of single core or multi-core technologies from general purpose processors (GPPs) to fixed ASICs. Industry's demands for high quality, high resolution, real-time video decoding, usually under low-power constraints, is a challenging task which continues to tax the ability of multimedia architectures to deliver a cost effective solution.

High performance processors offer a flexible solution by implementing video standards in software and hiding the underlying hardware organization from the application developer. Multimedia ISA extensions, like MMX, SSE etc., feature vector operations to exploit the data-level parallelism, which is abundant in such applications [1][2]. However, using multi-GHz processors [3] is out of the question for embedded systems due to the "power wall" problem and the high cost. At the other extreme, fixed ASICs target special cases that require very high throughput or very low power dissipation, yet they suffer from little or no programmability and high development costs [4].

Application Specific Instruction Processors (ASIPs) such as DSPs or multimedia processors are used to bridge these two extremes by combining the best of two worlds: programmability similar to GPPs and performance close to ASICs within a particular application domain. ASIPs are an appealing solution for applications with evolving standards that allow a high degree of added value on algorithmic IP innovation. Their superior efficiency comes at the cost of forcing the developer to think about how the application can be optimally mapped into the underlying architecture. The main problem with multi-core platforms is that modern compilers and run-time systems offer little or no help in extracting task level parallelism from the application.

This paper describes the porting of the AVS (Advanced Video Standard) [5] video decoder to a heterogeneous dual-core SIMD processor. AVS was drafted by the AVC work group of China to replace older and royalty-burdened standards such as MPEG-2 and H.264 mainly in consumer applications. The dual-core SIMD processor used for this work includes preconfigured versions of a 32-bit configurable architecture optimized for video encoding and decoding. Its enhanced instruction set supports all popular video codecs such as MPEG-4, H.264, VC-1 (all in Main Profile) for performance up to D1 resolution, i.e. 720×576×25 (PAL) or 720×480×30 (NTSC) pixels/sec. The procedure followed to port AVS decoder to this processor was to start from an open source implementation of the AVS decoder, OpenAVS, which targets a GPP platform, and to gradually transform the code so as to enable a dual core implementation on a SIMD processor. This procedure may serve as a guide to developers who wish to perform task-level decomposition of any video decoder in a multi-core system, while most of the optimization techniques presented in this paper are generally applicable to any SIMD processor. Our aim is to achieve real time, 25 fps, progressive D1 resolution (720×576) AVS video decoding.

The challenge of mapping a new video decoder in a heterogeneous multi-core engine is to detect and extract parallelism at all levels of granularity, especially at the higher levels. In this work we explain how different forms of

[1] M. Koziri is with the Department of Computer and Communication Engineering, University of Thessaly, Volos, 38221, Greece (e-mail: mkoziri@inf.uth.gr).

D. Zacharis is with the Department of Computer and Communication Engineering, University of Thessaly, Volos, 38221, Greece (e-mail: dzaharis@inf.uth.gr).

I. Katsavounidis is with the Department of Computer and Communication Engineering, University of Thessaly, Volos, 38221, Greece (e-mail: ioannis.k@inf.uth.gr).

N. Bellas is with the Department of Computer and Communication Engineering, University of Thessaly, Volos, 38221, Greece (e-mail: nbellas@inf.uth.gr).

Contributed Paper
Manuscript received 01/18/11
Current version published 06/27/11
Electronic version published 06/27/11.

0098 3063/11/$20.00 © 2011 IEEE

parallelism at the block level (instruction and SIMD parallelism) and at higher levels (task and pipeline parallelism) are exploited by the specific core, and we analyze the contribution of each form of parallelism in the total speed-up.

The rest of the paper is organized as follows. In Section II we briefly present the AVS Video Standard and the architecture of the target platform. Section III is dedicated to the process of porting AVC to target platform. More precisely, in Section III.A we describe the software optimizations made and in Section III.B the block-level parallelism. Section III.C illustrates the task-level parallelism, while Section III.D summarizes the overall optimizing process. Finally, Section IV concludes the paper.

II. AVS Video Standard And The Target Platform

A. AVS Video Standard

The Audio Video Coding Standard work group of China developed the Advanced Video Coding Standard (AVS), the first audio video standard developed by China independently. The first draft, which was completed in 2003, initially targeted at high definition, high quality broadcast and digital media storage applications. Due to its limited target, AVS achieves both higher efficiency and lower complexity compared to other video coding standards such as MPEG-2 [6], MPEG-4 [7] and H.264/AVC [8].

Although, from functional modules point of view, AVS is similar to H.264/AVC, as shown in Fig. 1, the techniques, used by AVS to implement each module, differ from those used in H.264. Some of these differences are presented bellow, while a complete overview of them can be found in [9].

- Entropy Coding: AVS uses k^{th} -order Exp-Golomb codebook (k=0, 1, 2, 3). It defines 19 mapping tables in order to map the coded symbols to the elements of Exp-Golomb codebooks more efficiently. The major improvement is that because of the regularization of the Exp-Golomb codebooks, the AVS decoder does not need to store these codebooks.
- Transform and Quantization: In order to reduce rounding errors, dequantization and inverse transform are considered in one process. AVS, unlike H.264/AVC, uses an 8×8 integer transform.
- Intra Prediction: Intra-frame prediction in AVS is performed in 8×8 luma/chroma blocks. AVS defines a total of 9 modes, whereas in H.264/AVC there is a total of 17 modes.
- Motion Compensation: AVS uses Variable Block Size Motion Compensation (VBSMC), with 4 block sizes, 16×16, 16×8, 8×16 and 8×8. The number of reference pictures is limited to maximum two.
- Deblocking Filter: AVS defines an adaptive in-loop deblocking filter to reduce blocking artifacts due to block-based coding. The filtering is applied to the boundaries of luma and chroma blocks, except for the boundaries of picture or slice.

