Decoder Hardware Architecture for HEVC

Decoder Hardware Architecture for HEVC

Mehul Tikekar, Chao-Tsung Huang, Chiraag Juvekar, Vivienne Sze, and Anantha Chandrakasan

Abstract This chapter provides an overview of the design challenges faced in the implementation of hardware HEVC decoders. These challenges can be attributed to the larger and diverse coding block sizes and transform sizes, the larger interpolation filter for motion compensation, the increased number of steps in intra prediction and the introduction of a new in-loop filter. Several solutions to address these implementation challenges are discussed. As a reference, results for an HEVC decoder test chip are also presented.

Acknowledgements The authors gratefully acknowledge the support of Texas Instruments for sponsoring the HEVC decoder test chip project and Taiwan Semiconductor Manufacturing Company (TSMC) University Shuttle program for manufacturing the chip.

1 Introduction

HEVC presents several new challenges for a hardware decoder implementation. HEVC's decoding complexity is found to be between $1.4\times - 2\times$ of H.264/AVC [1] when measured in terms of cycle count for software. In hardware, however, the increased complexity of HEVC entails significant increase in hardware cost over traditional H.264/AVC decoders, both at the top-level of the video decoder, and in the low-level processing blocks. Some of the challenges are listed below.

- The diverse sizes of Coding Tree Units (CTU), Coding Units (CU), Prediction Units (PU) and Transform Units (TU) require complex state machines to control the system pipeline and data paths in the individual processing blocks.

Mehul Tikekar · Chiraag Juvekar · Vivienne Sze · Anantha Chandrakasan
Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA

Chao-Tsung Huang
National Tsing Hua University, Taiwan

- The largest CTU (64×64) is 16× larger than the H.264/AVC macroblock (16×16), which means that the memories in pipeline stages need to be proportionately larger.
- The inverse transform block is considerably more complicated due to the large TU sizes and higher precision of the transform matrix. The largest TU size (32×32) requires a 16× larger transpose memory.
- HEVC uses an 8-tap luma interpolation filter for motion compensation as compared to the 6-tap filter in H.264/AVC. This increases the bandwidth required from the decoded picture buffer.

The architecture of the video decoder depends strongly on parameters such as the required throughput (i.e. pixel rate defined by the level limit in the HEVC specification), technology node, area and power budgets, control and data interface to the external world and memory technology used for the decoded picture buffer. In this chapter, we describe the architecture for an HEVC decoder for 4K Ultra HD decoding at 30 fps designed in 40 nm CMOS technology with external DDR3 memory for the decoded picture buffer. The decoder operates at 200 MHz and is frequency-scalable for lower resolutions and picture rates. Along with techniques used in H.264/AVC decoders, such as frame-level parallelism [2] and reference frame compression [3], and general VLSI techniques such as pipelining and dynamic voltage and frequency scaling, HEVC decoders can benefit from architectural techniques like:

- Variable-size pipelining to reduce on-chip SRAM and handle different CTU sizes.
- Unified processing engines for prediction and transform to manage the large diversity of PU and TU sizes.
- High-throughput motion compensation (MC) cache to address increased DRAM requirements for the longer interpolation filters.

2 System Pipeline

The granularity of the top-level pipeline is affected by processing dependencies between pixels. For example, computing the luma residue at any pixel location requires all transform coefficients in the TU that contains the pixel. Hence, it is not possible for the inverse transform block to use, say, a 4×4 pixel pipeline; the pipeline granularity must be at least one TU in size. In general, it is desirable to minimize the pipeline granularity to reduce processing latency and memory sizes.

The largest CTU needs 6 kB to store its luma and chroma pixels with 8-bit precision. The transform coefficients and residue are computed with higher precision (16-bit and 9-bit, respectively) and require larger storage accordingly. Other information such as intra-prediction mode, inter-prediction motion vectors, etc. needs to be stored at a 4×4 granularity. All of these require large pipeline buffers in SRAM and several techniques can be used to reduce their size as described in this chapter.

Line buffers are required to handle data dependencies between CTUs in the vertical direction. For example, the deblocking filter needs to store 4 rows of luma pixels and 2 rows of chroma pixels (per chroma component) due to the deblocking filter's support. The size of these buffers is proportional to the width of the picture. Further, if the picture is split into multiple tile rows, each tile row needs a separate line buffer if the rows are to be processed in parallel. Tiles also need column buffers to handle data dependencies between them in the horizontal direction. Traditionally, line buffers have been implemented using on-chip SRAM. However, for very large picture sizes, it may be necessary to store them in the denser off-chip DRAM. This results in an area and power trade-off as communicating to the off-chip DRAM takes much more power.

Also, off-chip DRAM is used most commonly to store the decoded picture buffer. The variable latency to the off-chip DRAM must be considered in the system pipeline. In particular, buffers are needed between processing blocks that talk to the DRAM to accommodate the variable latency. Motion compensation makes the most number of accesses to the external DRAM and a motion compensation cache is typically used to reduce the number of accesses. With a cache, the best-case latency for a memory access is determined by a cache hit and it can be as low as one cycle. However, the worse-case latency, determined by a cache miss, remains more or less unchanged thus increasing the overall variability seen by the prediction block.

To summarize, the top-level system pipeline is affected by:

1. Processing dependencies
2. Large CTU sizes
3. Large line buffers
4. Off-chip DRAM latency

2.1 Variable-sized Pipeline Blocks

Compared to the all-intra or all-inter macroblocks in H.264/AVC, the Coding Tree Units (CTU) in HEVC may contain a mix of inter and intra-coded Coding Units. Hence, it is convenient to design the pipeline granularity to be equal to the CTU size. If the pipeline buffers are implemented as multi-bank SRAM, the decoder can be made power-scalable for smaller CTU sizes by shutting down the unused banks. However, it is also possible to use the unused banks and increase the pipeline granularity beyond the CTU size. For example, a CTU-adaptive pipeline granularity shown in Table 2.1 is employed by [4].

The Variable-sized Pipeline Block (VPB) is as tall as the CTU but its width is fixed to 64 for a unified control flow. Also, by making the VPB larger than the CTU (for CTU 32×32 and 16×16), motion compensation can predict a larger block of luma pixels before predicting the chroma pixels. This reduces the number of switches between luma and chroma memory accesses which, as explained later in Section 6, can have benefits on the DRAM latency.

Table 1 CTU-adaptive pipeline granularity

Coding Tree Unit (CTU)	Variable-sized Pipeline Block (VPB)
64×64	64×64
32×32	64×32
16×16	64×16

2.2 Split System Pipeline

To deal with the variable latency of the cache+DRAM memory system, elastic pipelining can be used between the entropy decoder, which sends read requests to the cache, and prediction, which reads data from the cache. As a result, the system pipeline can be broken into two groups. The first group contains the entropy decoder while the second contains inverse transform, prediction and the subsequent in-loop filters. This scheme is shown in Fig. 1.

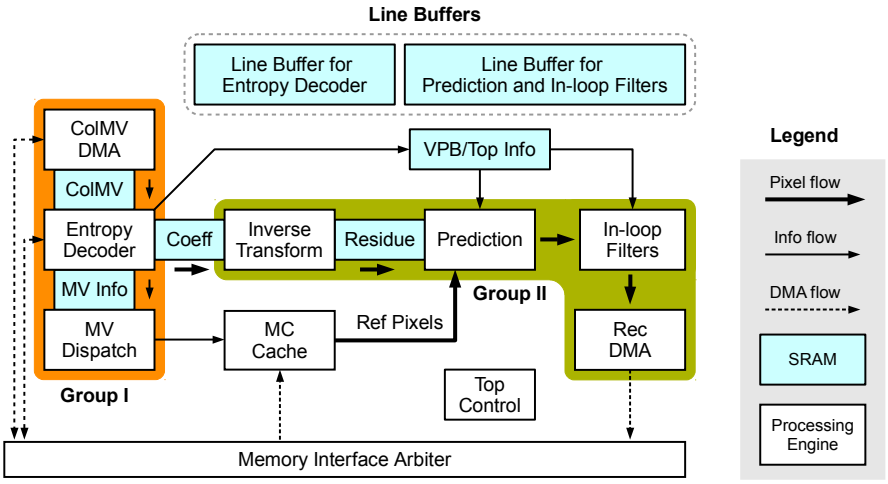


Fig. 1 System pipelining for HEVC decoder. Coeff buffer saves 20 kB of SRAM by TU pipelining. Connections to Line Buffers are omitted in the figure for clarity (see Fig. 3 for details).

Entropy decoder uses collocated motion vectors from decoded pictures for motion vector prediction. A separate pipeline stage, ColMV DMA is added prior to entropy decoder to read collocated motion vectors from the DRAM. This isolates entropy decoder from the variable DRAM latency. Similarly, an extra stage, reconstruction DMA, is added after the in-loop filters in the second pipeline group to write back fully reconstructed pixels to DRAM. Processing engines are pipelined with VPB granularity *within* each group as shown in Fig. 2. Pipelining *across* the groups is explained next.

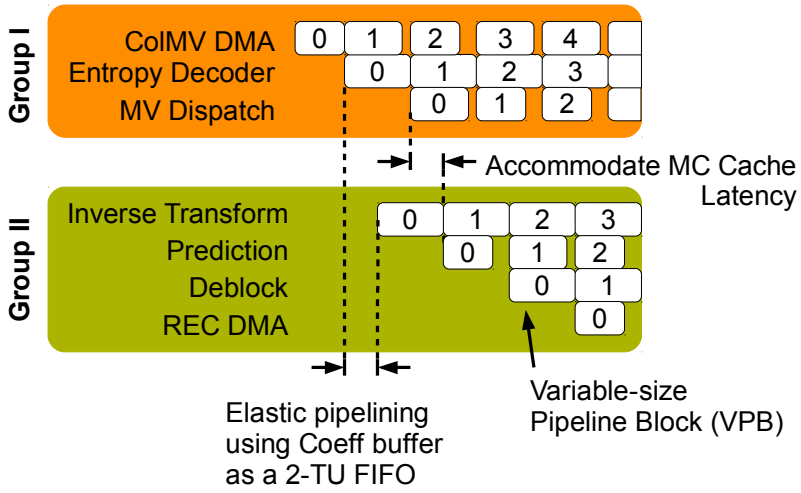


Fig. 2 Split system pipeline to address variable DRAM latency. Within each group, variable-sized pipeline block-level pipelining is used.

The entropy decoder must send residue coefficients and transform information such as quantization parameter and TU size to the inverse transform block. As residue coefficients use 16-bit precision, 12 kB of SRAM is needed for luma and chroma coefficients of one VPB. For full pipelining, storage for two VPBs is needed so that entropy decoder can write coefficients and inverse transform can read coefficients of the previous VPB simultaneously. Thus, VPB pipelining would need 24 kB of SRAM. But this can be avoided by using the fact that the largest TU size is 32×32 (a 64×64 CU must split its transform quadtree at least once). Hence, it is possible to use a 2-TU buffer instead. The entropy decoder writes to one TU while inverse transform reads from the previous TU. This buffer requires only 4 kB, thus saving 20 kB of SRAM.

In the first pipeline group, a line buffer is used by entropy decoder for storing prediction information of upper row VPBs. In the second pipeline group, the 9-bit residues are passed from inverse transform to prediction using 2 VPB-sized SRAMs in ping-pong configuration. (Inverse transform writes one VPB to one SRAM while prediction reads the previous VPB from the other SRAM. When both modules are finished processing their respective VPBs, the two SRAMs switch roles.) Prediction, in-loop filters and reconstruction DMA communicate using 3 VPB-sized SRAMs in a rotating buffer configuration as shown in Fig. 3. Another line buffer is used to communicate pixels and parameters across VPB rows. The line buffer must store:

- 4 luma and 2 chroma rows (pre-deblocking) for deblocking filter. Of these, 1 luma and 1 chroma rows are also used as top neighbor pixels for intra prediction.
- 1 luma and 1 chroma rows (post-deblocking) for SAO filter

- Prediction and transform parameters such as prediction mode, motion vectors, reference picture indices, intra-prediction mode and quantization parameter to determine deblocking filter parameters
- SAO parameters

To reduce the area of the line buffer, a single-port SRAM is used and requests from prediction, in-loop filters and reconstruction DMA are arbitrated. The access patterns of the three modules to the SRAM are designed to minimize the amount of collisions and the arbitration scheme gives higher priority to the deblocking filter as it has a lower margin in the cycle budget. This minimizes the performance penalty of the SRAM sharing.

