# Mapping the AVS Video Decoder on a Heterogeneous Dual-Core SIMD Processor

**Nikolaos Bellas, Ioannis Katsavounidis, Maria Koziri, Dimitris Zacharis**
**University of Thessaly**
**Volos, Greece**
**nbellas@uth.gr**

## 1. 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.

This paper describes the porting of the AVS (Audio Video Standard) [1] video decoder to Tensilica's Diamond 388VDO dual core [2], a heterogeneous dual core processor which consists of two cores with video specific instruction extensions. The AVS was drafted by the A/V work group of China to replace older and royalty-burdened standards such as MPEG-2 and H.264 mainly in consumer applications. We start from an open source implementation of the AVS decoder, OpenAVS [3], which targets a general processor platform (GPP), and we gradually transform the code to enable a dual core implementation on the Diamond 388VDO core. Our aim is to achieve real time, 25 fps, progressive D1 resolution (720x576) AVS video decoding.

The Diamond family of processors includes preconfigured versions of the 32-bit Xtensa configurable architecture optimized for video encoding and decoding. Diamond uses the capability of the Xtensa architecture to configure a processor core by adding extra application-specific instructions. 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. 720x576x25 (PAL) or 720x480x30 (NTSC) pixels/sec. Our contribution is to use the Diamond toolset (IDE, cross-compiler, functional and cycle-accurate simulators, debugger) to port a new video standard to the architecture based on ISA extensions tailored for older video standards.

As shown in Fig. 1, Diamond 388VDO is a heterogeneous dual core processor which consists of two Xtensa cores with video specific instruction extensions. The two Xtensa cores in 388VDO are referred to as Stream processor and Pixel processor. The main task of the Stream processor is to parse and decode the video bitstream. The Pixel processor performs most of the heavy duty computations of video decoding using SIMD instructions (called Tensilica Extension Instructions, TIEs). It is used to accelerate motion compensation (including quarter pixel interpolation and reconstruction), intra prediction, inverse quantization (optionally), inverse transform and the deblocking filter. Both these processors have tightly coupled instruction and data SRAM memories that are used to reduce memory access latency and increase bandwidth.

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.

The challenge of mapping a new video decoder in a multi-core engine like the 388VDO is to detect and extract parallelism at all levels of granularity, especially at the higher levels.

## 2. AVS Video Standard Optimization

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) at only 12.9%. According to the profiling, the baseline processor will have to be clocked at 1.73 GHz to meet a 25 frames/sec performance requirement under the perfect memory scenario.

Our optimization strategy focuses on the most expensive functions of the AVS decoder such as motion compensation and deblocking filter. We describe a sequence of three optimizations steps that improve the performance of the video decoder from 1.73 GHz down to 359 MHz (shown in Table I).

### Software Optimizations

The first step is to re-write parts of the code to improve the memory locality of the data accesses and minimize the need to access data from the main memory.

For example, the following OpenAVS code includes two main loops to process a frame as shown below:

```
for (MbIndex = 0; MbIndex < no_MBs; < MbIndex++) {
  ParseOneMacroblock;  // Parsing, VLD
  McIdctRecOneMacroblock; // Inverse transform, MC
}
for (MbIndex = 0; MbIndex < no_MBs; < MbIndex++) {
  DeblockOneMacroblock // Deblocking
}
```

The problem with this reference OpenAVS code is that using two separate loops causes the whole frame to spill to the main memory: the output frame of the MC has to be stored to the main memory, and then retrieved back by the Deblock 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(c,r) after MC. These 720x3x2 + 16x3x2 = 4416 bytes are the only pixel data needed for the Deblocking filter of MB(c,r). 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 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.

### SIMD level parallelism

#### Motion Compensation

Motion Compensation is the process of compensating for the movement of rectangular blocks of pixels between frames. 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.

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). The interpolation at quarter pixels requires integer and half sample values (Fig. 3). For example, the quarter pixel value *a* is given by: $a' = ee + 7D' + 7b' + E'$ and $a = clip((a' + 64) >> 7)$.

One of the main challenges to SIMD vectorization is that motion compensation may require memory loads of multiple bytes from memory positions which are not vector aligned. The Diamond 388VDO pixel processor supports a large number of unaligned load instructions that can be used for the implementation of motion compensation with the usage of TIEs. For example, the unaligned load instruction *xvd_lda_16x8* returns 16 bytes, i.e. an entire macroblock row, and by using the appropriate SIMD TIE instructions, one can calculate the vertical filter for any block size.