To give an indication of the relative complexity of the various modules, we profiled the OpenAVS code in a baseline Xtensa RISC processor with perfect (zero-wait) memory using an AVS input bitstream compressed at approximately 4 Mbps. Fig. 2 shows that Motion Compensation (MC) contributed almost 2/3 of the total execution time, whereas the second most computationally complex function is Deblocking filter (DB) with 12.9%.

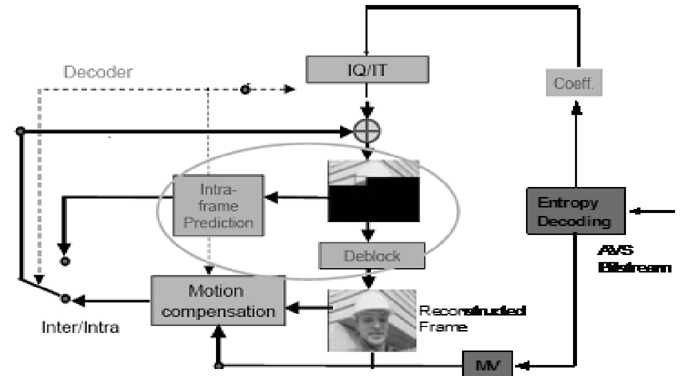

Fig. 1. Block diagram of the AVS decoder.

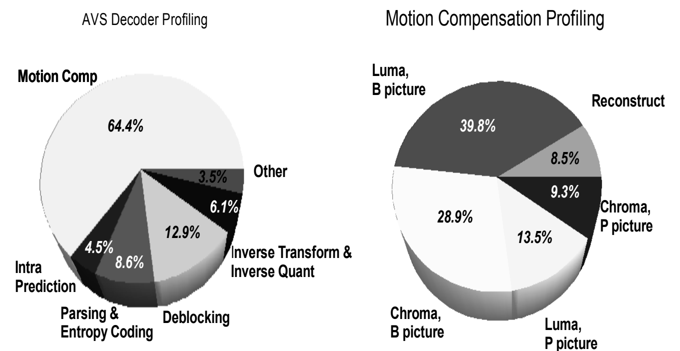

Fig. 2. Execution profiling of the software OpenAVS decoder (a) Motion Compensation (64.4% of the total execution time) (b) Details on the Motion Compensation profiling.

B. Target Platform

As shown in Fig. 3, the target platform is a heterogeneous dual core processor which consists of two different cores with video specific instruction extensions. The instruction set of the specific processor's architecture is optimized for embedded designs. The base 32-bit architecture has a 32-bit ALU, up to 64 general-purpose physical registers, 6 special purpose registers and 80 base instructions, including compact 16- and 24-bit (rather than 32-bit) RISC instruction encoding. One of the main features of the specific processor is the ability of extension through the usage of TIE, which can lead to acceleration of processor's performance. TIE is a Verilog-like language used to describe new instructions, new registers and execution units, and new I/O ports that are then automatically added to the processor. New elements (instructions, registers, etc.) described by TIE language, are called Tensilica Instructions Extension(s), or, in abbreviation, TIE(s). In the rest of the paper, the abbreviation TIE(s), will be used to indicate these new elements and not the language used to describe them.

The two cores in our dual-core processor, referred to as Stream processor (hereinafter SP) and Pixel processor (hereinafter PP), are enhanced with simple SIMD and video oriented TIEs, which are used to accelerate compute-intensive and data parallel hot spots in the code [10]. A SIMD TIE is, for example:

Res = xvd_add_16x12(A,B)

which adds two 16-element vectors, each being up to 12-bits in size. A rich repertoire of SIMD arithmetic, logic, and load/store instructions provide data parallel execution on vectors of 8 or 16 elements to match the sizes of block and macroblocks, respectively. An example of a video oriented TIE is:

data = xvd_bs_loadgetbits(numbits)

which extracts *numbits* bits from the input bitstream, and places them in the variable *data* for further processing.

The processor used in this work is preconfigured with video specific TIEs used for older video codecs, such as MPEG-4 or H.264. These instructions are optimized for the most performance-intensive algorithms used in video processing, including: CABAC/CAVLC (used in H.264), deblocking, transforms (especially for the 4×4 integer transform of H.264), motion compensation and motion estimation algorithms. Nonetheless, because these TIEs were designed and optimized for specific standards (H.264, MPEG-4, VC-1), they cannot be used efficiently for AVS. Unfortunately we were not allowed to implement new TIEs, therefore, we focused on making an optimal reuse of the available TIEs to accelerate the AVS decoder.

The main task of the SP is to parse and decode the video bitstream. It has TIE extensions to speed-up the parsing and decoding of the video headers, motion vectors, and transform coefficients and (optionally) to perform inverse quantization. The SP is based on a 5-stage pipeline, and has two tightly coupled local SRAMs: one 40 KB Instruction SRAM and one 32 KB Data SRAM. Placing instructions and data in these SRAMs is vital to achieve good performance as we will examine in the next section.

The PP performs most of the heavy duty computations of video decoding using SIMD TIE instructions. It is used to accelerate motion compensation (including quarter pixel interpolation and reconstruction), intra prediction, inverse quantization (optionally), inverse transform and the deblocking filter. PP has one 24 KB Instruction SRAM and one 40 KB Data SRAM. Fig. 4 shows how the different modules of the decoding process are shared between SP and PP.

Another element of the target platform, which is coupled with PP is the Transpose unit, named TIEQT in Fig. 3. TIEQT converts columns to rows, as shown in Fig. 5, allowing computations among columns, which is a necessity in video coding. In this way, the parallelization is increased, as SIMD instructions can be used even for columns (after the last are transposed to rows by TIEQT).

Data transfers between the local SRAMs of the two cores and between the SRAMs and main memory is accomplished with a multichannel DMA engine which runs asynchronously to the execution cores. Any of the two cores can set up and initiate a 2D DMA transaction by describing, among other things, the size of the memory access patterns, source and destination addresses, and the priority schemes between channels.