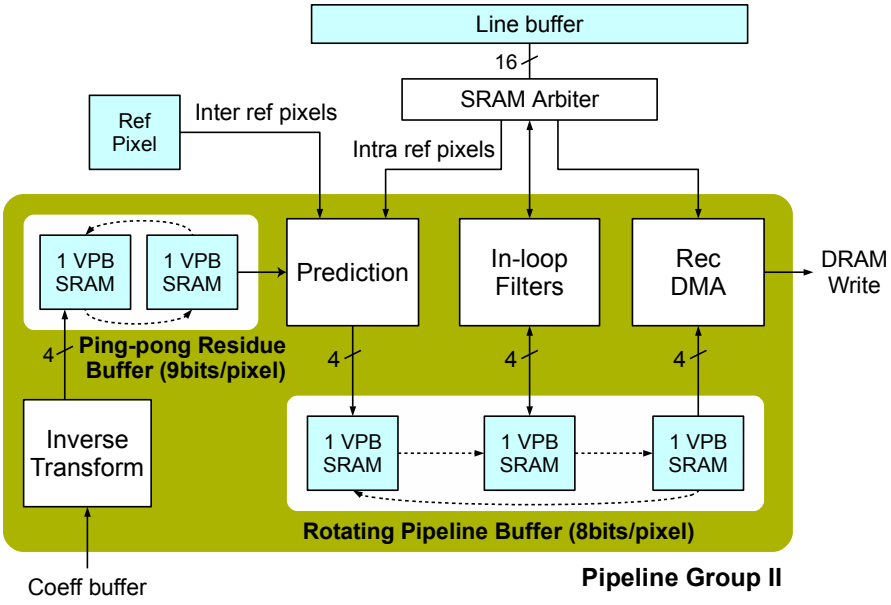


Fig. 3 Memory management in second pipeline group. A 2-VPB ping-pong and a 3-VPB rotating buffer are used as pipeline buffers. A single-port SRAM is used for pixel linebuffer to save area and access to it is arbitrated. Marked bus widths denote SRAM data width in pixels.

3 Entropy Decoding

HEVC uses a form of entropy coding called Context Adaptive Binary Arithmetic Coding (CABAC) to perform lossless compression on the syntax elements [5]. Fig. 4 shows the top level architecture of a CABAC entropy decoder. The arithmetic decoder decompresses the bitstream to generate a sequences of binary symbols (bins). The context selection finite-state-machine (FSM) determines which prob-

ability should be read from the context memory based on the type of the syntax element being processed, as well as the bin index, neighboring information (top neighbor is read from a line buffer), and component (i.e., luma or chroma). When the probability used to decode a bin is read from the context memory, it is referred to as a regular coded bin; otherwise, a probability of 0.5 is assumed and the bin is referred to as bypass coded. Bypass coded bins can be decoded much faster than regular coded bins. After each regular coded bin is decoded, an updated context with the updated probability estimate is sent back to the context memory. Finally, the debinarization module maps the sequence of bins to a syntax element.

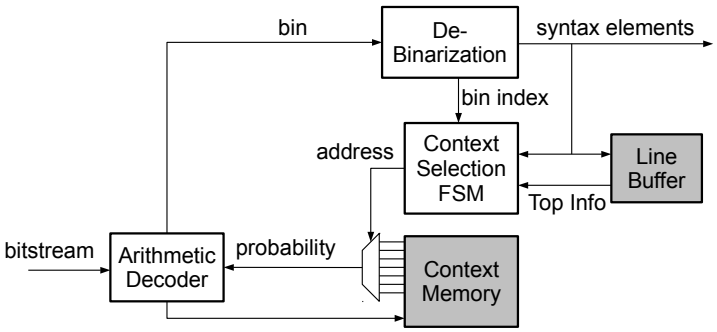


Fig. 4 Top-level architecture for CABAC. Memories are shown with grey boxes.

The CABAC in HEVC was redesigned for higher throughput [6]. Specifically, the CABAC in HEVC has fewer regular coded bins compared to H.264/AVC. In addition, the context selection FSM is simplified by reducing dependencies across consecutive states. Both the line buffer and context memory sizes are reduced. The number of types of binarization has increased in order to account for the reduction in regular coded bins, without coding loss. More details on this can be found in Chap. [CABAC]. HEVC uses the same arithmetic decoder as H.264/AVC.

3.1 Implementation Challenges

The challenge with CABAC is that it inherently has a different workload than the rest of the decoder. The workload of the entropy decoder varies based on bit-rate, while the rest of the decoder varies based on pixel-rate. The workload of CABAC can vary widely per block of pixels (i.e. CTU). Thus a high throughput CABAC is needed to order to handle the peaks of the workload variation to prevent stalls in the decoder pipeline. However, it is difficult to parallelize the CABAC due to its strong data dependencies. This is particularly true at the decoder where the data dependencies result in feedback loops. For H.264/AVC CABAC, decoders have throughput on the order of hundreds of Mbin/sec versus up to Gbin/sec for encoders.

3.2 Solutions

There are several approaches that have been explored to increase the throughput of CABAC, which is dictated by the number of binary symbols it can decode per second (bin-rate). One method is to pipeline the CABAC to reduce the critical path delay [7]. However, the deeper the pipeline, the more stalls or more speculative computations/branching required. Alternatively, multiple arithmetic decoders are concatenated to decode multiple bins per cycle [8, 9]. As the number of bins per cycle increases, the number of speculative computations increases exponentially and the critical path delay increases linearly. Finally, another approach is to decode a variable number of bins per cycle, and assume that the most probable bins are decoded each cycle [10]. As the number of bins increases, the number of speculative computations only increases linearly; however, the critical path delay also increases linearly and the number of bins decoded per cycle increases less than linearly, which results in lower bin-rate. More discussion on this can be found in [11]. To address these challenges, the CABAC in HEVC minimizes dependencies across consecutive bins, particularly for the residual coding, and has fewer regular coded bins in order to reduce the amount of speculative computation required when using the pipelining or multiple bins architectures. In addition, it also groups bypass bins to enable the decoder to fully leverage that fast decoding of bypass coded bins [12].

To address the imbalance in workload between entropy decoding (Group I in Fig. 2) and the rest of the decoder (Group II in Fig. 2), a very large buffer can be inserted after the entropy decoder to average out the workload. Note that the standard constrains the workload of the entropy decoder at the frame level (using max *BinCountsInNalUnits*) and across frames (using max bit-rate in the level limit); thus using frame level buffering between the entropy coder and the rest of the decoder can help to address this imbalance. This is commonly referred to as entropy decoupling. However, this comes at the cost of an additional frame delay and increased memory bandwidth. The memory bandwidth cost can be reduced if the intermediate values are stored as binary symbols of the CABAC rather than the reconstructed syntax elements [13]. An added advantage of having frame level buffering is that multiple rows of CTU can be decoded in parallel, since all the decoded syntax elements for the frame can be read from the buffer [14].

If latency cannot be tolerated, HEVC contains high level parallelism tools such as slices, tiles and wavefront parallel processing, which enable multiple CABAC decoders to operate in parallel on the same frame. However, there is no guarantee that these features will be enabled by the encoder.

4 Inverse Transform and Dequantization

Dequantization scales up coefficients decoded by the entropy decoder and inverse transform converts the scaled coefficients to residue pixels using a 2-D Inverse Discrete Cosine Transform (IDCT) or a 2-D Inverse Discrete Sine Transform (IDST).

As compared to H.264/AVC, the HEVC inverse transform involves significant challenges for hardware implementation. This is the result of the following factors:

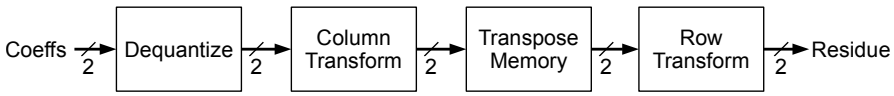
1. HEVC uses Transform Units (TUs) of size 4×4, 8×8, 16×16, and 32×32 pixels. This variety of TU sizes complicates the design of control logic as TUs of different sizes take different number of cycles for processing.
2. Like H.264/AVC, the 2-D transforms in HEVC are separable into 1-D transforms along the columns and rows. An $N \times N$ 2-D transform consists of N 1-D column transforms and N 1-D row transforms, each of which can be viewed as the product of an $N \times N$ transform matrix with $N \times 1$ input coefficients. The total number of multiplications is thus, $2N^3$ or $2N$ per coefficient. Hence, the largest IDCT in HEVC (32×32) takes 4× the number of multiplications per coefficient as compared to the largest IDCT in H.264/AVC (8×8). Furthermore, the increased precision in HEVC transforms doubles the cost of each multiplication. Hence, HEVC transform logic has 8× the computational complexity of H.264/AVC.
3. An intermediate memory is needed to store the TU between the column and row transforms operation. This memory must perform a transposition (i.e. columns are written to it and rows are read out). Previous designs for H.264/AVC used register arrays due to the small TU sizes. These do not scale very well to the higher TU sizes of HEVC and one must look to denser memories such as SRAM to achieve an area-efficient implementation. However, the higher density of SRAMs comes at the cost of lower memory throughput and less flexibility in read-write patterns.

A single-cycle 32-pt 1-D IDCT with Booth encoded shift-and-add multipliers takes about 145 kgate of logic. For comparison, a complete 1080p H.264/AVC decoder can be built in 160 kgate [15]. Hence, aggressive optimizations that exploit various properties of the transform matrix are necessary to achieve a reasonable area. Also, a single-cycle 32-pt IDCT provides much higher throughput than what is required for real-time operation. It is possible to reduce the area by computing the DCT over multiple cycles using partial matrix multiplication. A 2 pixel/cycle throughput at 200 MHz is sufficient for 4K Ultra HD decode at 30fps. The following subsections describe such a design.

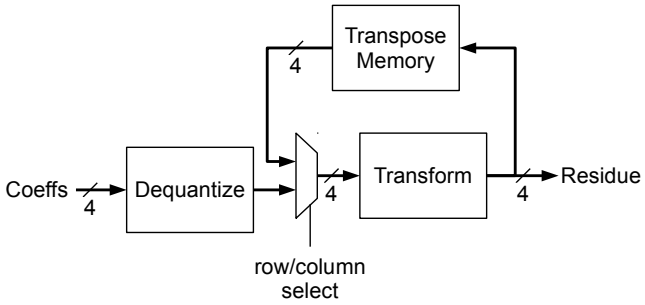
4.1 Top-level Pipelining

In general, two high-level architectures are possible for a 2 pixel/cycle inverse transform [16]. The first one, shown in Fig. 5(a) uses separate stages for row and column transforms. Each one has a throughput of 2 pixel/cycle and operates concurrently. The dependency between the row and column transforms (all columns of the TU must be processed before the row transform) means that the two stages must process different TUs at the same time. The transpose memory must have one read and one write port and hold two TUs - in the worst case, two 32×32 TUs. Also, the two TUs would take different number of cycles to finish processing. For example, if a

8×8 TU follows a 16×16 TU, the column transform must remain idle after processing the smaller TU as it waits for the row transform to finish the larger one. It can begin processing the next TU but managing several TUs in the pipeline at the same time will require complex control logic to avoid stalls.



(a) Separate row and column transform stages



(b) 1-D transform stage shared by row and column transform

Fig. 5 Possible high-level architectures for inverse transform with 2 pixel/cycle throughput. Buswidths are in pixels.

With these considerations, the second architecture, shown in Fig. 5(b) is preferred. This uses a single 4 pixel/cycle 1-D transform for both row and column transform to achieve the desired 2 pixel/cycle 2-D transform throughput. The 1-D transform works on a single TU at a time, processing all the columns first and then all the rows. Hence, the transpose memory needs to hold only one TU and can be implemented with a single port SRAM since row and column transforms do not occur concurrently.

4.2 Transpose Memory

The transform block uses a 16-bit precision input for both row and column transforms. The transpose memory must be sized for 32×32 TU which means a total size of 16 × 32 × 32 = 16.4 kbit. In comparison, H.264/AVC decoder designs require a much smaller transpose memory - 16 × 8 × 8 = 1 kbit. A 16.4 kbit memory with the necessary read circuit for the transpose operation takes up a lot of area (125 kgate) when implemented with registers and multiplexers. Also, the register-based transpose memory has a much higher throughput than required. SRAMs are more area-efficient than registers and have a lower throughput, which makes them

a good choice for an optimized implementation. The main disadvantage of SRAMs is that they are less flexible than registers. A register array allows reading and writing to arbitrary number of bits at arbitrary locations, although very complicated read(write) patterns would lead to a large output(input) mux size. The SRAM read or write operation is limited by the bit-width of its port. A single-port SRAM allows only one operation, read or write, every cycle. Adding extra ports is possible at the expense of significant area increase.

