Techniques to Improve Motion Compensation Performance of H264 Video Decoder using a Vector Processor

Vijaya Yajnanarayana , Raghavan Subramaniyan and Michael Schuette[†]

Motorola India Electronics Limited, Bangalore INDIA

[†]Motorola Labs, Schaumburg, IL, USA

Tel: +91-80-26017829, E-mail: vijaypy@motorola.com

Abstract— Motion Compensation for video decoding in standards like H.264 requires significant amount of computation. This is primarily because of H.264 six-tap FIR filtering for sub-sample computation. These algorithms typically take more than 50% of the computational time on a RISC processor like ARM. Novel algorithms proposed through this paper can be employed for systems which use vector processors as video decode accelerators to accelerate this process. The proposed algorithms are implemented on H264 video decode system with ARM9 host-processor and RSVP vector processor as an accelerator for key decode algorithms. By employing the proposed algorithms we were able to accelerate the motion compensation module by more than 4 times as compared to plain RISC implementation. This is achieved by efficiently vectorizing data on which FIR-filtering and reconstruction algorithm is operated on, together with optimal representation of FIR-filtering and reconstruction algorithm itself on vector processor.

I. INTRODUCTION

A. Vector Processor for multimedia computation

Vector Processors (VP) have traditionally been used in large scientific applications. But recently there is lot of research to use similar architectures for embedded applications. They are aimed at speeding up multimedia and other signal processing algorithms used in embedded applications. [1] [2]

Vector processor architectures offer several architectural advantages over other classical parallel processing architectures like SIMD, WW-SIMD and DSP architectures as explained in [3]. This paper presents various techniques for accelerating motion compensation. The success of our approach is demonstrated through its application to the Reconfigurable Streaming Vector Processor (RSVP) [3][4] developed by Motorola.

B. RSVP Vector Processor

RSVP is a low-power vector processor that can be used as co-processor or memory mapped device. Our embedded platform uses RSVP as a memory mapped device to host-processor ARM9. Programming the RSVP architecture consists of describing the input and output vectors and scalar

values for a particular computation, and describing the computation itself as a data-flow graph (DFG). [3]

Fig 1 shows the programmer-visible data description registers in the RSVP architecture. Vectors to be processed reside in memory, and are handled by vector stream units (VSU) in the RSVP architecture. Their description consists of a pointer to the first element in each vector and a description of the vector shape. The shape of the vector consists of three scalar values: stride, span, and skip. Stride describes the spacing between each fetched/stored element (inclusive of the

Fig 3: Block Partitions in H.264

element). Span describes how many elements to fetch/store at stride spacing before applying the second level skip offset. Computations on the vectors are defined using a DFG.

For a more detailed description of RSVP including architectural concepts such as vector shape description, tunneling and asynchronous scheduling refer to [3][4][6]. How architectural features offered by RSVP vector processor can be exploited to accelerate the motion compensation will be detailed in the future sections.

C. Motion Compensation in H.264

Like its predecessor video standards like MPEG-1, MPEG-2, H.263 and MPEG-4, H.264 uses block based motion compensation. Reference frame is compensated based on motion vectors encoded in the bit stream (Refer Fig. 2). Depending on the motion vector (MV) resolution of the macro-block (MB) partition sub-sample interpolation of the MB is required, which is achieved by a six-tap FIR filtering [5]

An important difference compared to the previous standards includes the support for variable block size partitioning (refer Fig 3) and finer motion vector resolution.

Motion compensation in video processing is broadly a two step process involving motion vector calculation and compensating the reference frame using the motion vectors and residual data. To calculate the motion vectors in most cases, decoder computes a predicted motion vector MV_p using a predictive algorithm and adds it to the difference motion vector MV_d as shown in the equation (1).

$$MV = MV_p + MV_d \quad (1)$$

Though Motion Vector calculation is a critical step in the motion compensation, the prediction algorithm itself is not amenable to vector processing since it predominantly involves control logic. Further, the prediction algorithms like median prediction etc, employed in H264 standard are not computationally intensive operation in motion compensation. Hence motion vector computation is not considered for vector processing.

Motion compensation needs to be done for each block partition based on the calculated motion vector. H.264 partition sizes range from 16X16 to 4X4 for luma as shown in Fig 3. The motion vectors for each partition are of quarter-sample- resolution. For luma the sub-pixel samples at half-pixel positions are computed using a six-tap FIR filter and samples at quarter pixel positions are computed by bilinear interpolation between half and integer pixel positions. These

algorithms are computationally intensive and significant performance improvements can be achieved by accelerating them using vector processor.