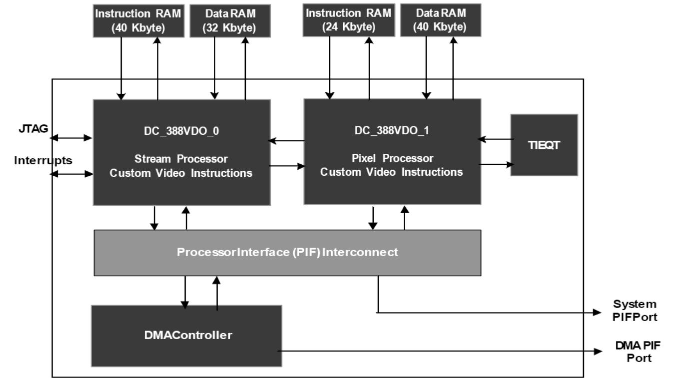

Fig. 3. Heterogeneous Dual-Core Video Engine Block Diagram.

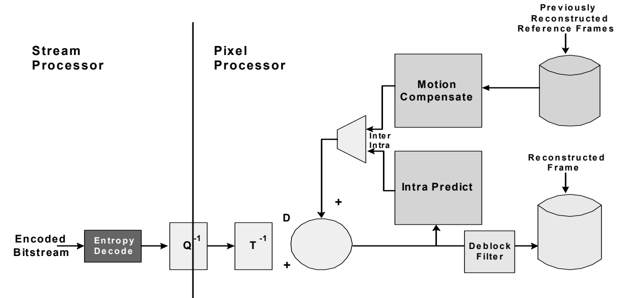

Fig. 4. Decoder's tasks' partitioning between Stream and Pixel processors.

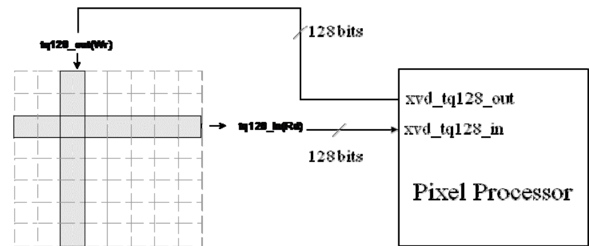

Fig. 5. Conversion of columns to rows by the TIEQT unit.

III. AVS VIDEO STANDARD OPTIMIZATION

This section describes the steps taken to transform the un-optimized OpenAVS code to a dual core implementation optimized for our target platform.

The OpenAVS code was profiled on the baseline processor of the target platform and the results indicated that the baseline processor will have to be clocked at 7.8 GHz to meet the desired performance requirements under a perfect (i.e. no latency) memory scenario. The same code, when benchmarked under the more realistic, non-perfect memory with default read latencies, resulted in an exorbitant 70GHz clock requirement, as shown on Table I – 1st row. This is not surprising, since video decoding is a memory-intensive operation and the target platform offers no cache, which means all operations are impacted from the slow memory subsystem. This phenomenon

can be emphasized if we assume 64-read, 32-write wait states, which models most of low-cost memory subsystems, resulting in 700.7 GHz (Table II – 1st row).

Clearly, there is a large potential for software optimizations of the AVS decoder using the TIEs and other code transformations.

We followed a two-phase approach: First, optimization of OpenAVS decoder targeting a single core SIMD processor and then mapping the optimized single core code to the target platform. The first phase includes software optimization to exploit the available block level parallelism of the algorithm. Optimization techniques used in this phase are generally applicable to any SIMD processor. In the second phase, the code is partitioned into tasks based on the architecture of the target platform (Fig. 1). Task-level decomposition includes extracting task-level parallelism, and orchestrating data exchanges between communicating threads running on the two cores. Since the two cores of the target platform are optimized for specific tasks (Fig. 4), the decomposition strategy is not directly applicable to any multi-core system. Nevertheless, the general task partitioning scheme can serve as yardstick for developers attempting to perform task-level decomposition of a video decoder in a multi-core system.

In the rest of this section we present the main techniques used in software optimization, block level parallelism and task parallelism.

A. Software Optimization

Our software optimization strategy focuses on the most expensive functions of the AVS decoder: motion compensation, deblocking filter, inverse transform, bitstream parsing and variable length decoding, and intra prediction.

The main effort in this task is to improve the memory access behavior of the original AVS decoder. The introduced changes (i) increase spatial locality of memory accesses, which is crucial in task level parallelism; (ii) eliminate frame-based computation that requires expensive main memory accesses; (iii) restructure the code to optimize data reuse in the internal SRAMs. Frequently accessed data structures are identified and explicitly copied to SRAM memory locations, and some computations are converted from frame-based to block (or macroblock)-based, and therefore to eliminate unnecessary data spilling to the main memory. This last step provided the maximum performance improvement for non-perfect memory, since frequently used data became immediately available.

For example, after inspecting the profiling results of the un-optimized OpenAVS, we noticed that the luma and chroma interpolation functions should be restructured. The pixel data that need to be interpolated are fetched from external memory, which is a particularly expensive operation. For an 8×8 luma block, we need to access a large number of integer pixels due to quarter pixel interpolation. For example, referring to Fig. 6:

- To compute the integer pixel *D,* we only need *D* itself.
- For each half pixels *b* and *h*, we require the access of 4 integer pixels (C,D,E,F for *b*).
- For each half-pixel *j*, we require the access of 16 integer pixels.
- For each quarter pixel *a, c, d* and *n*, we require the access of 10 integer pixels, etc. Note that some of these pixels may be the same.

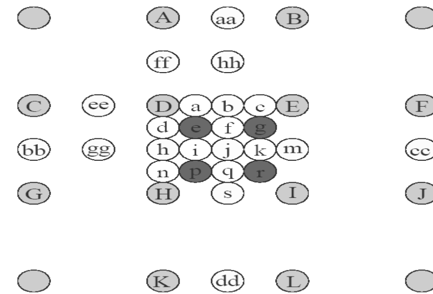

Fig. 6. Interpolation of Luma components.