It is possible to implement the 4-pixel/cycle transpose memory using 4 single-port banks of 4096 bits each with a port-width of 1 pixel. The pixels in a 32×32 TU are mapped to locations in the 4 banks as shown in Fig. 6. By ensuring that 4 adjacent pixels in any row or column sit in different SRAM banks, it is possible to write along columns and read along rows by supplying different addresses to the 4 banks.

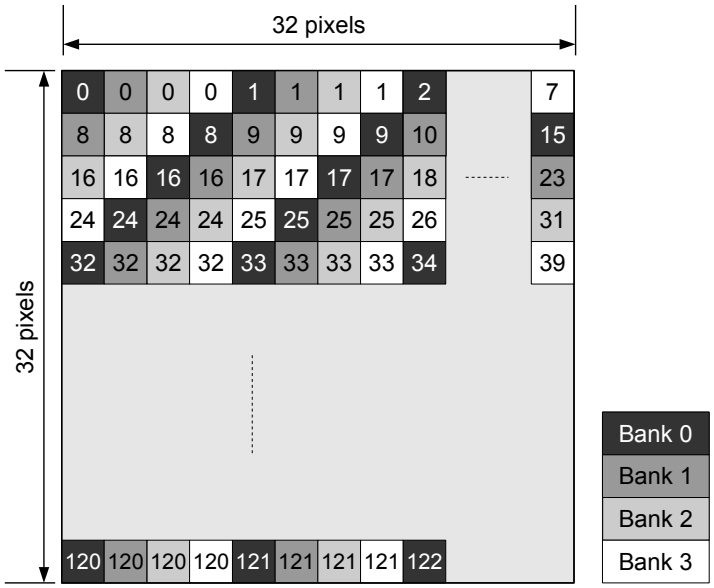
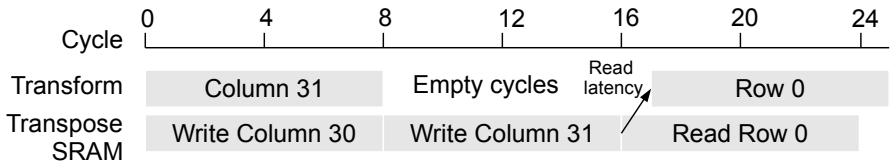


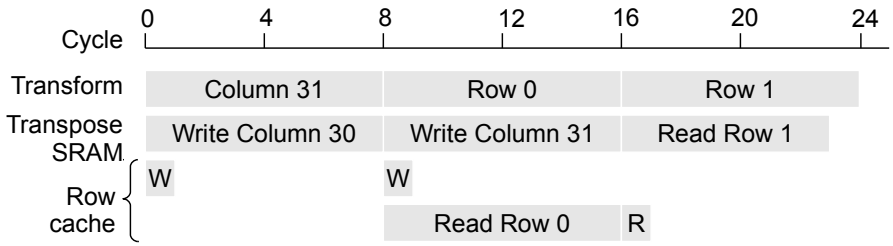
Fig. 6 Mapping a 32×32 TU to 4 SRAM banks for transpose operation. The color of each pixel denotes the bank and the number denotes the bank address.

After a 32-pt column transform is computed, the result is saved in a temporary register and is written to the transpose SRAM over 8 cycles. At the same time, the 1-D transform module processes the next column. This is shown in cycles $0-7$ in Fig. 7(a), where the result of column 30 is written to the SRAM while the 1-D transform module works on column 31. However, when the last column in a TU is processed, the transform module must wait for it to be written to the SRAM before it can begin processing the row. This results in a delay of 9 cycles for 32×32 TU. In general, for an N×N TU, this delay is equal to N/4 + 1 cycles. This results in a pipeline stall of 1.75% to 25% cycles depending on the TU size. This stall can be avoided through

the use of a row cache that stores the first N + 4 pixels in registers. As shown in Fig. 7(b), the row cache is read for the first 9 cycles of the row transforms while the last column is being stored in the SRAM.



(a) Pipeline stall due to transpose SRAM delay for 32×32 TU



(b) Row caching to avoid stall

Fig. 7 Eliminate read/write with registers for an SRAM-based transpose memory

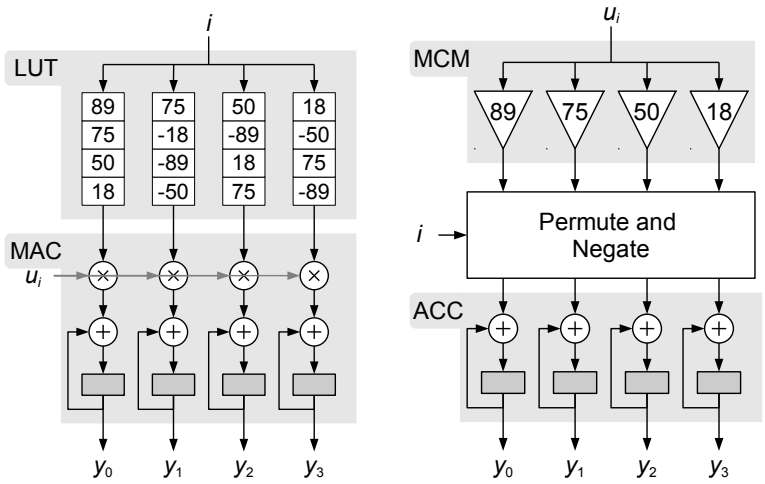
This transpose memory design using SRAM scales very well for lower throughputs. A 2-pixel/cycle transpose memory would need 2 banks each with 512 entries (16-bit/entry). For higher throughputs, one needs more banks each with fewer entries. Such short SRAM banks have a larger area overhead of sense-amplifiers and other read-out circuitry. For throughputs higher than 32-pixel/cycle, register based transpose memory [17] is more area-efficient.

4.3 Inverse DCT Engine

The IDCT engine can be optimized by observing that the N-pt IDCT matrix has at most N unique coefficients differing only in sign. This is also true of the matrices obtained by even-odd decomposition of the IDCT matrix, such as the 16×16 matrix of the 32-pt IDCT. This 256-element matrix contains 15 unique numbers: 90, 88, 85, 82, 78, 73, 67, 61, 54, 46, 38, 31, 22, 13, 4 (and their additive inverses). The matrix is multiplied with the odd-indexed coefficients in the 32-pt IDCT. In a 4-pixel/cycle case, only 2 of these inputs are available per cycle. So, it is enough to perform a partial 2×16 matrix multiplication every cycle and accumulate the outputs over 8 cycles. In general, this would require 32 full multipliers and 32 lookup tables to store the matrix. However, knowing that the matrix has only 15 unique numbers, we can simply instantiate 15 constant multipliers with some negators and multiplexers

to implement the matrix multiplication. This is shown for the 4×4 odd matrix multiplication (Eq. 1) of the 8-pt IDCT in Fig. 8(b). The area savings are shown in Table 2.

$$\begin{bmatrix} y_0 & y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} u_0 & u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} 89 & 75 & 50 & 18 \\ 75 & -18 & -89 & -50 \\ 50 & -89 & 18 & 75 \\ 18 & -50 & 75 & -89 \end{bmatrix} \tag{1}$$



(a) Generic implementation (b) Exploiting unique operations

Fig. 8 4×4 matrix multiplication in Eq. (1) without and with unique operations

Matrix multiplication	Area for generic implementation (kgates)	Area exploiting unique operations (kgates)	Area savings
4×4	10.7	7.3	32%
8×8	23.2	13.5	42%
16×16	46.7	34.4	26%

Table 2 Area reduction by exploiting unique operations

4.4 Implementation Results

Breakdown of the post-synthesis logic area at 200 MHz clock frequency in 40 nm CMOS is given in Table 3. The total area is 104 kgate of logic (in terms of 2-input NAND gates) and 16.4 kbit of SRAM.

Table 3 Area breakdown for inverse transform

Module	Logic area (kgates)
Partial transform	71
Accumulator	5
Row cache	4
FIFOs	5
Scaling + Control	19
Total	104

Table 4 Area for different transforms. Partial 32-pt IDCT contains all the smaller IDCTs

Module	Logic area (kgates)
4-pt IDCT	3
Partial 8-pt IDCT	10
Partial 16-pt IDCT	24
Partial 32-pt IDCT	57
4-pt IDST + misc.	14

5 Inter Prediction

HEVC inter prediction uses motion vectors pointing to one reference frame (uni-prediction) or two reference frames (bi-prediction) to predict a block of pixels. The size of the predicted block, called Prediction Unit (PU), is determined by the Coding Unit (CU) size and its partitioning mode. For example, a 32×32 CU with $2N\times N$ partitioning is split into two PUs of size 32×16, or a 16×16 CU with $nL\times2N$ partitioning is split into 4×16 and 12×16 PUs.

For luma pixels, the motion vectors for each PU have a resolution of 1/4-th pixel. The predicted pixels at non-integer pixel positions are obtained by interpolating between the reference pixels using an 8-tap FIR filter, first along the horizontal direction and then along the vertical as shown in Fig. 9. (The reverse order, i.e. vertical followed by horizontal also gives the same result). For chroma, the motion vector is halved and has a 1/8-th pixel resolution computed using a 4-tap interpolation filter. From Table 5, which shows the cost of interpolating a block of pixels, we see

that smaller pixel blocks have a proportionately higher overhead in the number of reference pixels and number of horizontal interpolations. To reduce the worst case overhead, 4×4 PUs are not allowed by the standard and 8×4/4×8 PUs are allowed to use only uni-prediction.

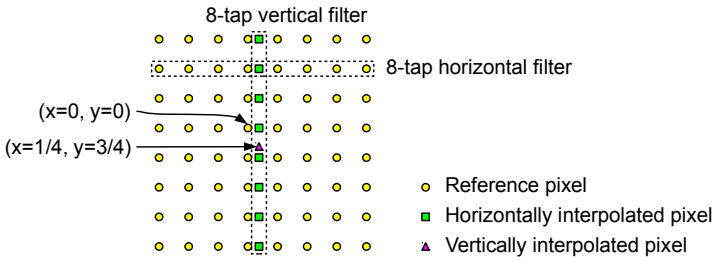


Fig. 9 Interpolation process for a pixel at a fractional location $x = 1/4, y = 3/4$.

Table 5 Example costs for interpolating a block of pixels. Values in brackets denote overhead over the block size. Costs are for uni-prediction only. For bi-prediction, all the costs are doubled.

	Block type	Generic	Y64×64	Y16×16	U4x4
Parameters	Block size	$w \times h$	64×64	16×16	4×4
	Filter size	$n+1$ taps	8 taps	8 taps	4 taps
Costs	Reference pixels	$(w+n) \times (h+n)$	71×71 (23%)	23×23 (106%)	7×7 (206%)
	Horizontal interps.	$w \times (h+n)$	64×71 (11%)	16×23 (43%)	4×7 (75%)
	Vertical interps.	$w \times h$	64×64 (0%)	16×16 (0%)	8×8 (0%)

Compared to H.264/AVC, HEVC uses

1. Larger PUs which require fewer interpolations per pixel but more on-chip SRAM
2. More varied PU sizes which increase complexity of control logic
3. Longer interpolation filters which require more datapath logic and more reference pixels

Reference frames may be stored in off-chip DRAM for HD and larger picture sizes, or in on-chip SRAM for smaller sizes. At a PU level, it is observed that reference pixels of adjacent PUs have significant overlap. Due to this spatial locality, fetching the reference pixels into a motion-compensation (MC) cache helps reduce the latency and power required to access external DRAM and large on-chip SRAMs. Considering this, a top-level architecture (showing only the data-path) for an HEVC inter-prediction engine would look like Fig. 10.

The Dispatch module generates the position and size of the reference pixel block according to the decoded motion vectors (MVs). The MC Cache will send read requests to reference frame buffer over the direct-memory-access (DMA) bus for cache misses. When all the reference pixels are present in the MC cache, the Fetch module will fetch them from the cache for the 2-D Filter module. Note that it could

take many cycles to get data from DMA bus, due to latencies of bus arbiters, DRAM controller, and DRAM Precharge/Activate operations.