II. GENERAL CHALLENGES AND METHODS FOR MOTION COMPENSATION

Data being processed in the motion compensation is streaming in nature that is data is produced and acquired as a stream of elements, each of which is relevant for a short period of time as it is block based processing. Also each vector elements (partition) undergoes the same computation or set of computations involving FIR filtering and reconstruction. When these computations are complete for all the partitions, the result is a reconstructed frame. Macro-block (MB) partition data from the reference frame that fed into this calculation are not used again. Vector processor architectures can take advantage of the above data characteristics for efficiently utilizing its computational units for optimal performance.

Each processing call to the RSVP operates on data which is called input vector and generates an output vector [3]. It is desirable to have longer vectors to improve the performance. The advantages of longer vectors are:

- Fewer calls to RSVP from the host processor. This results in less setup time and lower context switching overhead.
- Reduced startup time - This reduces the amount of time consumed to fill the RSVP pipeline.
- Improved opportunities for host-processor and RSVP execution overlap. If the host-processor is capable of performing useful work while the RSVP is busy (asynchronous) [6], longer vectors for RSVP will result in larger chunks of time being available to the host to do useful work and should result in a more efficient use of that time. The challenges to operate on longer vectors on RSVP include
- Ability to define operations that are to be performed on the vectors.
- Ability to define the shape, location and size of vector elements.
- Ability to repeat the operations defined number of times.

The addressing of the vector (shape of the vector) is an important issue since arbitrary addressing can't be followed with RSVP. Vectors are characterized by skip, span and stride [3][4]. The data elements being processed need to be arranged in a manner that can be described using these parameters.

RSVP also supports asynchronous operation which allows the execution of the code on RSVP in parallel with the host processor [6]. The software architecture needs to be tailored to take maximum advantage of this parallelism by having a good balance between host processing and RSVP processing throughout the processing chain. How these constructs are exploited from the motion compensation perspective is discussed in section IV.

Depending on the motion vector (MV) resolution sub-sample interpolation of reference frame data is required,

which is achieved by a six-tap FIR filtering [5]. The key challenge to accelerate these interpolation and reconstruction algorithms on vector processor involves

- Reference frame which acts as a input vector to the interpolation need to be vectorized efficiently.
- Representing the FIR-filtering (Interpolation) and reconstruction logic on RSVP vector processor efficiently.

Motion vectors are encoded in bit stream for each partition, one technique to increase the vector length for FIR filtering is to decode the motion vectors for the entire frame first and adjust the shape of the vector operated by the filtering algorithm in such a way as to address all the partitions having same motion vector. Fig 4 Part A shows the 3 partitions having the same MV's thus the interpolation algorithm for the sub-sample computation (from reference frame) for these three partitions will be same. However to address these partitions to operate on the same DFG having the interpolation and reconstruction algorithm, complex vector addressing schemes are required as the partitions with similar

motion vectors can occur at arbitrary positions in the reference frame (Refer Fig 4 Part A). Simple vector addressing mechanisms provided by RSVP like span, skip and stride are not sufficient to implement this technique. However if the neighboring partitions are likely to have same MVs and partition-types, then operating vector lengths can be increased by grouping them in to bigger blocks which can be easily addressed by RSVP (Refer Fig 4 Part B). Low motion sequence performance can be significantly enhanced using this. In high motion video sequence, motion vectors are highly un-correlated thus this technique may not yield much. Section III describes the gain achieved in RSVP on low motion video by grouping of macro-blocks.

Representing FIR-filtering and reconstruction algorithms on vector processor involves writing efficient filter Data Flow Graphs (DFG) as RSVP programs use DFG language to express the vector operations in a machine independent manner. All dependencies are explicitly stated which facilitates scheduling for parallel execution. The good DFG should execute in minimum instructions cycles with maximum parallelism. A simple DFG graph for a 6-TAP FIR filter is shown in Fig 5. This can be used as an interpolation filter for sub sample computation (Referred as TYPE-1 Algorithm).