Assuming a uniform distribution among all 16 possible pixel positions we have an average of (1+2*4+16+4*10+4*17+4*20)/16 = 13.3125 integer pixel accesses per pixel, or 852 accesses per 8×8 block. We exploit this spatial locality by moving the bytes of the block from external memory to the local SRAM of the pixel processor, and reuse them from there as many times as needed, with no wait cycles.

Another optimization example concerns the restructuring of the code so as to convert computations from frame-based to block-based, and, therefore to eliminate unnecessary data spilling to the main memory.

As shown in Fig. 7, OpenAVS includes two main loops to process a frame. The two separate loops are needed since, according to the AVS standard, Motion Compensation should be applied on macroblocks that have not been processed by the deblocking filter.

```
for (MB_index=0; MB_index <num_of_MBs; MB_index++)
        ParseOneMacroblock;  //Parsing, VLD
        McIdctRecOneMacroblock;  //MC, Inverse transform, Reconstruction
endfor
for (MB_index=0; MB_index <num_of_MBs; MB_index++)
        DeblockOneMacroblock;  //Deblocking
endfor
```

Fig. 7. Pseudo-code for the two main loops in OpenAVS.

If the *DeblockOneMacroblock* function is called in the first loop immediately after *McIdctRecOneMacroblock*, the inputs to the deblocking filter for MB(col,row) would have already been filtered, thus violating the AVS standard. The problem with the reference OpenAVS code is that it causes spill of the whole frame to the main memory: the output frame of the MC has to be stored to the main memory, and then retrieved back from the deblocking filter.

We introduce a data structure that stores the three pixel rows above and the three pixel columns on the left of the macroblock MB(col,row) after MC. These 720×3×2 + 16×3×2 = 4416 bytes are the only pixel data needed for the deblocking filter of MB (col, row). By using these pixel data as inputs to

the deblocking filter, we can fuse the two loops, and avoid spilling a whole frame to the main memory. This optimization is similar to loop tiling which is frequently used by optimizing compilers to improve spatial locality in the cache hierarchy of a processor [11].

Significant performance can also be achieved by reducing check points, i.e. branches, in the code. The initial OpenAVS code made extensive use of the *min* and *max* functions, in the Motion Compensation module, to crop the motion vector (MV) access range within the frame. We eliminate these checks by explicit frame boundary extension (also known as frame padding) when we write the reconstructed frame back to main memory. Although this change entails extra memory accesses to store the out-of-bound pixels, the overhead is much less than the over-use of checks in the initial OpenAVS code. It should be noted that a smarter DMA controller could automatically extrapolate data for motion vectors beyond frame boundaries and eliminate the need of the software overhead for checks or the extra bandwidth from frame padding, however the target platform's DMA controller has no such feature.

Code modifications, similar to those described in this sub-section, are applied to other parts of the code such as Intra Prediction, which also requires pixels from neighboring blocks. The resulting code was profiled in the baseline core of the target platform under non perfect memory scenario and a speed-up of 47.9× with respect to the initial OpenAVS code was obtained, as shown in Table I.

We also ran the same code version in the baseline processor of the target platform with a more realistic memory configuration consisting of a 4KB ICache (2-way set associative, 64 byte line) and an 8KB write-back DCache (2-way set associative, 64 byte line). The Read/Write latency from/to main memory was 32 cycles for the first cycle of a burst, and one cycle after that. For this case, the performance fell to around 1.95 GHz.

B. Block Level Parallelism

The next step is to exploit instruction and data level parallelism at the basic block level. The target core can schedule up to two instructions per cycle in the form of a 64-bit VLIW instruction. These instructions are automatically generated by the compiler without user intervention and are freely intermixed with the rest 32/24/16 bit instructions.

The next two sub-sections describe how we exploit data level parallelism in two computationally demanding kernels of the AVS decoder: motion compensation and deblocking filter.

1) Motion Compensation

Motion Compensation is the process of compensating for the movement of rectangular blocks of pixels between frames. In contrast to H.264 which supports blocks with size as small as 4x4, AVS supports only four block sizes, i.e. 16x16, 8x16, 16x8 and 8x8, since smaller blocks are rarely used in high resolution video coding. The precision of motion vectors is quarter pixel for luma components and 1/8 pixel for chroma. As luma and chroma samples at sub-sample positions do not exist, it is necessary to generate them from nearby coded samples. Most of the complexity of the MC module, approximately 40% of the total execution time, is due to the quarter pixel interpolation. Since motion compensation is the most computationally expensive tool of AVS, its TIE acceleration, and especially the implementation of the filters used in vertical and horizontal interpolation, is critical to achieve real-time, high resolution video decoding.

In AVS, the predictive value at half sample position can be obtained with horizontal or vertical interpolation using the four- tapping filter F1 (-1, 5, 5, -1) and the predictive value at quarter sample position can be obtained with interpolation using the four-tapping filter F2 (1, 7, 7, 1). For example, the interpolation of half sample *b* in Fig. 7 is given by: $b' = -C + 5D + 5E - F$ and $b = clip((b'+4) >> 3)$. The interpolation at quarter pixels requires integer and half sample values. For example, the quarter pixel value *a* is given by: $a' = ee + 7D' + 7b' + E$ and $a = clip((a'+64) >> 7)$.

A key characteristic of interpolation is the significant amount of data reuse. Fig. 6 shows the positions of integer, half and quarter samples for luma components. In order to calculate sample *a*, we need the values from samples *D, E, ee* and *b*. Although samples *D* and *E* are integer pixels, *ee* and *b* are half-pixels and need to be re-calculated.

