A Streaming Implementation of Transform and Quantization in H.264

Haiyan Li, Chunyuan Zhang, Li Li, and Ming Pang

School of Computer Science, National University of Defense Technology,
Changsha, Hunan, P. R. China, 410073
hy_lee@163.com

Abstract. The H.264 video coding standard uses a 4*4 multiply-free integer transform, minimizing computational complexity. The emerging programmable stream architecture provides a powerful mechanism to achieve high performance in media processing and signal processing. This paper analyzes the algorithm characteristics of transform and quantization in H.264 and presents a streaming implementation of transform and quantization on Imagine stream processor. We evaluate our implementation on a cycle-accurate simulator of Imagine and demonstrate stream processing efficiency by comparing its performance against other implementations. Experimental results show that our streaming implementation deals with transform and quantization of a 4*4 block in 6.875ns. The coding efficiency can satisfy the real-time requirement of current video applications.

1 Introduction

H.264 [1], proposed by Joint Video Team (JVT), is a new digital video coding standard. Some highlighted features are applied in H.264 for improved coding efficiency. Small block-size integer transform is one of the enhanced techniques to avoid inverse transform mismatch problem. It uses a 4*4 transform block size to somewhat reduce the block artifacts. All operations in transform process can be carried out in integer arithmetic only requiring additions and shifts. While a scaling multiplication is integrated into the following quantizer to decrease the total number of multiplications. By short tables, a set of new scalar quantization formulas use multiplications but avoid divisions [2].

Transform and quantization is a computationally-intensive component. H.264 adopts block-based motion prediction, so residual difference between current frame and predicted frame is organized in block. Each block is independent of others, exposing a great deal of data parallelism. The issue of optimizing transform and quantization in H.264 has been addressed in various research domains. For example, enhanced SIMD technologies such as MMX and SSE2 are used to improve coding rate of H.264 [3,4]. DSP acceleration is always the favor of product researchers [5,6]. In addition, special-purpose hardware implementations for transform and quantization in H.264 have emerged in succession [7,8,9,10].However, a new exploration on stream processing for H.264 has still remained.

M. Gerndt and D. Kranzlmüller (Eds.): HPCC 2006, LNCS 4208, pp. 41 – 50, 2006.

In this paper we develop a streaming implementation of transform and quantization in H.264 on the programmable Imagine stream processor [11]. Two computational kernels, corresponding to transform and quantization respectively, are constructed to operate on large homogeneous stream elements. In the end, we evaluate its performance by comparing it against other implementations. Experimental results show that the streaming transform and quantization implementation deals with the transform and quantization of a 4*4 block in 6.875ns, namely processing 145.5 millions of inputs per second. The coding efficiency can satisfy the real-time require-ment of current video applications.

Implementing on stream processor requires us to modify algorithms on the basis of stream processing and to arrange data in a streaming sequence in order to efficiently utilize the SIMD manner of stream architecture. Exploiting data parallelism and locality are encouraging for high performance applications.

The remainder of this paper is organized as follows. In Section 2, the principle and the algorithm of transform and quantization in H.264 are given. Section 3 describes the details of Imagine stream architecture. In Section 4, we discuss a streaming implementation of transform and quantization. Our experimental results and discussions are presented in Section 5. Finally, the paper is summarized in Section 6.

2 Transform and Quantization of H.264

The 4*4 transform adopted in the H.264 standard is an integer orthogonal computation [12]. This allows for bit-exact implementation for all encoders and decoders. Another important feature in the new standard is the removal of computationally-expensive multiplication that appears in the conventional standards.

The 4*4 integer transform of an input array X is shown in Equation (1).

$$W = C_f X C_f^T \tag{1}$$

where the matrix C_f is given by Equation (2).

$$C_f = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \tag{2}$$

In Equation (2), the absolute values of all the coefficients of the C_f matrix are either 1 or 2. Thus, the transform operation presented by Equation (1) can be computed using signed additions and left-shifts only to avoid expensive multiplications. While a scaling multiplication is integrated into the following quantizer to minimize the total number of multiplications.

H.264 uses a scalar quantizer. The post-scaling and quantization formulas are shown in Equation (3) and (4).

$$qbits = 15 + (QP \quad DIV \quad 6) \tag{3}$$

$$Z_{ij} = round\left(W_{ij}\frac{MF}{2^{qbits}}\right) \tag{4}$$

where QP is a quantization parameter. It can take any integer value from 0 up to 51. The wide range of 52 quantizer step sizes makes it possible for an encoder to accurately and flexibly control the trade-off between bit rate and image quality. Z_{ij} is a quantized coefficient. MF is a multiplication factor that depends on QP and the position (i, j) of the element in the matrix, referred to [12].