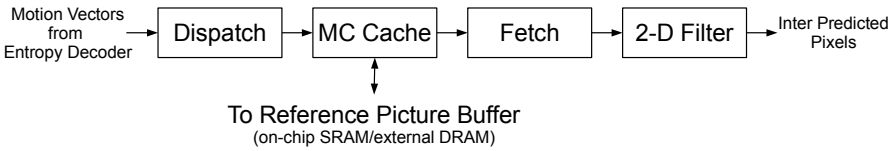


Fig. 10 System architecture for HEVC inter prediction. Only main data flow is shown.

The following subsections describe techniques used to address the important challenges of implementing HEVC inter prediction in hardware.

1. A fixed pipelining across the Dispatch, Fetch and 2-D Filter modules for simpler control and reduced on-chip SRAM
2. A PU-adaptive scheduling within each module to handle the variety of PU sizes
3. Time-multiplexed Multiple Constant Multiplication (TMMCM) [18] to reduce interpolation filter size

Section 6 describes the design of a motion compensation cache used to reduce the memory bandwidth requirement and power consumption of the reference picture buffer.

5.1 Fixed Pipelining across Modules

In HEVC, it is possible to predict a large block of pixels in smaller pipeline blocks by treating the smaller blocks as independent PUs with the same motion vector information. So, to deal with all the variety of PU sizes, one can use a constant block size of 4×4. This drastically reduces the size of pipeline buffers between the modules in Fig. 10. However, as explained previously, the smaller blocks have a larger overhead in terms of fetching reference pixels and performing horizontal interpolations.

In [4], 16×16 pipeline blocks are used to tradeoff SRAM size and computation overhead. For chroma, since a block of 16×16 luma pixels corresponds to two 8×8 chroma pixels in the 4:2:0 format, chroma pixels from two 16×16 blocks are combined and used as a single pipeline block of four 8×8 pixels. As compared to a 64×64 CTU granularity, this requires 24× smaller pipeline buffers. The worst case overhead of this scheme is seen when a 64×64 PU is split into 16×16 pipeline blocks. For luma pixels, this PU originally requires $64 \times 71 = 4544$ horizontal interpolations but processing it in smaller blocks increases that by 30% to $16 \times (16 \times 23) = 5888$. For PU sizes smaller than 16×16, multiple such PUs are combined into one pipeline block.

5.2 PU-Adaptive Pipelining in 2-D Filter

The 2-D Filter must handle PUs of size 16×16 and smaller for luma and chroma which require different number of interpolations as shown in Table 6. Y16×4 PU requires the most number of horizontal interpolations (5.5 per pixel) and so, for a 2 pixel/cycle throughput, 11 horizontal filters are required. By a similar analysis, 4 vertical filters are required. However, this would result is a mismatch between the peak throughput of the horizontal filters (11 pixel/cycle) and the vertical filters. The designer can choose to add a buffer after the horizontal filters to handle the mismatch or match the peak throughput with 11 vertical filters.

Table 6 Number of horizontal interpolations for each PU type. Some PU types are restricted to uni-prediction while other types can use either.

PU Type	Uni/bi directional	No. of horizontal interpolations per PU	per pixel	No. of vertical interpolations per PU	per pixel
Y16×16	Uni/bi	2×16×23	2.875	2×16×16	2
Y8×8	Uni/bi	2×8×15	3.75	2×8×8	2
Y16×4	Uni/bi	2×16×11	5.5	2×16×4	2
Y4×16	Uni/bi	2×4×23	2.875	2×4×16	2
Y8×4	Uni	8×11	2.75	8×4	1
Y4×8	Uni	4×15	1.875	4×8	1
UV8×8	Uni/bi	2×8×11	2.75	2×8×8	2
UV4×4	Uni/bi	2×4×7	3.5	2×4×4	2
UV8×2	Uni/bi	2×8×5	5	2×8×2	2
UV2×8	Uni/bi	2×2×11	2.75	2×2×8	2
UV4×2	Uni	4×5	2.5	4×2	1
UV2×4	Uni	2×7	1.75	2×4	1

5.3 TMMCM for Interpolation Filter

The 6-tap interpolation filter in H.264/AVC is easy to optimize due to its symmetry and simple coefficients [19]. However, HEVC uses longer 8-tap and 4-tap filters for luma and chroma coefficients respectively, and the filter coefficients are also more complex. In [20], a 1-D luma filter design with 16 adders and a 2-D filter reuse scheme for sub-block 4×4 are proposed. A 1-D filter design using only 13 adders is also possible by unifying the luma and chroma filters into one single design and optimizing it with time-multiplexed multiple-constant multiplication (TMMCM). TMMCM is similar to MCM seen in Section 4 on Inverse Transform. However, exactly one of the MCM outputs is needed every clock cycle and this allows further optimizations by placing multiplexers within the MCM adder tree. One such TMMCM optimization is explained in some detail next.

A reorder of the filter inputs is first applied to reduce complexity based on symmetry as shown in Fig. 11. Note that two sets of the chroma filter coefficients are

placed in x_1 and x_6, instead of x_2 and x_5, due to the similarity with the luma coefficients 4 and 1. There are only seven cases left. The design principle adopted here is to optimize TMMCM coefficients for each filter input. As an example, the design for x_3 is shown in Fig. 12.

Selection	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Y 0, U 0				64				
Y 1/4, 3/4	-1	4	-10	58	17	-5	1	0
Y 1/2	-1	4	-11	40	40	-11	4	-1
UV 1/8, 7/8		-2		58	10		-2	
UV 1/4, 3/4		-4		54	16		-2	
UV 3/8, 5/8		-6		46	28		-4	
UV 1/2		-4		36	36		-4	

Fig. 11 Unified luma and chroma interpolation filters with inputs reordered. The coefficients for x_3 (in dashed box) can be implemented with 2 adders and 3 multiplexers as shown in Fig. 12.

In the canonical signed digit representation, the coefficients have at most 3 non-zero digits which determines the number of adders to be 2. The non-zero digits are partitioned into three groups (n, m and r) such that each group has at most 1 non-zero digit. Finally, the three partitions are summed with partitions having similar bitwidths added first.

Compared to algorithmically generated filter designs using [21], this design has a 5%∼31% lower area as shown in Table 7.

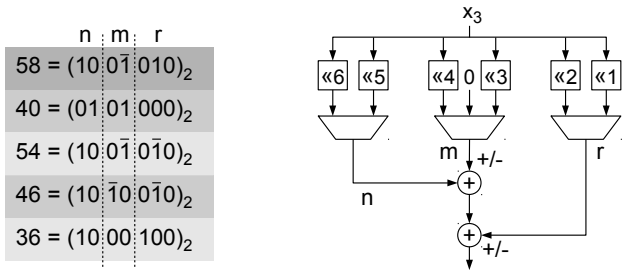


Fig. 12 Time-multiplexed Multiple Constant Multiplication for x_3

Combining all the presented techniques, the complete 1-D filter is shown in Fig. 13 using only 13 adders. Regarding the bitwidth increase between the input and output, the case of luma 1/2-pel position gives the largest values for both unsigned and signed inputs, and the outputs can be magnified at most by 88 and 112 times respectively. So, the 1-D horizontal filter has 8-bit unsigned input and 16-bit signed output, and the vertical one has 16-bit signed input and 23-bit signed output.

Table 7 Gate counts of the described and reference designs for the x_3, x_4, and $x_2|x_5$ TMMCM in the vertical filter based on 40 nm process synthesis results. The reference designs for x_3 and x_4 are generated by [21], and the reference for $x_2|x_5$ is designing x_2 and x_5 separately.

Design	x_3 TMMCM		x_4 TMMCM		$x_2\|x_5$	
Timing	1 ns	2 ns	1 ns	2 ns	1 ns	2 ns
Reference (gates)	1144	547	557	526	2284	845
Proposed (gate)	1036	518	442	361	1578	738
Area reduction	9.4%	5.4%	20.6%	31.4%	30.9%	12.6%

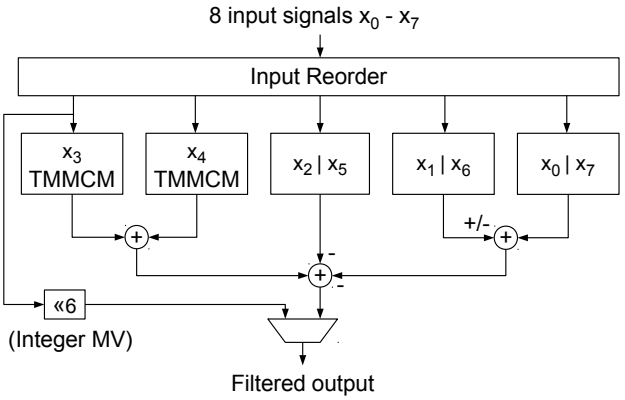


Fig. 13 HEVC interpolation filter design using 13 adders.

5.4 Implementation Results

For supporting 4K Ultra-HD 30 fps videos, this architecture is synthesized at 200 MHz in 40 nm CMOS. The result is shown in Table 8. The total gate count is 69.4k, of which 50.0k for the 2-D filter. The Fetch module mainly consists of large multiplexers and results in 12.0 kgate. The Dispatch module occupies 4.7 kgate for the block size and position calculation. The total SRAM size is 31 kbit, including the two-port 2.2 kbit Dispatch Info SRAM and the single-port 28.8 kbit Reference Data SRAM.

Since most of the gates are for the 2-D filter, its gate count is given in more detail in Table 9. For the 2-D filter, the horizontal and vertical filters occupy the most, and the area of horizontal ones is nearly one half of that of vertical ones due to their smaller internal bitwidth. This test-chip does not implement all PU Types used in the HEVC standard (Asymmetric Motion Partitions 32×8, 8×32, 16×4, 4×16 are not implemented), and so, uses only 8 horizontal and 8 vertical filters.

6 MC Cache and DRAM Mapping

HEVC's longer interpolation filters cause a significant increase in the required motion compensation (MC) bandwidth to the reference picture buffer (a.k.a. decoded

Table 8 Gate count of inter architecture when synthesized at 200MHz in 40 nm CMOS. SRAM sizes are also summarized.

Module	Logic area (kgates)	SRAM (kbit)
Dispatch	4.7	2.2 (two-port)
Fetch	12	28.8 (one-port)
2-D Filter	50.0	
Inter Ctrl	2.7	
Total	69.4	31

Table 9 Gate count breakdown for the 2-D filter

Sub-module	Logic area (kgates)
Input Mux	4.8
H Filter	12.0
V Filter	21.8
Register Chain	9.4
Bi-Sum	1.2
Ctrl	0.8
Total	50.0

picture buffer - DPB) as compared to H.264/AVC. However, there is significant overlap in the reference pixel data required by neighboring inter PUs which can be exploited by a cache. Most video codecs use DRAM based memory to store the DPB since it can be several megabytes large. In such a scenario, in addition to reducing the bandwidth requirement, the cache also hides the variable latency of the DRAM. This section describes the design of a read-only MC cache to support real-time decoding of 4K Ultra-HD HEVC video.

The target DRAM system is intended to store six reference pictures at 4K Ultra-HD resolution (corresponding to HEVC level 5) in addition to the collocated motion vector data. The DRAM system is composed of two 64M$\times$16-bit DDR3 DRAM modules with a 32 byte minimum access unit (MAU). A single MAU is mapped to a cache line.

6.1 DRAM Latency Aware Memory Map

An ideal mapping of pixels to DRAM addresses should minimize the number of DRAM accesses and the latency experienced by each access. This can be achieved by minimizing the fetch of unused pixels and the number of row precharge/activate operations respectively. Note that the above optimization only fixes how the pixels are stored in DRAM and can be performed even in the absence of an MC cache. Also, the DRAM addresses should be mapped to cache lines such that conflict misses are minimized. To enable a coherent presentation, we explain these ideas

with respect to a specific memory map. The underlying principles are quite general and can be easily reused.