Taking advantage of the available reuse can significantly speed-up the entire process. One solution is to store all samples already computed and needed to interpolate other samples. However, this approach requires a significant memory footprint, which makes it prohibitive. Therefore, we aimed at using the data provided each time for computing multiple samples. In doing so we implemented a software "pipeline" such that no reload of the same pixel data is done, although we may still compute half pixel values more than once. In this way, the number of loads is dramatically reduced and a significant speed-up is achieved. Following is an example that demonstrates the software "pipeline" implementation of calculating half-pixel interpolation.

Fig. 8 illustrates the computation of the vertical 4-tap filter: dp' = ffp + 7*D*8 + 7*hp + H*8, dp = clip((dp'+64)>>7), where dp indicates quarter-pixel 'd', ffp, half-pixel 'ff', hp half-pixel 'h', D integer pixel 'D' and H integer pixel 'H' (Fig. 6).

```
for (i=0; i<SizeY; i++)
  for (j=0; j<SizeX; j++)
    ffp = - MC[(i-2),j] + 5*MC[(i-1),j] + 5*MC[i,j] - MC[(i+1),j];
    hp = - MC[(i-1),j] + 5*MC[i,j] + 5*MC[(i+1),j] - MC[(i+2),j];
    D = MC[i,j];
    H = MC[(i+1),j];
    pPred[i,j] = Clip ((ffpie + 7*Dpie*8 + 7*hpie + Hpie*8 + 64) >> 7);
  endfor
endfor
```

Fig. 8. Pseudo-code for interpolation of quarter-pixel d.

OpenAvs calculates the values of quarter-pixel 'd' for a block with size SizeY×SizeX. MC is an array with the integer pixel values and pPred an array where the calculated (predicted) values are kept. By using SIMD TIEs the inner

loop is eliminated and the loads are reduced by a factor of SizeX. However, ten loads (four each of ffp and hp and two for D, H) are still needed in each iteration. Taking a closer look, one can see that data from position (i-1) are used in the 2nd factor for the computation of ffp and in the 1st factor of hp. In the same way data from position (i) are used in the 3rd factor of ffp, in the 2nd factor of hp and as D, and so on. With the software "pipeline" implementation data loaded from a position are used to calculate all the factors in which they take part and store each computed factor in a register. When data from position (i+1), for example, are loaded we calculate factor –MC[(i+1),j] for ffp, 5* MC[(i+1),j] for hp and H and keep the results in registers. In this way the number of loads needed per iteration is reduced, from ten, to only five.

A final note concerns data alignment. MC requires memory loads of multiple bytes from memory positions that are not vector aligned, which is an impediment to SIMD vectorization. Fortunately, target platform's PP supports a large number of unaligned load instructions that can be used for the implementation of motion compensation with the usage of TIEs. However, a prospective developer must tackle the problem if the chosen processor does not support unaligned loads.

The aforementioned SIMD optimizations provide a 4.8× speed-up to the interpolation kernel. The effect on the total execution time is a 1.8× speed-up compared to the version with Variable Length Decoding (VLD) optimizations (Table I).

2) *Deblocking Filter*

The deblocking filter is a low pass filter across block boundaries applied as a last step in the decoder just before storing the reconstructed block of pixels back in the main memory. It is used to smooth block edges to improve the appearance of the reconstructed frame in image areas with low spatial frequency. Filtering is applied in two steps; first along horizontal edges and then across vertical edges of each 8x8 block. Fig. 9 shows that only the top rows of the current 8x8 luma block B (col, row) and the bottom rows of the luma block B (col, row-1) are affected from the deblocking filter, depending on the value of the boundary strength parameter *Bs*. This is a parameter that depends on the difference in coding types, motion vectors or quantization parameters between the two blocks on the two sides of an edge, and estimates how much low-pass filtering needs to be performed. It can take the value 0 (no filtering), 1 (medium filtering) and 2 (heavy filtering). It is worth mentioning that all variables needed in order to determine the filter strength of a given edge are related to syntax elements and as such are available to the stream processor, SP, who is responsible for the boundary strength calculation, while the filtering operation itself depends on the actual pixel values and therefore is performed by the pixel processor, PP. Fig. 10 shows the flow of the algorithm to produce the output pixels for Bs==2 or 1.

The algorithm of Fig. 10 has to be invoked 8 times to produce a row of output pixels on an 8x8 block, or 16 times to produce a row of pixels on a 16x16 macroblock. For the latter,

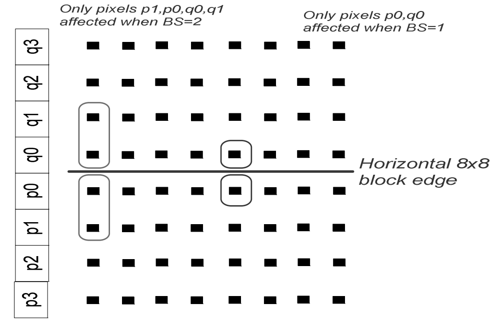

Fig. 9. Adjacent pixels for the horizontal deblock filter.

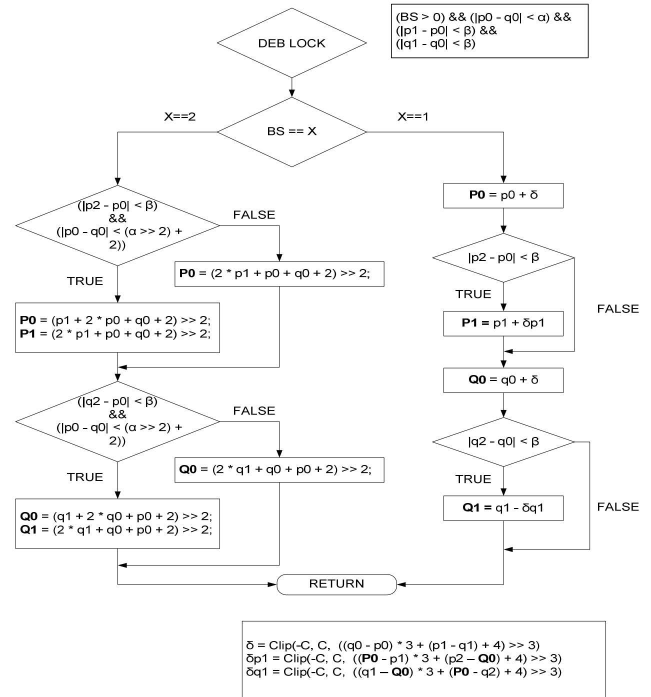

Fig. 10. The AVS Luma Deblocking Block Diagram.

the boundary strengths of the top two blocks of the macroblock must be equal.