These SIMD optimizations provide a 4.8x speed up to the interpolation kernel, the function that iterates in an 8x8 pixel block to compute the interpolated pixels. The effect to the total execution time is 1.8x speed up compared to the version with Variable Length Decoding (VLD) optimizations (Table I).

### 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. 4 shows that only the top rows of the current 8x8 luma block B(c, r) and the bottom rows of the luma block B(c, r-1) are affected from the deblocking filter, depending on the value of the boundary strength parameter *Bs*. It can take the value 0 (no filtering), 1 (medium filtering) and 2 (heavy filtering).

A data parallel implementation of the Deblocking filter uses the pixel processor TIEs to implicitly unroll the loop and vectorize the computations of Fig. 5. The pixels *p0, p1, p2, q0, q1, q2* of Fig. 4 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. The average speed up of all the Deblocking filter kernels is 3.35. The collective effect of SIMD parallelization improves total execution time by an additional 2.26x, for a total speed up of 2.67 compared to the initial OpenAVS code. As before, we assume a perfect memory system with zero-wait cycles.

### 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 the related data structures should be partitioned and allocated to corresponding memory spaces. Second, a communication mechanism must be set up to transfer data between the two cores.

Functions are dedicated to one core only, and not split across different cores. Each executable is placed in the local SRAM of the corresponding processor. It is important to achieve a balanced load partitioning between the two cores in order to achieve the theoretical maximum speed up of 2. In our case, the stream processor to pixel processor load ratio was 45%-55%.

The multi-channel DMA is used in three different cases:

- to transfer the luma and chroma coefficient blocks together with other control parameters for each macroblock from the local SRAM of the stream processor to the local SRAM of the pixel processor,
- to transfer pixel blocks of previous frames from main memory to the local SRAM of the pixel processor. These blocks are used for interpolation and motion compensation in the pixel processor.
- To transfer the final pixel blocks from the local SRAM of the pixel processor back to the main memory – both in order to be displayed and in order to be used for motion compensation during decoding of subsequent frames.

The DMA engine interleaves data transfer and computation to increase system performance. The non-blocking functionality of the DMA requires that the stream and pixel processors synchronize their execution at specific points. The DMA unit decouples the execution of the two cores which can schedule their data transfer and data receipt at their own pace, without executing at lock step to each other.

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

The dual core mapping resulted in an additional performance improvement of 1.8x, out of the ideal 2x, due to the overhead associated with the DMA set up, and the load imbalance. The total speed up of almost 5x compared to a software x86-based implementation, enables real-time, 25fps decoding of D1 frames.

## 3. References

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

[2] “Diamond Standard Core Processor Architecture,” *Tensilica White Paper,* July 2007.

[3] *http://sourceforge.net/projects/openavs*

[4] “VDO Instruction Set Architecture (ISA) Extensions Reference Manual,” July 2007

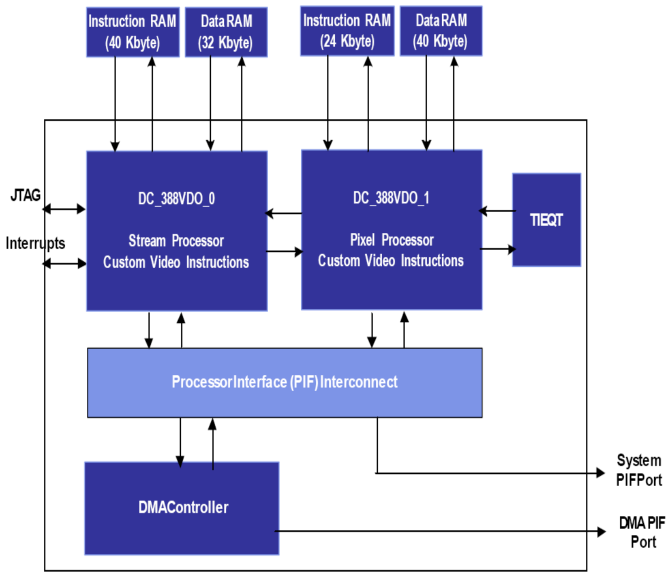

Instruction RAM (40 Kbyte)
Data RAM (32 Kbyte)
Instruction RAM (24 Kbyte)
Data RAM (40 Kbyte)
JTAG
Interrupts
DC_388VDO_0
Stream Processor Custom Video Instructions
DC_388VDO_1
Pixel Processor Custom Video Instructions
TIEQT
Processor Interface (PIF) Interconnect
DMA Controller
System PIF Port
DMA PIF Port