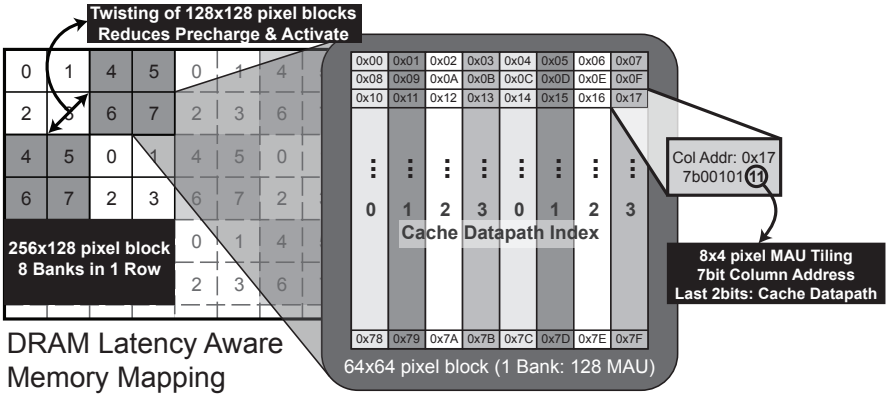


Fig. 14 Latency Aware DRAM mapping. 128 8×4 MAUs arranged in raster scan order make up one block. The twisted structure increases the horizontal distance between two rows in the same bank. Note how the MAU columns are partitioned into 4 datapaths (based on the last 2 bits of column address) for the four-parallel cache architecture.

Fig. 14 shows an example latency aware memory map. The luma color plane of a picture is tiled by 256×128 pixel blocks in raster scan order. Each block maps to an entire row across all eight banks. These blocks are then broken into eight 64×64 blocks which map to an individual bank in each row. Within each 64×64 block, 32-byte MAUs map to 8×4 pixel blocks that are tiled in a raster scan order. In Fig. 14, the numbered square blocks correspond to 64×64 pixels and the numbers stand for the bank they belong to. Note how the mapping of 128×128 pixel blocks within each 256×128 regions alternates from left to right. Fig. 14 shows this twisting behavior for a 128×128 pixel region composed of four 64×64 blocks that map to banks 0, 1, 2 and 3.

The chroma color plane is stored in a similar manner in different rows. The only notable difference is that an 8×4 chroma MAU is composed of pixel-level interleaving of 4×4 U and V blocks. This is done to exploit the fact that U and V have the same reference region.

Minimizing fetch of unused pixels Since the MAU size is 32 bytes, each access fetches 32 pixels, some of which may not belong to the current reference region as seen in Fig. 16. These can be minimized by using an 8×4 MAU to exploit the rectangular geometry of the reference region. When compared with a 32×1 cache line this reduces the amount of unused pixels fetched for a given PU by 60% on average.

Since the fetched MAU are cached, unused pixels may be reused if they fall in the reference region of a neighboring PU. Reference MAUs used for prediction at the right edge of a CTU can be reused when processing CTU to its right. However the

lower CTU gets processed after an entire CTU row in the picture. Due to limited size of the cache, MAUs fetched at the bottom edge will be ejected and are not reused when predicting the lower CTU. When compared to 4×8 MAUs, 8×4 MAUs fetch more reusable pixels on the sides and less unused pixels on the bottom. As seen in Fig. 15(a), this leads to a higher hit-rate. This effect is more pronounced for smaller CTU sizes where hit-rate may increase by up to 12%.

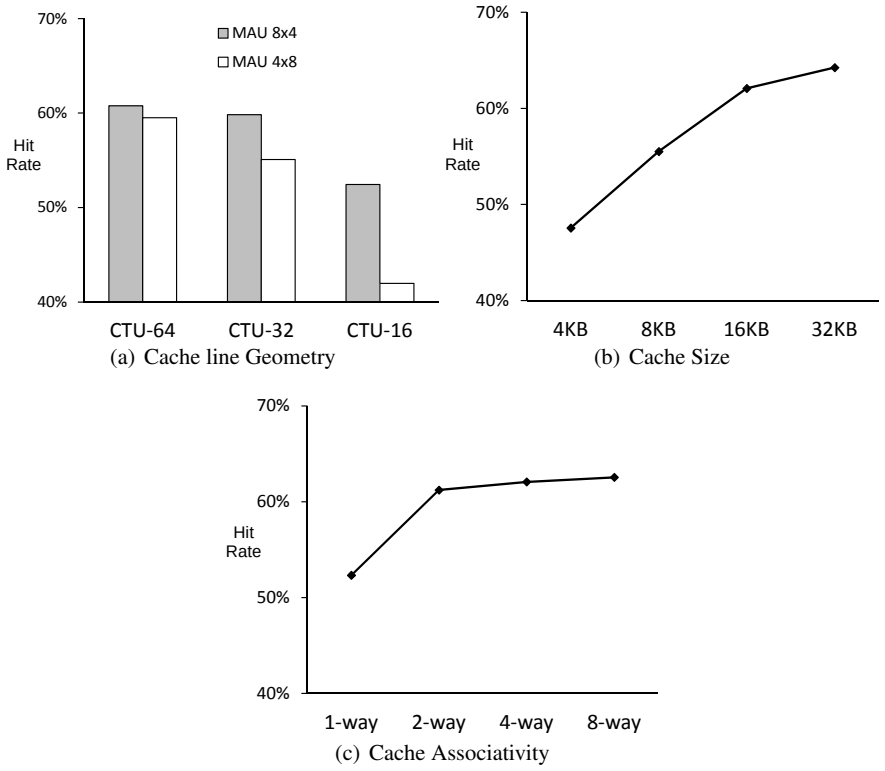


Fig. 15 Cache hit rate as a function of CTU size, cache line geometry, cache-size and associativity. Experiments averaged over six sequences - *Basketball Drive, Park Scene, Tennis, Crowd Run, Old Town Cross* and *Park Joy*. The first are Full HD (240 pictures each) and the last three are 4K Ultra HD (120 pictures each). CTU size of 64 is used for the cache-size and associativity experiments.

Minimizing row precharge and activation The Twisted 2D mapping of Fig. 14 ensures that pixels in different DRAM rows in the same bank are at least 64 pixels away in both vertical and horizontal directions. It is unlikely that inter-prediction of two adjacent pixels will refer to two entries so far apart. Additionally a single dispatch request issued by the MC engine can at most cover 4 banks. It is possible to keep the corresponding rows in the four banks open and then fetch the required data. These two factors help minimize the number of row changes. Experiments

show that twisting leads to a 20% saving in bandwidth over a direct mapping as seen in Table 10.

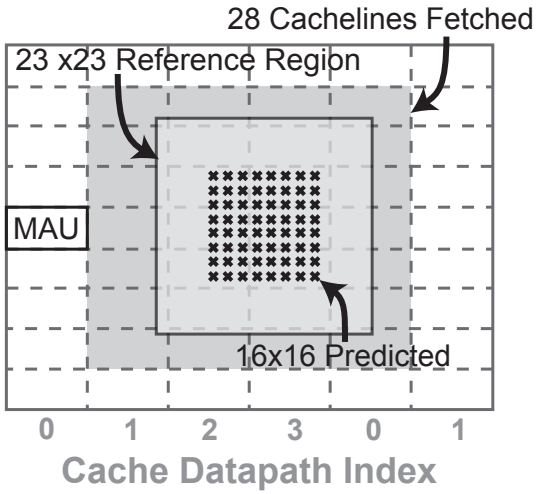


Fig. 16 Example of MC cache dispatch for a 23×23 reference region of a 16×16 PU. 7 cycles are required to fetch the 28 MAU at 4 MAU per cycle. Note that the dispatch region and the four parallel cache datapaths may be misaligned, thus requiring a reordering. For example, the region in this figure starts from datapath #1.

Table 10 Comparison of Twisted 2D Mapping and Direct 2D Mapping

Encoding Mode		LD			RA		
CTU Size		64	32	16	64	32	16
ACT BW	Direct 2D	272	227	232	690	679	648
(MBytes/s)	Twisted 2D	219	183	204	667	659	636
Gain		20%	20%	12%	3%	3%	2%

Minimizing conflict misses A conflict miss occurs when two locations in memory map to the same cache line. To mitigate this, we need to select an appropriate mapping between the DRAM addresses and the cache line indices. Setting the line index to the 7 bit column address of the MAU ensures that two conflicting pixel location in the same picture are at least 64 pixels apart. However, the same pixel location across two pictures will map to the same cache line. Similarly a luma and an unrelated chroma address may also map to the same cache line. Using 4-way set associativity in the cache helps resolve both these conflicts.

Alternative techniques to tackle conflict misses include having separate luma and chroma caches. Similarly offsetting the memory map such that the same location in

successive frames maps to different cache lines can also reduce conflicts. For our chosen configuration, the added complexity for these techniques outweighed the observed hit-rate increases.

6.2 Four-Parallel Cache Architecture

This section describes a four parallel MC cache architecture. Datapath parallelism and outstanding request queues for hiding the variable DRAM latency ensure a high throughput. As seen in Fig. 17, there are four parallel paths each outputting up to 32 pixels (1 MAU) per cycle.

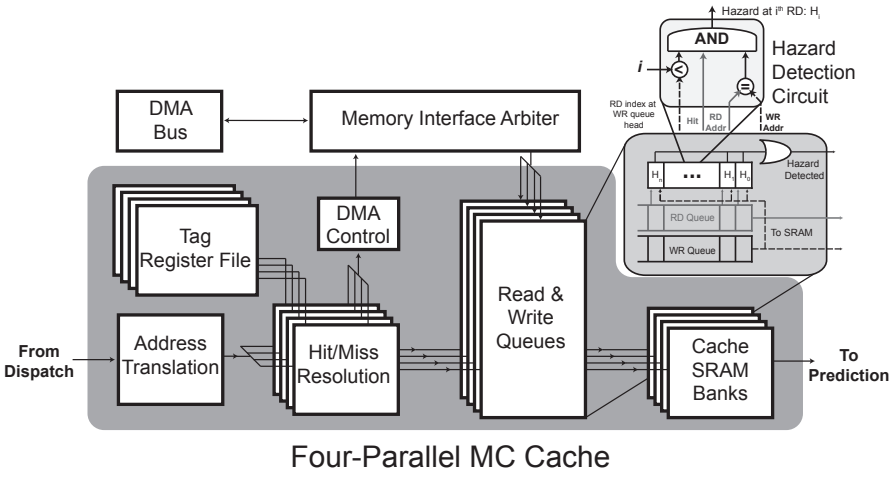


Fig. 17 Proposed four-parallel MC cache architecture with 4 independent datapaths. The hazard detection circuit is shown in detail.

6.2.1 Four-parallel data flow

The parallelism in the cache datapath allows up to 4 MAUs in a row to be processed simultaneously. The MC cache must fetch at most 23×23 reference region corresponding to a 16×16 PU, which is the largest PU processed by Inter Prediction (see Section 5.1). This may require up to 7 cycles as shown in Fig. 16. The address translation unit in Fig. 17 reorders the MAUs based on the lowest 2 bits of the column address. This maps each request to a unique datapath and allows us to split the tag register file and cache SRAM into 4 smaller pieces. Note that this design cannot output 2 MAUs in the same column on the same cycle. Thus our design trades unused flexibility in addressing for smaller tag-register and SRAM sizes.

The cache tags for the missed cache lines are immediately updated when the lines are requested from DRAM. This preemptive update ensures that future reads to the same cache line do not result in multiple requests to the DRAM. Note that behavior is similar to a simple non-blocking cache and does not involve any speculation. Additionally since the MC cache is a read only cache there is no need for write-back in case of eviction from the cache.