A data parallel implementation of the deblocking filter uses the pixel processor TIEs to implicitly unroll the loop and vectorize the computations of Fig. 10. The pixels *p0, p1, p2, q0, q1, q2* of Fig. 10 become 8 or 16-pixel vectors P0, P1, P2, Q0, Q1, Q2. The vectorization has the potential to speed-up execution time of the inner loop by a factor of 8 or 16 provided that the vectors are 8 or 16 bytes aligned, respectively.

In the vectorized version of the code, the parts of the code with conditional execution semantics are predicated, so that an instruction has effect only if the predicate is true. We use bit vectors as predicates to merge the boolean results of the conditions to vectors, and we use these vectors as a predicate to commit or not a particular operation.

The same algorithm is used for the deblocking filter of the vertical edges. The main difference is that the SIMD operations we described are suitable for row processing, but not for the column processing required for the vertical edges.

This problem was solved by the use of the TIEQT unit of the target platform.

The extra overhead to perform the transpose limits the performance gains of the vertical filters. For example, the speed-up for the horizontal deblocking filter of Fig. 9 is 6.5× after SIMD acceleration, whereas the speed-up for the vertical filter is only 2.5×. The average speed-up of all the deblocking filter kernels comes down to 3.35×.

The collective effect of SIMD parallelization improves total execution time by an additional 2.24×, for a total speed-up of 107× compared to the initial OpenAVS code (Table I).

C. Task Level Parallelism

A heterogeneous dual core processor allows simultaneous execution of different parts of the AVS decoder for a single or even for multiple macroblocks. There are two major steps to port the AVS decoder to a multi-core system. First, the code and related data structures should be partitioned and assigned to the appropriate core. Second, a communication mechanism must be set up to transfer data between the two cores.

In the first and most crucial step emphasis is given to improving the load balance between the two cores. In the target platform, PP executes disproportionately larger workload, compared to SP, due to the already existing task partitioning (Fig. 4).

The function-flow in the macroblock loop of the optimized single core coder is:

1. *ParseOneMacroBlock* (SP) :
   - Bitstream parsing
2. *MciDCTReconOneMacroBlock* (PP) :
   - *InitOneMacroblock*
   - *IntraPredLuma& IntraPredChroma/InterPredP_exec/InterPredB_*
   - *iDCT+Recon/Copy*
3. *DeblockOneMacroblock* (PP)
4. *CopyMBPictureData (DMA transfers, memory stores)*

Given the set of TIEs available in the two cores, *ParseOneMacroBlock* is the only function that can be mapped to SP, while *MciDCTRecOneMacroBlock* and *DeblockOneMacroBlock* map to PP. Finally, *CopyMBPictureData* is a function that can be directly substituted by a set of DMA calls and internal SRAM copies, since it performs the last part of decoding, which is to copy the decoded macroblock data from the internal SRAM of the pixel processor back to the external frame buffers. That gives us a stream processor to pixel processor load ratio of about 20% : 80%, while the identical is 50% : 50%.

To improve load balance between the two cores we had to thoroughly investigate the code and track the parts of each function that, when appropriately modified, could be executed by the stream processor. We briefly present the process for the *MciDCTRecOneMacroBlock* function, which was the most time-consuming.

MciDCTRecOneMacroBlock first calls *InitOneMacroBlock* which initializes the structure which keeps the needed info for the current macroblock. Afterwards, for P- and B-inter predicted macroblocks, it processes the entire Luma 16x16 macroblock first (in *InterPredLumaP* or *InterPredLumaB*), followed by a loop over the four 8x8 blocks for inverse transform and reconstruction. Then, it processes both Chroma 8x8 blocks (in *InterPredChromaP* or *InterPredChromaB*), followed by the two 8x8 chroma blocks processing for inverse transform and reconstruction. The structure for the *InterPredLumaP/InterPredLumaB* and *InterPredChromaP/ InterPredChromaB* functions is very similar. First, there is a large switch statement that checks the macroblock type and then a call to the appropriate, according to the macroblock type, function which checks for the block location and then determines the motion vector(s) to be used for motion compensation, based on the block type and availability of its neighbors. Finally, the core motion-compensation function, *GetBlock* is called, which contains all the four-tap luma filtering to achieve quarter-pixel motion interpolation.

Most of the functions invoked by *MciDCTRecOneMacroBlock* are block based functions, which means that the preparation of all appropriate data and information (such as motion vectors), along with the interpolation are executed for each block separately. By being so, the load-from-external memory and filtering operations for a macroblock are interleaved. This produces a major bottleneck in the load balancing between the two cores, as they are macroblock based, i.e. SP prepares and PP 'consumes' data for an entire macroblock. Therefore, the code must be restructured so as the data and information collection (which can be executed by the stream processor) is done per macroblock and not per block.