Figure 1: Diamond 388VDOVideo Engine Block Diagram

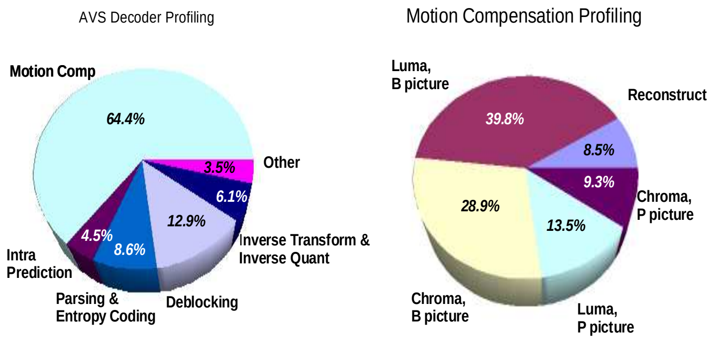


Figure 2: Execution profiling of the software OpenAVS decoder (a) Motion Compensation amounts to 64.4% of the total execution time (b) Details on the Motion Compensation profiling

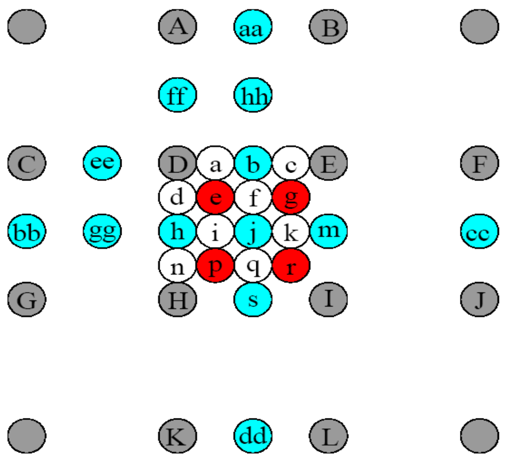


Figure 3: Interpolation of Luma components

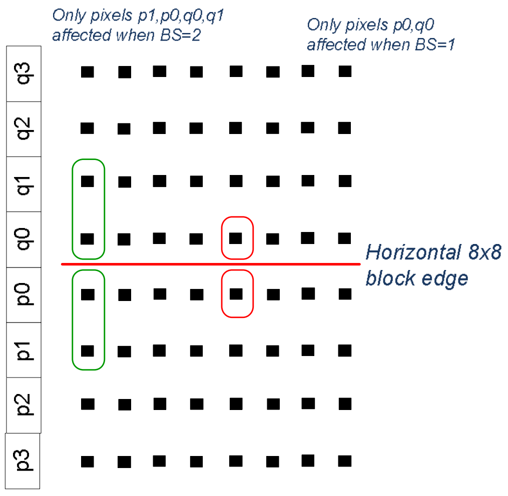
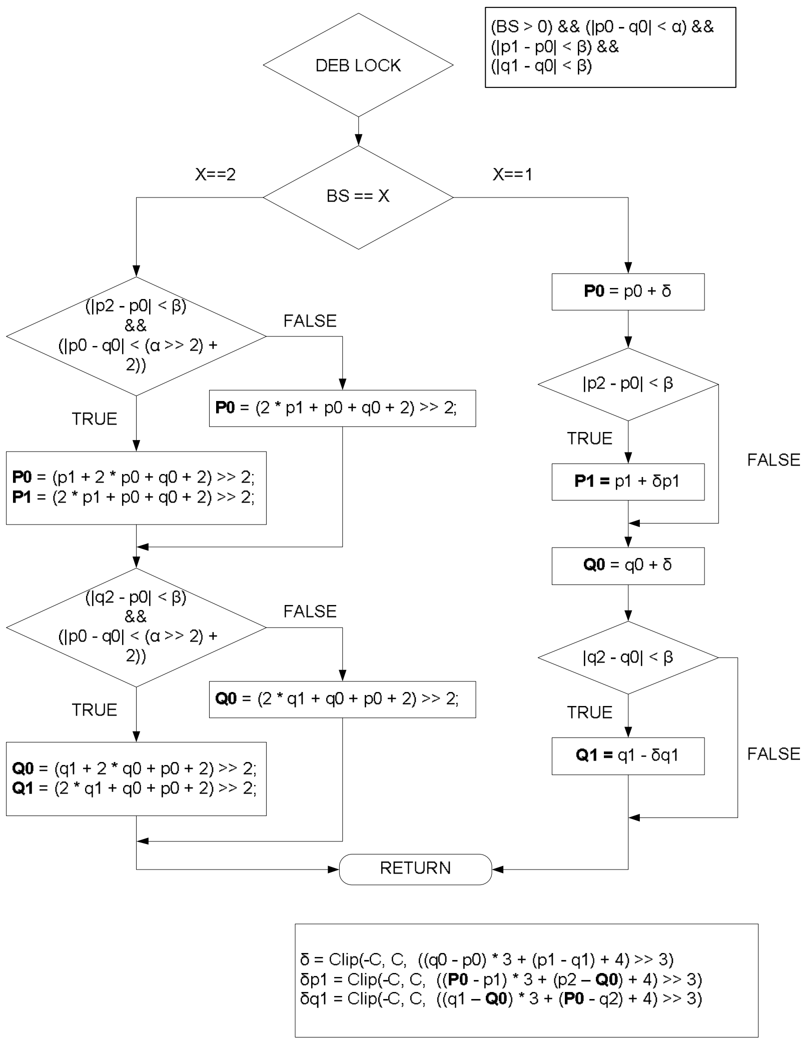