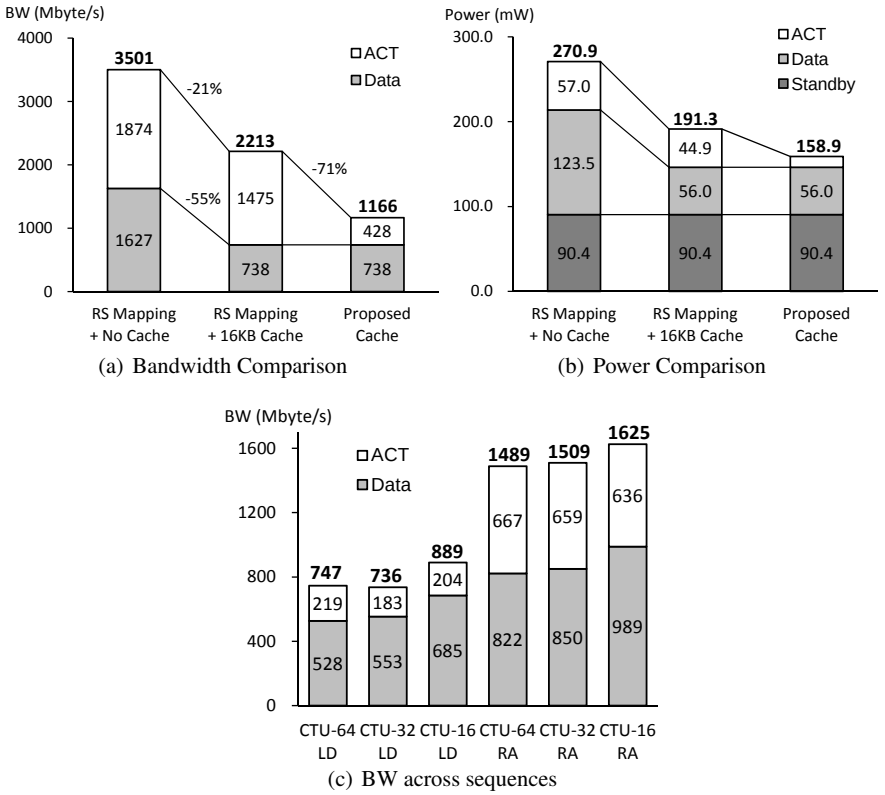
6.2.2 Queue management and hazard control

Each datapath has independent read and write queues which help absorb the variable DRAM latency. The 32 deep read queue stores pending requests to the SRAM. The 8 deep write queue stores pending cache misses which are yet to be resolved by the DRAM. The write queue is shorter because fewer cache misses are expected. Thus the cache allows for up to 32 pending requests to the DRAM. At the system level the latency of fetching the data from the DRAM is hidden by allowing for a separate motion vector (MV) dispatch stage in the pipeline prior to the Prediction stage. Thus, while the reference data of a given block is being fetched, the previous block is undergoing prediction. Note that the queue sizes here are decided based on the behavior of the target DMA arbiter and DRAM latency, and for different systems they should be optimized accordingly.

Since the cache system allows multiple pending reads, write-after-read hazards are possible. For example, consider two MAUs A and B that are mapped to the same cache line. Presently, the cache line contains A, the write queue contains a pending cache miss for B and the read queue contains pending requests for A and B in that order. If B arrives from the DRAM, it must wait until A has been read from the cache to avoid evicting A before it has been read. The Hazard Detection Circuit in Fig. 17 detects this situation and stalls the write of B.

6.2.3 Cache parameters

Figs. 15(b) and 15(c) show the hit-rates observed as a function of the cache size and associativity respectively. A cache size of 16 kB was chosen since it offered a good compromise between size and cache hit-rate. The performance of FIFO replacement is as good as Least Recently Used replacement due to the relatively regular pattern of reference pixel data access. FIFO was chosen because of its simple implementation. The cache associativity of 4 is sufficient to accommodate both Random Access GOP structures and the three component planes (Y, U, V).



(a) Bandwidth Comparison (b) Power Comparison

(c) BW across sequences

Fig. 18 Comparison of DDR3 bandwidth and power consumption across 3 scenarios. RS mapping maps all the MAUs in a raster scan order. ACT corresponds to the power and bandwidth induced by DRAM Precharge/Activate operations.

6.3 Hit Rate Analysis, DRAM Bandwidth and Power

The rate at which data can be accessed from the DRAM depends on two factors: the number of bits that the DRAM interface can (theoretically) transfer per unit time and the precharge latency caused by the interaction between requests. The precharge latency can be normalized to bandwidth by multiplying with the bitwidth. This normalized figure (called ACT BW) is the bandwidth lost in the precharge and activate cycles - the amount of data that could have been transferred in the cycles when the DRAM was executing row change operation. The other figure, Data BW, refers to the amount of data that needs to be transferred from the DRAM to the decoder per unit time for real-time operation. Thus, a better hit-rate reduces the Data BW and a better memory map reduces the ACT BW. The advantage of defining Data BW and ACT BW as mentioned above is that (Data BW + ACT BW) is the minimum bandwidth required at the memory interface to support real-time operation.

The performance of the cache and the twisted address mapping is compared with two reference scenarios: raster-scan address mapping with no cache and raster scan address mapping with the cache. As seen in Fig. 18(a), using a 16 kB cache reduces the Data BW by 55%. The Twisted 2D mapping reduces ACT BW by 71%. Thus, the cache results in a 67% reduction of the total DRAM bandwidth. Using a simplified power consumption model [22] based on the number of accesses, this cache is found to save up to 112 mW, a 41% reduction in DRAM access power as shown in Fig. 18(b).

Fig. 18(c) compares the DRAM bandwidth across various encoder settings. Smaller CTU sizes result in a larger bandwidth because of lower hit-rates. Thus, larger CTU sizes such 64 can provide smaller external bandwidth at the cost of higher on-chip complexity. Also, Random Access mode typically has lower hit rate when compared to Low Delay. This behavior is expected because the reference pictures are switched more frequently in the former.

6.4 Implementation Results

This design is synthesized at 200 MHz in 40 nm CMOS. The total area is 90.4 kgate of logic and 16 kB (or 131.1 kbit) of SRAM. The bulk of the logic area is taken by the 8960 bit tag register file and can be replaced by a 2-port SRAM (which is denser than register file) at the cost of an extra access cycle. Breakdown of the logic area is presented in Table 11.

Table 11 Breakdown of logic area for motion compensation cache

Module	Logic area (kgate)
Address Translation	1.1
Hit/Miss Resolution	3.9
Queue	20.5
Tag Register File	64.9
Total	90.4

7 Intra Prediction

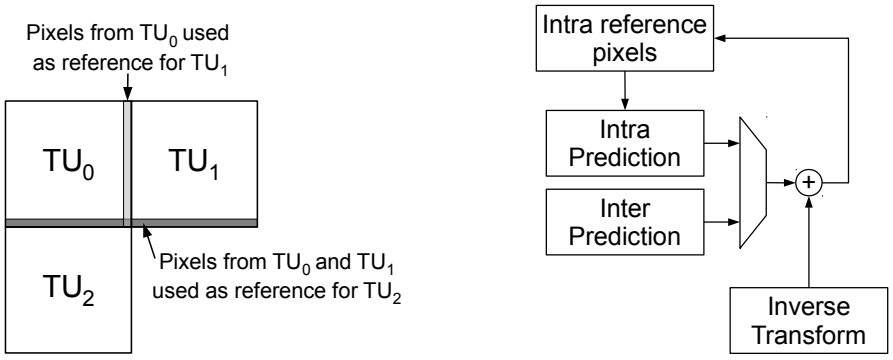
Intra prediction predicts a block of pixels based on neighboring pixels in the same picture. The neighboring pixels are extrapolated into the block to be predicted along one of 33 directions or using two other intra modes - DC and Planar. The neighboring pixels are taken from one row of pixels to the top and one column to the left.

The key operations in intra-prediction are:

1. Read neighboring pixels and perform padding for unavailable pixels
2. Reference preparation: filter neighboring pixels to obtain intra reference pixels and extend the top-left reference pixels for angular modes
3. Prediction: bilinear interpolation for angular and planar modes, and pixel copy for DC, horizontal and vertical modes

When the current block of pixels is predicted, its residues need to be immediately added so that it can be used as neighboring pixels for the next block. This results in a tight feedback loop for intra-prediction as shown in Fig. 19. As a result of this feedback loop, it is not possible to pipeline the above three operations, which increases the throughput requirement from these blocks. It should be noted that the feedback loop operates at a TU granularity and not a PU granularity. For example, for a 16×16 CU with a 2N×2N intra partition (i.e. a single 16×16 PU) and a residue quad tree (RQT) of four 8×8 TUs, the 8×8 blocks must be predicted serially and the intra neighboring pixels must be updated after every block's prediction and reconstruction.

This dependency also has implications for the top-level pipelining - in order to keep inverse transform and prediction decoupled, the inverse transform must be performed one pipeline granularity before prediction.



(a) Intra-prediction dependency between neighboring pixel blocks

(b) Dependency results in a tight feedback loop

Fig. 19 Tight feedback loop in intra prediction due to dependency between neighbors

The 35 intra prediction modes in HEVC are well designed to reduce complexity. The planar mode is much simpler than the one in H.264/AVC, and the 33 angular modes are also well organized to avoid increasing the complexity when increasing the angular precision. However, the larger TU sizes increase the hardware complexity due to larger pipeline and reference buffers. In H.264/AVC, one macroblock can contain only one kind of intra block size, which can be used to design optimized pipeline schedules as in [23, 24]. Since a CTU in HEVC can have a variety of TUs

and a mix of intra and inter CUs, such pipeline schedules will be too complex to optimize for every possible combination.

As the result, designing a data-flow that respects across-TU dependencies and provides high throughput is a bigger challenge than the pixel computation involved in reference preparation and prediction. In this chapter, we focus on the data-flow management used in [25], which uses a hierarchical memory deployment for high throughput and low area. The intra engine operates on blocks of 32×32 luma pixels and two 16×16 chroma pixels since those are the largest TU sizes. In the complete decoder pipeline, it communicates with entropy decoder and inverse transform at a Variable-sized Pipeline Block (VPB) granularity. (The mapping between VPB and CTU is shown in Table 2.1. For 16×16 CTU, four CTUs are combined into one intra pipeline block.)

7.1 Hierarchical Memory Deployment

The bottom row pixels of all VPBs in a row of VPBs needs to be stored since they are top neighbors for VPBs in the row below. This buffer must be sized proportional to the picture width and may be implemented in on-chip SRAM or external DRAM. Storing VPB-level neighboring pixels in registers as previous designs for H.264/AVC have done can provide the required high-throughput access. But this will require a lot of area as the VPB can be as large as 64×64. This issue can be addressed by storing the neighboring pixels in SRAM to save area and storing them in registers at a TU level for higher throughput. A memory hierarchy is thus formed:

1. VPB-row-level top neighbors in SRAM or external memory
2. VPB-level neighboring pixels in SRAM
3. TU-level reference pixels in registers

The hierarchical memory deployment is shown in Fig. 20 and the memory elements are explained next:

1. VPB-Row top neighbors: In [4], this buffer is implemented in an on-chip SRAM that is shared with deblocking filter. The deblocking filter stores 4 top rows of which, intra prediction uses one row.
2. VPB top neighbors: This buffer is implemented using a pair of SRAMs in a ping-pong fashion. One SRAM is used in the intra-prediction of the current VPB. It is updated every TU with neighboring pixels for the next TU. At the same time, the other SRAM updates the VPB-Row top SRAM with pixels from the previous VPB and loads top row pixels for the next VPB. The size of each SRAM is 192 pixels (64 Y top + 32 Y top-right + 64 UV top + 32 UV top-right).
3. VPB left neighbors: This buffer is implemented using one SRAM containing 128 pixels (64 Y + 64 UV). It is updated every TU with neighboring pixels for the next TU. Because the TUs are processed in z-scan order, at the end of all TUs in the current VPB, it automatically contains the left neighbors for the next VPB.

4. VPB top-left neighbors: The TU-based update scheme for VPB top and left neighbors could overwrite some pixels which will be the top-left neighbor of some following TUs. The VPB top-left neighbor buffer is introduced to solve this problem. As shown in Fig. 20, pixels on the 4×4 grid are written to the VPB top-left neighbor buffer.
5. Reference pixels: At the start of every TU, neighbors are read from the VPB-level SRAMs into registers. Padding and preparation operations are then performed on the registers to obtain reference pixels. Using registers allows for these operations and the final intra prediction to be performed at a high throughput. A total of 129 reference pixels (32 bottom-left, 32 left, 1 top-left, 32 top, 32 top-right) are needed for all angular modes. But since only one angular mode is used at a given time, the horizontal modes can be treated as vertical modes by swapping x and y axes to reduce the number of reference pixels to 99. Reference pixels are read by both preparation and prediction, and a combined read-out circuit shared between the two operations can reduce the number of multiplexers by exploiting similarities in their access patterns.