From the above, only the initialization by *InitOneMacroBlock* can obviously be executed by the stream processor. In order to move more tasks to SP we performed the following changes:

i. The motion compensation code for luma and chroma was merged. This results in the sequential processing of Luma (Y), Chroma-U and Chroma-V in the same function.
ii. The internal motion compensation memory (*_mc_memory*) was increased from the previously single-block worst case to the full-macroblock worst case.
iii. The *GetBlock* and *GetChromaBlock* functions were first merged into one (*GetBlock*) that performs motion interpolation for all 3 components (Y/U/V) and then split into two functions *GetBlock_load* and *GetBlock_exec. GetBlock_load* is a simple wrapper for the core data-movement function, that copies data from the appropriate source addresses in the external reference frame Y/U/V buffers to the appropriate (sequential) destination addresses in the internal *_mc_memory* memory. *GetBlock_exec* retrieve the appropriate MC Y/U/V pointers and performs the actual luma and chroma filtering.

Through the above modifications, nearly half of the initial *MciDCTRecOneMacroBlock* can be executed by SP.

Following the same approach the deblocking filter functionality was partitioned into the functions *DeblockOneMacroBlock_sp* and *DeblockOneMacroBlock_pp*. The resulted function-flow in the macroblock-loop is:

1. *ParseOneMacroBlock* (SP) :
   - Bitstream parsing
   - *InitOneMacroblock*
   - *InterPredP_load/InterPredB_load* (for P and B-coded macroblocks): These 2 functions eventually fill up the *_mc_memory* internal memory with the appropriate data and prepare all information (mainly motion vectors) that is needed later.
2. *DeblockOneMacroblock_sp* (SP)
3. *MciDCTReconOneMacroBlock* (PP) :
   - *IntraPredLuma& IntraPredChroma/InterPredP_exec/InterPredB_exec* (for I/P/B macroblocks): These result in the MC Y/U/V pointers retrieval and the actual filtering of data, storing them to the Pred array.
   - *iDCT+Recon/Copy*
4. *DeblockOneMacroblock_pp* (PP)

After repartitioning of the code as described, the stream processor to pixel processor load ratio was improved from 20%-80% to 45%-55%.

The final porting step was to set up a communication mechanism to transfer data between the two cores. The target platform uses a DMA engine to interleave data transfer with computation and increase system performance. The non-blocking functionality of the DMA requires that SP and PP synchronize their execution at specific points. The DMA unit decouples the execution of the two cores allowing for non-blocking transfers. The running number of concluded data transfers can also be used as a synchronization mechanism by having the producing core transmitting data up to a specific number, and the receiving core waiting for a predefined number of transfers.

To increase the degree of decoupling, multiple buffering is used to allow the two cores to work on different macroblock data. Our current implementation uses a two MB overlap between the two cores, which means that SP is processing MB_{n+2}, whereas PP is still at MB_n. Deeper buffering schemes require a substantial increase of internal SRAM requirements.

Moreover, the presented code restructuring guarantees that data flows only from SP to PP, from main memory to PP, and from PP back to main memory. In other words, there is no transfer from the pixel processor to the stream processor that would create a cyclic dependency and would reduce the performance potential of the code. Fig. 11 shows a graphical representation of the data transfers for the initialization and for the first couple macroblocks.

The dual core mapping resulted in an additional performance improvement of 1.8×, out of the ideal 2×, due to overhead associated with the DMA set-up, and some small residual load imbalance.

D. Summary

The previous sub-sections showed how we ported the un-optimized OpenAVS code to the heterogeneous dual core target platform. Our approach was to optimize the code for the single core model and then move to the final optimization for the dual core model. Although we worked with specific video decoder and processor, many of the optimizations presented can be generalized. A summary of the most important of them follows.

- Optimization for single core model: after profiling the initial code, we tracked the most expensive functions and focused the optimization strategy on them. Optimizations target the improvement of memory access and the reduction of complexity. This is accomplished by:

  i. Identifying frequently accessed data structures and copying them to internal memory.
  ii. Restructuring the code to optimize data reuse in the internal memory.
  iii. Restructuring the code to convert computations from frame-based to block-based.
  iv. Reducing check points, i.e. branches, in the code.
  v. Reducing data loads by implementing software pipelines.

- Optimization for dual core model: the load between the two cores should be as much balanced as possible (ideally 50% - 50%). This is accomplished by:

  i. Identifying functions in tasks, operated by the most 'burdened' core, that can be executed by the less 'burdened' core.
  ii. Restructuring the remaining functions so as to create new ones that can be executed by the less 'burdened' core.
  iii. Restructuring the code so as to guarantee that data flows only one way, thus preventing cyclic dependencies between the two cores.

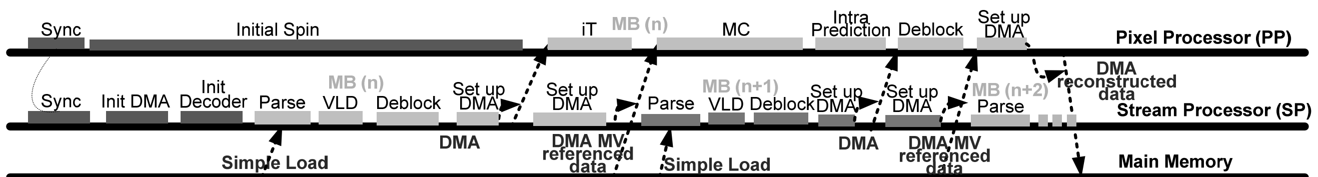

Fig. 11. Task scheduling of the AVS decoder in the target dual core engine using DMA transfers.

TABLE I
PERFORMANCE RESULTS ON TARGET PLATFORM USING MEMORY SUBSYSTEM WITH DEFAULT WRITE READ LATENCIES. EQUIVALENT F_{CLK} IS THE CORE CLOCK FREQUENCY TO DECODE PAL-D1 VIDEO (720×576×25 FPS).