Dedicated hardware and software constructs provided by the vector processors can be used to further optimize the filter design. In RSVP a mechanism called tunneling is provided. Using tunnels the limited data dependencies between iteration to iteration can be expressed. The tunnel nodes saves the result of an operation in current iteration while providing the result produced in the previous iteration (ie., the source and the sink of the data-flow is same) [3][4]. These tunnel

```

vname      6TAP_FIR_FILTER
vbegin     YEnd-Yegin,0
YBegin:
vinner
Y1: vcountinner
Y2: vconst 5
Y3: vconst -5
Y4: vconst 20
Y5: vld     (v1)
Y6: vmul    Y3, Y5
Y7: vmul    Y4, Y5
Y8: vtunnel Y5, z0
Y9: vadd    Y8, Y6
Y10: vtunnel Y9, z1
Y11: vadd   Y10, Y7
Y12: vtunnel Y11, z2
Y13: vadd   Y12, Y7
Y14: vtunnel Y13, z3
Y15: vadd   Y14, Y6
Y16: vtunnel Y15, z4
Y17: vadd   Y16, Y5
Y18: vsrnd  Y17, Y2 // div by 32 and round
Y19: vsat   Y18,u8
Y20: vge Y1, Y2 //Ignore first 5 samples
Y21: vstc.u8 Y20, Y19, (v0)
vpost
YEnd:
Vend

```

Fig 6: DFG for Type-2 FIR Interpolation Algorithm using tunneling

registers, provide a way to handle recurrence relations efficiently. The operation on this register retrieves the current tunnel value while injecting a new value into the tunnel register. In essence performing get/set operation in one instruction and enables creation of compact schedules. The value retrieved during iteration N of the DFG is that injected during iteration $N-1$. The value injected during iteration N will be retrieved during iteration $N+1$. [Fig – 6]. Delay-shift-accumulate logic required for the FIR Filtering can be efficiently designed using tunnel registers. Fig - 6 shows more efficient graph using the tunnels (Referred as TYPE-2 algorithm).

On an average performance gain of 2 cycle/pixel gain is achieved by using the Type-2 algorithm.

Reconstruction logic which involves adding the residual with the motion compensated frame can also be added to filter DFG itself to further optimize the performance. The similar procedures can also be extended to 2 dimensional filtering and reconstruction requirements.

Compensation for chroma component requires 1/8 resolution which is achieved by linear interpolation. This too is computationally intensive operation and can be targeted for optimization using the dedicated vector processor. The techniques discussed for luma can also be extended for chroma calculations.

III. METHOD FOR MOTION COMPENSATION OF LOW MOTION VIDEO.

The performance of the motion compensation algorithms on vector processor can be significantly enhanced by operating on larger vectors as discussed earlier.

In low motion video sequence like news reader etc, you will notice that the properties of motion compensation blocks such as MB-type (Macro-Block type), partition-size (block-size) and motion-vectors are highly correlated so that it can be grouped in to larger blocks. Applying filtering and reconstruction on these larger blocks yields better result compared to applying the same filtering algorithm on smaller partitions.

In a high motion sequence the macro-blocks can be rarely grouped as the block properties such as MB-type, partition-size and motion vectors are highly uncorrelated. The gain you would obtain by grouping is offset by the overhead of grouping, thus these techniques does not fare well for high motion video.

Fig - 7, shows a P-frame from a low motion CIF akio video sequence. The figure has 2-Arrays, Array-1 indicate the macro-block and its type. Array-2 indicates the range to which they can be grouped, Example “..... 22” means you can group 22 consecutive macro block partitions before applying motion compensation on them. Using this approach, notice that for the frame shown in Fig 7, the number of calls to RSVP gets reduced by 80.3%. Average reduction in vector processor call per frame computed over 150 frames of Akio CIF sequence is 56%. Performance incentive achieved by this approach varies across vector processors and will depend on the hardware architecture and tools. In the experiment grouping is done only in the horizontal direction, however one can also group the partitions in both direction to increase the vector sizes further, and their by achieving further performance gain.

On RSVP for a low motion akio CIF video sequence a performance gain of 1.5cycle/pixel is achieved using this method.

IV. METHOD FOR MOTION COMPENSATION TO EXPLOIT PROCESSOR CONCURRENCY

RSVP can also operate asynchronously with the host as a loosely coupled accelerator. Significant performance gain can be achieved by concurrent execution (asynchronous) of the RSVP and ARM host [6]. Host can synchronize with the RSVP by calling `_vdone()` function call, this call blocks until RSVP finishes. Have a look at the simple RSVP code fragment shown in Fig 8 and Fig 9.