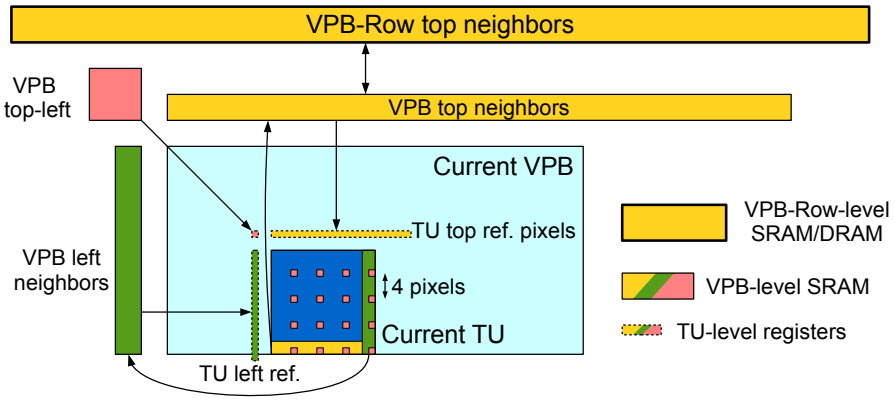


Fig. 20 Hierarchical memory deployment with VPB-Row level SRAM/DRAM and VPB-level SRAM for neighboring pixels, and TU-level registers for reference pixels

7.2 Reference Preparation and Prediction

As mentioned in Section 7, due to the tight dependency loop in Intra processing it is hard to pipeline the three pixel processing operations of reference padding, reference preparation and prediction. Another factor is that the three operations require different amount of computation. For an $N \times N$ TU, reference padding and preparation require $O(N)$ computation while prediction is $O(N^2)$.

The reference preparation operation in HEVC varies depending on the prediction mode. DC mode requires the accumulation of the reference pixels in order to compute the DC value. An angular extension of the reference pixels is required before prediction can begin. A mode dependant intra smoothing (MDIS) filter is applied to the reference pixels for TU sizes 8, 16 and 32 depending on the intra mode.

7.3 Implementation Results

Table 13 shows the synthesis results for the intra prediction architecture in 40 nm CMOS. Reference pixel registers and their read-out take the most area. The area for reference preparation, which is a new feature in HEVC, is about 1.3 kgate. The design is synthesized at 200 MHz and can support 4K Ultra-HD decoding at 30 fps.

Table 12 SRAMs for neighboring pixels

SRAM	Bits
VPB top	3072
VPB left	1024
VPB top-left	768
Total	4864

Table 13 Gate-count (in kgates) breakdown for Intra prediction

Module	Logic area
Reference pixel registers and padding	12.1
Reference pixel preparation	1.3
Prediction	8.1
Control	5.5
Total	27.0

8 In-loop Filters

HEVC uses two in-loop filters - deblocking filter and sample adaptive offset (SAO) - that attempt to reduce compression artifacts and improve coding efficiency. The deblocking filter in HEVC processes edges on an 8-pixel grid and thus, has lower computational complexity than H.264/AVC's deblocking filter which uses a 4-pixel grid. SAO involves selecting an offset type for each pixel based on its neighboring pixels and adding the offset. Deblocking and SAO can be implemented in a single pipeline stage as described in [26].

In [4], a VPB-based pipelining is used between deblocking filter and prediction stages. This allows the scheduling within the deblocking filter to be scheduled independent of the coding tree structure. A smaller granularity can also be used to save pipeline buffer SRAM at the cost of scheduling complexity. Since the in-loop filtering process for the current block of pixels depends on blocks to the right and bottom which have not yet been reconstructed, the entire block cannot be processed completely. The output of the deblocking filter is shifted from the input by 4 luma pixels and 2 chroma pixels to the left and the top, and the output of SAO is shifted by another pixel for all color components in both directions.

8.1 Deblocking Filter

Compared to H.264/AVC, HEVC's deblocking filter has several simplifications related to processing dependencies. The luma deblocking filter operates on edges lying on an 8×8 grid and filter takes 4 pixels on either side of the edge as input and writes up to 3 pixels on either side. As a result, unlike H.264/AVC, filters on adjacent edges are completely decoupled and it is possible to filter 8×8 pixel blocks independently. The key challenge in the deblocking filter architecture is designing an efficient data flow to handle cross-CTU dependencies.

The bottom four rows and right-most four columns of luma pixels (and two rows and columns of chroma pixels) in a CTU depend on the CTUs to the bottom, right and bottom-right for their deblocking. Accordingly, their processing must be delayed until those CTUs are available and they must be temporarily stored until then. Along with the pixels, parameters such as prediction mode, motion vectors, TU and PU boundaries, and quantization parameter which are required for computing the boundary strength also need temporary storage. The right-most four columns need a 1-CTU-high buffer (called Last CTU buffer) while the bottom four rows need a 1-Picture-wide buffer (called Line buffer).

The boundary strength parameters are available at a worst-case granularity of 4×4 pixels and take about 78 bits (64 bits for two motion vectors, 4 bits for two reference list indices, 6 bits for quantization parameter, 2 bits for prediction mode - intra-prediction, uni-prediction, bi-prediction - and one bit each for TU boundary, PU boundary). For example, for a 4K Ultra-HD (3840×2160) picture and 64×64 CTU, the Last CTU buffer must hold 64×4 luma pixels, $2\times32\times2$ chroma pixels and 16 boundary strength parameters resulting in a total of 4320 bits. The Line buffer must hold 3840×4 luma pixels, $2\times1920\times2$ chroma pixels and 960 boundary strength parameters resulting in a total of 96 kbit. While the Last CTU buffer can be stored in registers or SRAM, it might be necessary to store the Line buffer in external DRAM depending on area constraints. However, due to the regular access pattern on the Line buffer, it is possible to prefetch the data and hide the DRAM bandwidth (at the cost of on-chip memory for request and response queues to and from the DRAM).

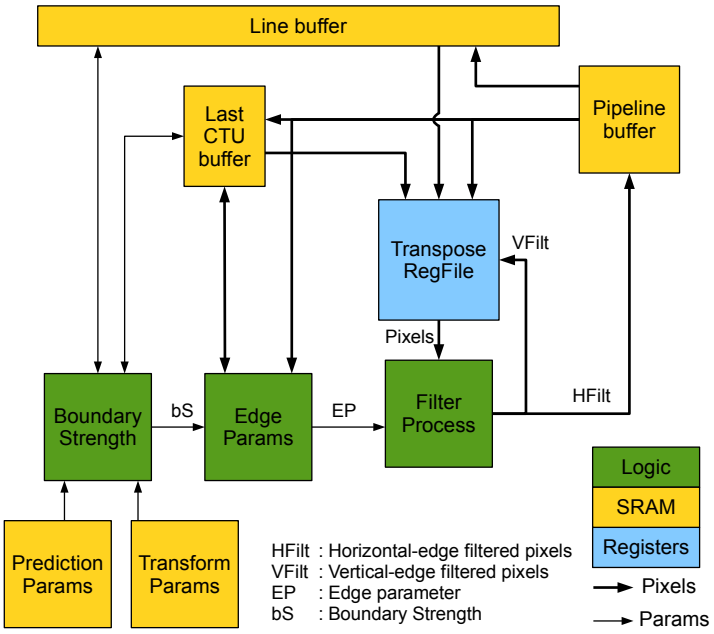


Fig. 21 Top-level architecture of deblocking filter

The top-level architecture of the deblocking filter is shown in Fig. 21. The transpose memory needs to be only 8×8 pixels (as compared to 32×32 pixels for inverse transform). Hence it is possible to implement it using registers. For a very high throughput design which filters an entire 8×8 block in one cycle [26], it is possible to eliminate the transpose memory completely and have a purely combinational design.

8.2 Sample Adaptive Offset (SAO)

SAO classifies each pixel into one of four bands or one of four edge types and adds an offset to it. For band offsets, the band of each pixel depends on its value and the position of the four bands. For edge offsets, the edge of each pixel depends on the whether its value is larger or smaller than two of its neighbors. The selection between band offsets and edge offsets, position of bands, choice of neighbors for edge offsets, and values of the offsets are signaled at the CTU level for luma and chroma separately. For chroma, the offsets are also signaled for the two components separately.

SAO has dependencies on neighboring pixels similar to intra prediction and hence, a similar data-flow management must be used. Like intra prediction, a picture-width sized top row buffer and a CTU-height sized left column buffer are

needed. These buffers store pre-SAO pixels and their SAO parameters. However, unlike intra prediction, the choice of pipeline granularity is very flexible and can be chosen based on throughput requirements. Unlike deblocking filter which operates on a edge basis, SAO operates on a per-pixel basis. So, the two in-loop filters have a comparable computational complexity even though SAO computation involves mainly comparison and addition.

[26] describes an architecture for SAO that is capable of 8K Ultra-HD (7680x4320) at 120 fps. In spite of such high throughput requirement, the design takes only 36.7 kgates in 65 nm technology.

9 Implementation Results for Decoder Test Chip

A decoder test chip was implemented in [4] with a core size of $1.77mm^2$ in 40 nm CMOS, comprising 715K logic gates and 124KB of on-chip SRAM. Fig. 22 shows the micrograph of the test chip. It is compliant to HEVC Test Model (HM) 4.0, and the supported decoding tools in HEVC Working Draft (WD) 4 are listed in Table 14 along with the main specs. The main differences from the final version of HEVC are that SAO is absent and Context-Adaptive Variable Length Coding (CAVLC) is used in place of CABAC in the Entropy Decoder. This chip achieves 249 Mpixels/s decoding throughput for 4K Ultra HD videos at 200 MHz with the target DDR3 SDRAM operating at 400 MHz. The core power is measured for six different configurations as shown in Fig. 23. The average core power consumption for 4K Ultra HD decoding at 30 fps is 76 mW at 0.9 V which corresponds to 0.31 nJ/pixel. Logic and SRAM breakdown of the chip is shown in Fig. 24. Similar to H.264/AVC decoders, we observe that prediction has the most significant resource utilization. However, we also observe that inverse transform is now significant due to the larger transform units while deblocking filter is relatively small due to simplifications in the standard. Power breakdown from post-layout power simulations with a bi-prediction bitstream is shown in Fig. 25. We observe that the MC cache takes up a significant portion of the total power. However, the DRAM power saving due to the cache is about six times the cache's own power consumption.

Table 15 shows the comparison with state-of-the-art video decoders. We observe that the $2\times$ compression efficiency of HEVC comes at a proportionate cost in logic area. The SRAM utilization is much higher due to larger coding units and use of on-chip line-buffers.

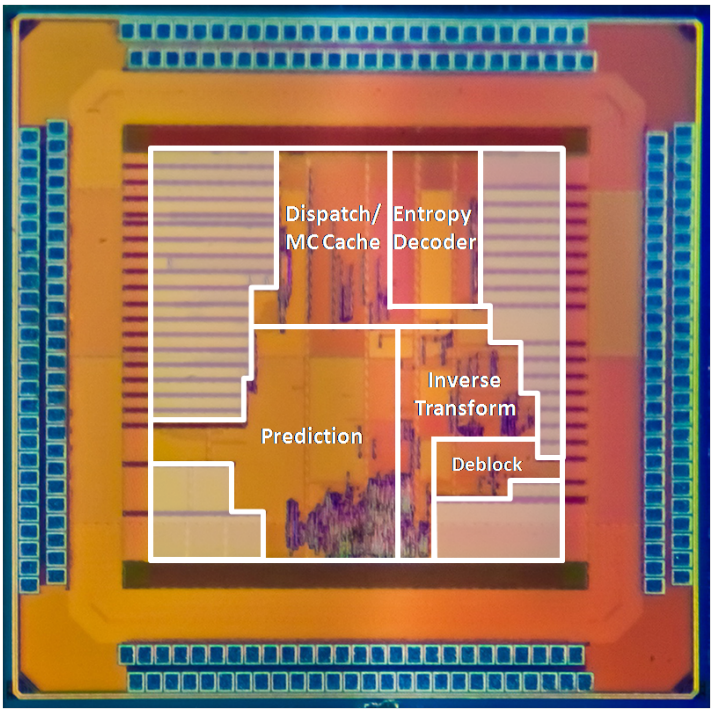


Fig. 22 Chip micrograph. Main processing engines are highlighted and light grey regions represent on-chip SRAMs.