Optimization		Equivalent F_{clk}	Speed-up Factor
0. Baseline OpenAVS code		70073 MHz	1
1. Software Optimizations and SRAM placement		1463 MHz	47.89
2. TIE optimization	1. Parsing and VLD	1365 MHz	51.34
	2. (1) plus MC, Intra Prediction, Inverse Transform	761 MHz	92.08
	3. (2) plus Deblocking	654 MHz	107.1
3. Dual Core		359 MHz	195.2

TABLE II
PAL-D1 CORE CLOCK FREQUENCIES IN MHZ ON TARGET PLATFORM USING: A) (0-READ, 0-WRITE), B) DEFAULT (8-READ, 4-WRITE), C) (64-READ, 32-WRITE) WAIT LATENCIES.

Optimization		a)	b)	c)
0. Baseline OpenAVS code		7805	70073	700704
1. Software Optimizations and SRAM placement		1216	1463	1533
2. TIE optimization	1. Parsing and VLD	1164	1365	1417
	2. (1) plus MC, Intra Prediction, Inverse Transform	632	761	814
	3. (2) plus Deblocking	544	654	706

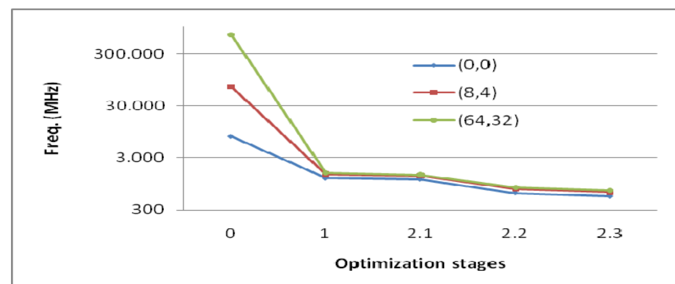

Fig. 12. PAL-D1 core clock frequencies on target platform using: a) (0-read, 0-write), b) default (8-read, 4-write), c) (64-read, 32-write) wait latencies for TABLE II optimization stages.

IV. CONCLUSION

This paper outlined the steps taken to optimize and extract block and task level parallelism from a video decoding application for a heterogeneous dual core processor with SIMD instructions. The total speed-up of more than 195× compared to a software x86-based implementation, enables real-time, 25fps decoding of D1 video.

Through the process of porting AVS, the Chinese video coding standard, to the target dual core processor, we highlighted a number of optimization techniques that can be generalized to any SIMD processor. Moreover, we showed how to detect and extract parallelism at all levels of granularity and how their exploitation can lead to significant performance improvement.

Given the similarities of AVS to H.264 (AVC), and the availability of custom-TIEs that handle bitstream parsing, transform and deblocking according to H.264 standard, the same steps presented here can be used to accelerate H.264 decoding on the same hardware platform, with similar results. In the future, we plan to test our approach on alternative platforms and compare its efficiency with symmetric multi-core processors that allow for data-decomposition multi-threading.

REFERENCES

[1] R. B. Lee, "Multimedia extensions for general-purpose processors," *Proc. IEEE Workshop on Signal Processing Systems*, pp. 9-23, Nov. 1997.

[2] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, vol. 20, no. 2, pp. 85-95, Mar/Apr 2000.

[3] S. H. Jo, S. Jo, Y. H. Song, "Efficient Coordination of Parallel Threads of H.264/AVC Decoder for Performance Improvement", *IEEE Transactions on Consumer Electronics,* Vol. 56, No. 3, pp 1963-1971, August 2010.

[4] J. Huang, J. Lee, "Efficient VLSI Architecture for Video Transcoding", *IEEE Transactions on Consumer Electronics*, Vol. 55, No. 3, pp1462-1470, August 2009.

[5] J. Lau, "MPEG-4, AVS deliver better video compression more flexible format," *Electronic Times Asia,* June 1st, 2006.

[6] ISO/IEC IS 13818, General Coding of Moving Picture and Associated Audio Information, 1994.

[7] Information technology – Coding of audio-visual objects – Part 2: Visual (ISO/IEC FCD 14496), July 2001.

[8] Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC), May 2003.

[9] L. Fan, S. Ma, F. Wu, "Overview of AVS Video Standard," *2004 IEEE International Conference on Multimedia and Expo (ICME '04).*

[10] "VDO Instruction Set Architecture (ISA) Extensions Reference Manual," July 2007.

[11] J.L. Hennessy, D.A. Patterson, "*Computer Architecture. A Quantitative Approach*," Morgan Kaufmann, 4th Edition, 2006.

BIOGRAPHIES

Maria G. Koziri (M'08) received the Dip. Eng. degree in computer engineering from Technical University of Crete, Greece, in 2003 and the Ph.D. degree from University of Thessaly, Greece, in 2007.

Her research interests include video compression, scalable video coding, rate–distortion optimization, as well as computer architecture.

Dimitrios T. Zacharis is currently a Ph.D. candidate at the University of Thessaly, Greece, where he received the Dip. Eng. and the M.Sc. degrees in Computer and Telecommunications Engineering, in 2006 and 2009 respectively.

Ioannis Katsavounidis (S'95-M'98) received the Ph.D. degree in Electrical Engineering from the University of Southern California, Los Angeles, in 1998.

He was with InterVideo Inc. in Fremont, CA, as Director of Software from 2000-2007, where he worked on a number of video codec problems, including error resilience, encoder rate-distortion optimization and multi-core HD-resolution decoder software optimizations. He was one of the co-founders and CTO of Cidana Corp. working on multimedia and DTV applications on embedded devices from 2007-08. He joined the Computer and Communications Department, University of Thessaly, Volos, Greece in 2008, where he is currently Associate Professor.

Nikos Bellas is currently an Associate Professor at the Electrical and Computer Engineering Department at the University of Thessaly, Greece. He received his M.Sc. and Ph.D. degrees from the ECE Department of the University of Illinois at Urbana-Champaign in 1995 and 1998, respectively. From 1999 to 2007 he was a principal member of technical staff at the Embedded Imaging Systems Lab of Motorola Labs, Chicago, IL working on chip design for multimedia processors, and CAD tools for architectural synthesis.