In Fig-8, notice that when `rsvp_function` makes a synchronous RSVP call(`_vloop()`). The total time for the Host_function execution T_{syn} is given by

$$T_{syn} = T_1 + T_2 + T_3 \text{ where}$$

T_1 is the cycle time for instructions in the Host_function

T_2 is the cycle time for instructions in the Rsvp_function

T_3 is the cycle time for the RSVP graph execution on RSVP-co-processor (`vloop Call`). In Fig-9, notice that when

```

Frame#:1
1=Skip-MB with Partision Size 16X16
2=P-MB with Partision Size 16X16
0=MB with Partision Size other than above two
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1
1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 2 1 1
1 1 1 1 1 1 1 1 1 1 0 2 2 1 1 1 1 1 1 1 1 1
1 1 1 1 2 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 0 2 0 2 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0 2 1 1 2 1 1 1 1 1 1 1 1
1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1

. . . . . 22
. . . . . 7
. . . . . 11
. . . . . 9
. . . . . 2
. . . . . 9
. . . . . 9
. . . . . 13
. . . . . 8
. . . . . 19
. . . . . 6
. . . . . 22
. . . . . 17
. . . . . 22
. . . . . 2
. . . . . 22
. . . . . 2

Red in #of call = 80.808083

```

Fig 7: A p-frame from Akio cif sequence

rsvp_function makes a asynchronous RSVP call (`_valoop()`)
The total time for the Host_function execution T_{async} is given by

$$T_{async} = T_1 + T_2 + T_d$$

T_1 is the cycle time for instructions in the Host_function

T_2 is the cycle time for instructions in the Rsvp_function

T_d is the time duration where the host was blocked in `_vdone()` call, waiting for the RSVP to finish. Unlike `vloop()`, `valoop` return immediately so that host can continue with other independent tasks, while RSVP is busy processing the DFG on the input vectors.

The efficient design should involve careful planning and code organization so that there is no immediate dependency on the RSVP processed data, and the host cycles should be efficiently utilized while RSVP is busy. Lesser the blocking duration of the `_vdone()` call, better will be the design.

Typical H.264 video decoding involves bit stream parsing, variable length decoding (VLD), IDCT, and motion compensation done sequentially on all macro blocks. Out of all these, motion compensation is time consuming operation and is accelerated using RSVP. While RSVP is busy performing the motion compensation; It is possible to do other operations like bit stream parsing, VLD, IDCT etc on the future macro blocks, because for most cases, results of the

<pre> Host_function() { rsvp_function() ... } </pre>	<pre> Rsvp_function () { vloop() //sync RSVP call ... } </pre>
--	--

Fig 8: Synchronous RSVP Call from ARM Host

<pre> Host_function() { rsvp_function() _vdone() } </pre>	<pre> Rsvp_function () { valoop() //Async RSVP call ... } </pre>
---	--

Fig 9: Asynchronous RSVP Call from ARM Host

Fig 10: A circular Q having 3 elements

Fig 11 : Histogram of cycles taken by various interpolation and reconstruction algorithm and frequency of occurrence (Taken for foreman CIF 512Kbps sequence)

motion compensation of previous MBs(macro-block) are not needed.

Programming approach proposed to optimally utilize the host cycles while RSVP is busy is to design a circular service queue (Q); Each queue element contains a pointer to a motion compensation (interpolation & reconstruction) RSVP function, which will call RSVP loop asynchronously and a generic data structure encapsulating the arguments required for this