Table 14 Chip Specifications

Technology	TSMC 40 nm CMOS
Supply Voltage	Core: 0.9 V, I/O: 2.5 V
Chip Size	2.18mm×2.18mm
Core Size	1.33mm×1.33mm
Gate Count	715K (2-input NAND)
On-Chip SRAM	124 kB
Maximum Throughput	249 Mpixel/s @ 200 MHz
Decoding Tools	HEVC WD4 (HM 4.0 low complexity w/o SAO) CTU size: 64×64, 32×32, 16×16 B-frame: Low Delay(LD)/Random Access(RA) Symmetric and asymmetric motion partitions: 4×4 - 64×64 Square and non-square transform units: 4×4 - 32×32 All intra modes: DC, Planar, 33 Angular, LMChroma
Measured Core Power	76 mW @ 0.9 V 200 MHz, 3840×2160 @ 30fps (average) 51 mW @ 0.9 V 100 MHz, 1920×1080 @ 60fps (average) 31 mW @ 0.9 V 25 MHz, 1280×720 @ 30fps (average)

Table 15 Comparison with state-of-the-art video decoders

	HEVC test chip [4]	A-SSCC'13 [3]	ISSCC'12 [2]	JSSC'11 [27]	ISSCC'10 [28]	JSSC'09 [29]	ISSCC'07 [30]
Standard	HEVC WD4	HEVC	H.264/AVC HP/MVC	H.264 HP	H.264/AVC HP SVC/MVC	H.264/AVC BP	JPEG, MPEG-1/2, MPEG-4, H.264 BP
Maximum Specification	3840×2160 @ 30 fps	1920×1080 @ 35 fps	7860×4320 @ 60 fps	4096×2160 @ 60 fps	4096×2160 @ 24 fps	1280×720 @ 30 fps	1920×1088 @ 30 ps ×1080 @ 30 fps
Gate count	715K	447K	1338K	662K	414K	315K	252K
On-chip SRAM	124KB	10KB	80KB	60KB	9KB	17KB	5KB
Technology	40 nm/0.9 V	90 nm/1.0 V	65 nm/1.2 V	90 nm/1.0 V	90 nm/1.0 V	65 nm/0.7 V, 0.85 V	130 nm/1.2 V
Core power	76 mW	139 mW	410 mW	189 mW	60 mW	1.8 mW	71 mW
Normalized core power	0.31 nJ/pixel	1.92 nJ/pixel	0.21 nJ/pixel	0.36 nJ/pixel	0.28 nJ/pixel	0.07 nJ/pixel	1.13 nJ/pixel
Normalized DRAM power	0.88 nJ/pixel	N/A	1.27 nJ/pixel	1.11 nJ/pixel	N/A	N/A	N/A
Normalized system power	1.19 nJ/pixel	N/A	1.48 nJ/pixel	1.47 nJ/pixel	N/A	N/A	N/A
DRAM configuration	32b DDR3	N/A	64b DDR2	64b DDR	N/A	ZBT SRAM	SDR

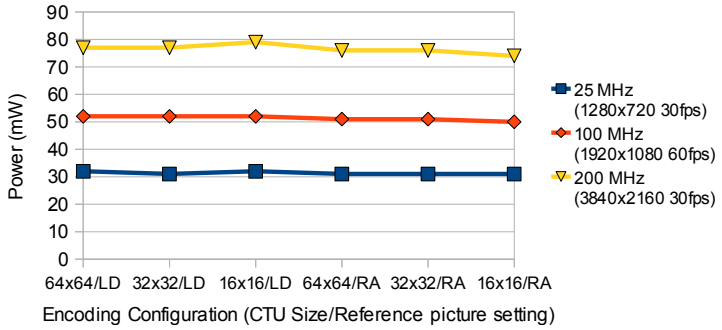
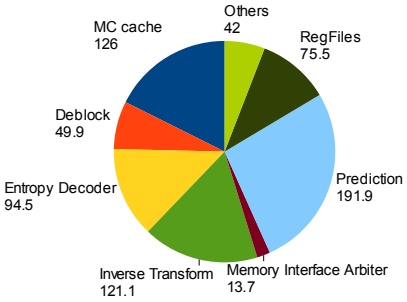
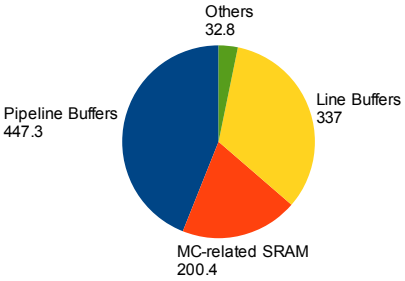


Fig. 23 Core power is measured for six different combinations - Random Access and Low Delay encoder configurations each with all three sizes of coding tree units. The core power is more or less constant due to our unified design.



(a) Logic utilization in kgates (total 715 kgate)



(b) SRAM utilization in kbits (total 1018 kbit)

Fig. 24 Logic and SRAM utilization for each processing engine.

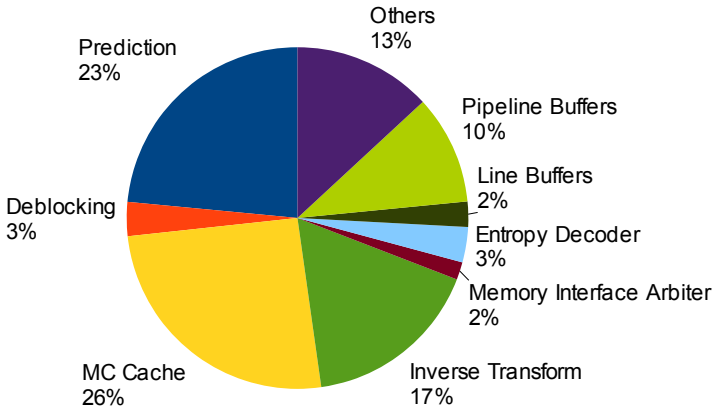


Fig. 25 Relative power consumption of processing engines and SRAMs from post-layout simulation with bi-prediction

10 Conclusion

This chapter presented the key challenges in implementing a hardware decoder for HEVC and techniques to address the challenges. The architecture of a test chip was described in detail. The test chip uses a variable-sized split system pipeline to process the wide range of Coding Tree Unit sizes and account for variable DRAM latency. The challenge of large and varied sizes of Transform Units can be addressed using Multiple Constant Multiplication and an SRAM-based transpose memory for an area-efficient implementation. Similarly, the use of Time-Multiplexed Multiple Constant Multiplication to optimize HEVC's longer interpolation filter was described. The longer interpolation filter also results in increased bandwidth requirement from reference picture buffer which is addressed by a cache and a DRAM-latency aware memory mapping. The design of a hierarchical memory organization was described to handle the pixel flow for intra-prediction and the main considera-

tions for designing HEVC's in-loop filters were enumerated. Finally, simulated and measured power results for the test chip were shown.

References

1. J. Vanne, M. Viitanen, T.D. Hamalainen, and A. Hallapuro. Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs. *IEEE Trans. Circuits Syst. Video Technol.*, 22(12):1885–1898, 2012.
2. Dajiang Zhou, Jinjia Zhou, Jiayi Zhu, Peilin Liu, and S. Goto. A 2Gpixel/s H.264/AVC HP/MVC video decoder chip for super hi-vision and 3DTV/FTV applications. In *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, pages 224–226, 2012.
3. Chang-Hung Tsai, Hsiuan-Ting Wang, Chia-Lin Liu, Yao Li, and Chen-Yi Lee. A 446.6k-gates 0.55-1.2v h.265/hevc decoder for next generation video applications. In *Solid-State Circuits Conference (A-SSCC), 2013 IEEE Asian*, pages 305–308, Nov 2013.
4. Chao-Tsung Huang, M. Tikekar, C. Juvekar, V. Sze, and A. Chandrakasan. A 249Mpixel/s HEVC video-decoder chip for Quad Full HD applications. In *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pages 162–163, 2013.
5. D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. *IEEE Trans. Circuits Syst. Video Technol.*, 13(7):620– 636, July 2003.
6. V. Sze and M. Budagavi. High throughput CABAC entropy coding in HEVC. *IEEE Trans. Circuits Syst. Video Technol.*, 22(12):1778–1791, 2012.
7. Yongseok Yi and In-Cheol Park. High-Speed H.264/AVC CABAC Decoding. *IEEE Trans. Circuits Syst. Video Technol.*, 17(4):490 –494, April 2007.
8. Y. C. Yang and J. I. Guo. High-Throughput H.264/AVC High-Profile CABAC Decoder for HDTV Applications. *IEEE Trans. Circuits Syst. Video Technol.*, 19(9):1395 –1399, September 2009.
9. P.-C. Lin, T.-D. Chuang, and L.-G. Chen. A branch selection multi-symbol high throughput CABAC decoder architecture for H.264/AVC. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 365–368, 2009.
10. Peng Zhang, Don Xie, and Wen Gao. Variable-bin-rate CABAC engine for H.264/AVC high definition real-time decoding. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 17(3):417–426, 2009.
11. V. Sze. *Parallel Algorithms and Architectures for Low Power Video Decoding*. PhD thesis, Massachusetts Institute of Technology, 2010.
12. V. Sze and M. Budagavi. A comparison of CABAC throughput for HEVC/H.265 VS. AVC/H.264. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 165–170, 2013.
13. Kentaro Kawakami, Jun Takemura, Mitsuhiko Kuroda, Hiroshi Kawaguchi, and Masahiko Yoshimoto. A 50% Power Reduction in H.264/AVC HDTV Video Decoder LSI by Dynamic Voltage Scaling in Elastic Pipeline. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E89-A(12):3642–3651, 2006.
14. D.F. Finchelstein, V. Sze, and A.P. Chandrakasan. Multicore Processing and Efficient On-Chip Caching for H.264 and Future Video Decoders. *IEEE Trans. Circuits Syst. Video Technol.*, 19(11):1704–1713, November 2009.
15. C.C. Lin, J.I. Guo, H.C. Chang, Y.C. Yang, J.W. Chen, M.C. Tsai, and J.S. Wang. A 160kgate 4.5kB SRAM H.264 video decoder for HDTV applications. In *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, pages 1596–1605, 2006.
16. Daniel Frederic Finchelstein. *Low-power Techniques for Video Decoding*. Thesis, Massachusetts Institute of Technology, 2009.
17. Thucydides Xanthopoulos. *Low Power Data-Dependent Transform Video and Still Image Coding*. Thesis, Massachusetts Institute of Technology, 1999.

18. P. Tummeltshammer, J. C. Hoe, and M. Puschel. Time-multiplexed multiple-constant multiplication. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 26(9):1551–1563, 2007.
19. J.-W. Chen, C.-C. Lin, J.-I. Guo, and J.-S. Wang. Low complexity architecture design of H.264 predictive pixel compensator for HDTV application. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 932–935, May 2006.
20. Z. Guo, D. Zhou, and S. Goto. An optimized MC interpolation architecture for HEVC. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1117–1120, March 2012.
21. *Multiplexed Multiplier Block Generator*. Available: http://www.spiral.net/hardware/mmcm.html.
22. Micron. DDR3 SDRAM system-power calculator.
23. K. Xu and C.-S. Choy. A power-efficient and self-adaptive prediction engine for H.264/AVC decoding. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 16(3):302–313, March 2008.
24. X. He, D. Zhou, J. Zhou, and S. Goto. High Profile intra prediction architecture for H.264. In *IEEE International SoC Design Conference*, pages 57–60, November 2009.
25. C.-T. Huang, M. Tikekar, and A.P. Chandrakasan. Memory-hierarchical and mode-adaptive hevc intra prediction architecture for quad full hd video decoding. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1–1, 2013.
26. Jiayi Zhu, Dajiang Zhou, Gang He, and Satoshi Goto. A combined SAO and de-blocking filter architecture for HEVC video decoder. In *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pages 1967–1971, Sept 2013.
27. Dajiang Zhou, Jinjia Zhou, Xun He, Jiayi Zhu, Ji Kong, Peilin Liu, and S. Goto. A 530 Mpixels/s 4096x2160@60fps H.264/AVC High Profile video decoder chip. *IEEE J. Solid-State Circuits*, 46(4):777 –788, April 2011.
28. Tzu-Der Chuang, Pei-Kuei Tsung, Pin-Chih Lin, Lo-Mei Chang, Tsung-Chuan Ma, Yi-Hau Chen, and Liang-Gee Chen. A 59.5mW scalable/multi-view video decoder chip for Quad/3D full HDTV and video streaming applications. In *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, pages 330–331, 2010.
29. Vivienne Sze, Daniel F. Finchelstein, Mahmut E. Sinangil, and Anantha P. Chandrakasan. A 0.7-V 1.8-mW H.264/AVC 720p video decoder. *IEEE J. Solid-State Circuits*, 44(11):2943–2956, November 2009.
30. C. D Chien, C. C Lin, Y. H Shih, H. C Chen, C. J Huang, C. Y Yu, C. L Chen, C. H Cheng, and J. I Guo. A 252kgate/71mW multi-standard multi-channel video decoder for high definition video applications. In *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, page 282603, 2007.