Only pixels p1,p0,q0,q1 affected when BS=2
Only pixels p0,q0 affected when BS=1
q3
q2
q1
q0
p0
p1
p2
p3
Horizontal 8x8 block edge

Figure 4: Adjacent pixels for the horizontal deblock filter



DEB LOCK
(BS > 0) && (|p0 - q0| < α) && (|p1 - p0| < β) && (|q1 - q0| < β)
BS == X
X==2
X==1
(|p2 - p0| < β) && (|p0 - q0| < (α >> 2) + 2))
FALSE
TRUE
P0 = (2 * p1 + p0 + q0 + 2) >> 2;
P0 = (p1 + 2 * p0 + q0 + 2) >> 2;
P1 = (2 * p1 + p0 + q0 + 2) >> 2;
(|q2 - p0| < β) && (|p0 - q0| < (α >> 2) + 2))
FALSE
TRUE
Q0 = (2 * q1 + q0 + p0 + 2) >> 2;
Q0 = (q1 + 2 * q0 + p0 + 2) >> 2;
Q1 = (2 * q1 + q0 + p0 + 2) >> 2;
P0 = p0 + δ
|p2 - p0| < β
TRUE
FALSE
P1 = p1 + δp1
Q0 = q0 + δ
|q2 - q0| < β
TRUE
FALSE
Q1 = q1 - δq1
RETURN
δ = Clip(-C, C, ((q0 - p0) * 3 + (p1 - q1) + 4) >> 3)
δp1 = Clip(-C, C, ((P0 - p1) * 3 + (p2 – Q0) + 4) >> 3)
δq1 = Clip(-C, C, ((q1 – Q0) * 3 + (P0 - q2) + 4) >> 3)

Figure 5: The AVS Luma Deblocking Block Diagram.

Table I: Summary performance results for the Diamond 388VDO acceleration. The Equivalent $F_{clk}$ shows the clock frequency of the Diamond core to decode 25fps at 720x576 D1 resolution. All the rows except for the last one refer to a single core engine.

| **Optimization** | **Equivalent $F_{clk}$ (MHz)** | **Speedup factor** |
|---|---|---|
| **Baseline OpenAVS code** | 1730 | 1 |
| **Software Optimizations** | 1461 | 1.18 |
| **SIMD parallelization (TIEs)** | | |
| **1. Parsing and VLD only** | 1354 | 1.28 |
| **2. (1) plus MC, Intra Prediction, inverse Transform** | 748 | 2.31 |
| **3. (2) plus Deblocking** | 649 | 2.67 |
| **Dual Core** | 359 | 4.8 |