		ARM-9 Only				ARM-9 + RSVP				
		CPU Load				CPU Load				
		Total		Motion Comp		Total		Motion Comp		Overall Cycle/Pixel for Motion Comp
Format	sequence	CPU Load	Cycle/Pixel	% of total cycles for Motion Comp	Cycle/Pixel	CPU Load	Cycle/Pixel	% of total cycles for Motion Comp	Asynchronous Gain (cyc/pix)	
CIF @ 15fps / 192Kbps										
	Akiyo (Low Motion)	112.9 MHz	74.3	27.3%	20.3	91.7 MHz	60.3	14.1%	2.2	6.3
	Foreman (Medium Motion)	178.5 MHz	117.4	44.5%	52.2	117.9 MHz	77.5	18.1%	1.7	12.4
	Mobile (High Motion)	174.3 MHz	114.6	51.4%	58.9	104.1 MHz	68.5	20.6%	1.4	12.7
	Stefan (High Motion)	167.3 MHz	110.1	42.6%	46.9	111.8 MHz	73.5	17.2%	2.3	10.4
CIF @ 30fps / 256Kbps										
	Akiyo (Low Motion)	201.3 MHz	66.2	27.0%	17.9	161.5 MHz	53.1	13.2%	2.2	4.8
	Foreman (Medium Motion)	321.0 MHz	105.5	48.1%	50.8	199.3 MHz	65.5	18.8%	1.6	10.7
	Mobile (High Motion)	303.0 MHz	99.6	51.0%	50.8	177.1 MHz	58.2	18.9%	1.7	9.3
	Stefan (High Motion)	294.6 MHz	96.9	45.7%	44.3	188.4 MHz	61.9	18.2%	1.9	9.4

Table 1: Results after applying the algorithms discussed above for various video sequences.

function. Simple circular Q as shown in the Fig-10 can be employed for this purpose.

Rather than calling the RSVP functions for filtering/reconstruction for the MV directly for motion compensation; the decoder will enqueue the filter/reconstruction function pointer in to the queue. The histogram of the various motion compensation calls and its cycle duration is noted to compute the mean service rate of the queue this is shown in Fig 11. We discovered that with arm host and RSVP as the decode accelerator mean cycle time for RSVP motion compensation call is 953 cycles with a standard deviation of about 300 cycles. So the service rate of the queue is chosen as 950 cycles.

Notice that the queue acts as a repository for RSVP motion compensation calls to execute at later time, while host is busy doing bitstream-parsing VLD, IDCT etc for the future macro-blocks The dequeue calls are placed in these algorithms such that the service rate of the queue is 950 cycles.

If the previous macro-block (MB) motion compensation results are required for processing current MB OR when a logical end of processing is reached, (Example if an Intra MB is encountered), OR before starting a new frame, The decoder should flush the Q-module to finish all the pending RSVP calls synchronously.

By using the asynchronous operations on an average 2 cycle/ pixel gain is obtained.

V. RESULTS AND CONCLUSIONS

The simulations are done using an RSVP+ARM cycle accurate simulator that was developed by Motorola labs. For application benchmarking, we ran code with an SDRAM memory system (12-2-2-2 core-cycle burst access, with 4kB instruction and data caches for ARM9 and the 8kB data buffer for the RSVP processor). Table 1 shows the performances using the proposed algorithms for standard video sequences. Notice an average of 4X gain in the motion compensation.

Accelerating the motion compensation using any vector processor involves efficiently vectorizing the reference data, performing the interpolation and reconstruction of reference frame data efficiently. The techniques proposed in this paper on how to organize the reference frame data so that the vector processor is operated with large vectors, How to utilize the processor constructs to efficiently define the interpolation algorithm and how the proposed program model can be used to exploit the concurrency of host and co-processor (accelerator) can in general be used in any embedded system using vector processor as accelerator. The proposed algorithms are demonstrated through its application to the Reconfigurable Streaming Vector Processor (RSVP). From the results discussed above it is clear that significant performance improvement can be achieved.

REFERENCES

- [1] Lee, C.G. Stoodley, M.G. "Simple VectorMicroprocessors for Multimedia Applications" Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture, (MICRO-31 1998), Pages 25-36.
- [2] Christoforos E. Kozyrakis and David A. Patterson. "A New Direction for Computer Architecture Research." IEEE Computer.,May 1998, Pages 24-32.
- [3] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi, " The Reconfigurable Streaming Vector Processor (RSVP)", Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03), Pages141-151.
- [4] Chiricescu, S.; Schuette, M.; Essick, R.; Lucas, B.; May, P.; Moat, K.; Norris, J.; "RSVP: an automotive vector processor" Intelligent Vehicles Symposium, 2004 IEEE 14-17 June 2004, Pages 200 – 205.
- [5] "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC)", JVT-G050, Pattaya, Thailand, 7-14 March, 2003.
- [6] Chiricescu, S.; Chai, S.; Moat, K.; Lucas, B.; May, P.; Norm, J.; Essick, R.; Schuette, M. "RSVP II: a next generation automotive vector processor ", Intelligent Vehicles Symposium, 2005. Proceedings. IEEE 6-8 June 2005, Pages 563 – 568.