Fast algorithms for traditional DCT are also suited for integer transform, such as butterfly transform [2]. It converts matrix multiply into matrix-vector multiply, ensuring that there is no dependency between different vector columns in a matrix. And its algorithm structure is relatively simple (see Fig.1), regarded as a good algorithm for hardware implementation.

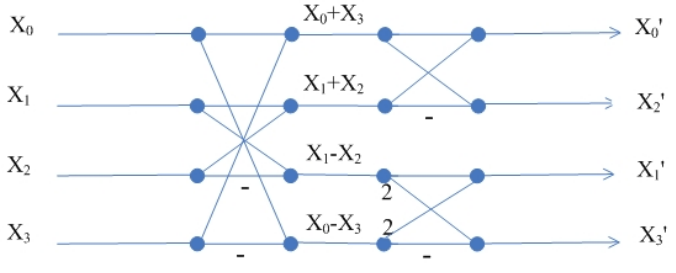

Fig. 1. Butterfly transform, where x_n (n=0..3) is a column of encoded matrix, and x_n' (n=0..3) is its corresponding filtered results. No multiplications are needed, only additions and shifts.

3 Imagine Stream Architecture

Imagine [13] can be considered representative of stream architecture (see Fig.2). We have done thorough research on Imagine [14], and proposed a new MASA [15] supporting multiple execution morphs. Imagine is programmable and flexible since it directly maps applications into streams and kernels. We have mapped many applications, such as fluid computation [15], Reed-solomon decoder [16], and motion estimation of H.264 [17] and so on. Imagine can provide high performance in so many domains including media processing and signal processing. For example, Imagine is able to sustain performance of 15.35 giga operations per second (GOPS) in MPEG-2 encoding application, corresponding to 287 frames per second (fps) on a 320*288-pixel, 24-bit color image [18].

In fact, the implementation of stream application is casted by a collection of *streams* passing through a series of computational *kernels*. A kernel is a small program executed in arithmetic clusters that is repeated for each successive element of its

input streams to produce output stream for the next kernel in the application. Streams are ordered finite-length sequences of data records. Each record in a stream is a set of related data elements of a single arbitrary data type. The semantics of applying a kernel to a stream are completely parallel, so the computation of a kernel can be performed on different independent elements in the input stream(s) in parallel. Kernel reads its inputs from *stream register file* (SRF). During computation, all temporary data are stored in the *local register file* (LRF) of each cluster. And the output stream of a kernel are sent back to SRF. Only the initial and final data streams need to be transferred through streaming memory to the off-chip SDRAM. This three level memory hierarchy is able to meet the large instruction and data bandwidth demands of computationally intensive applications well.

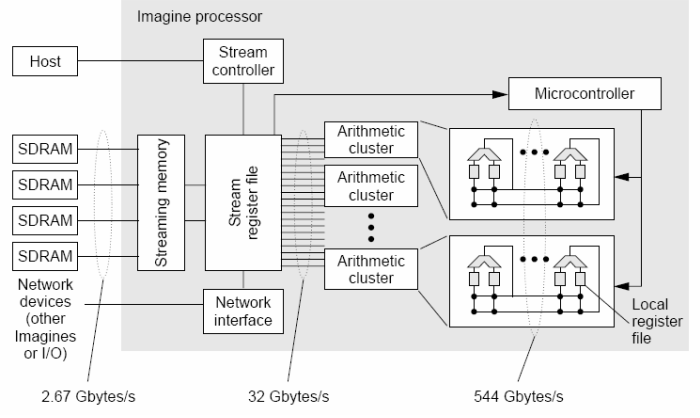

Fig. 2. Imagine stream architecture. The three-level memory hierarchy is shown in the figure.

4 Implementation

4.1 Characteristics Analysis

Stream programming model can match the requirement of media processing very well. Before mapping transform and quantization of H.264 video encoder on Imagine stream processor, it is necessary to analyze the inherent characteristics especially in computation intensity, parallelism and locality.

Computation Intensity

Integer transform is a computationally intensive module like block searching in motion estimation except for extra decision-making. If using butterfly algorithm mentioned in Section 2, a transform needs 64 additions and 16 shifts. As a result, 12.16 million additions and 3.04 million shifts are executed in one second for a CIF image of 352*288 pixels at 30fps.

Parallelism

Large data parallelism exists in transform process. Based on 8-cluster structure of Imagine, different columns of encoded matrix can be performed in parallel. Efficient data organization may help magnify the advantage of data-level parallelism. Besides, transform has obvious instruction-level parallelism. The pure additions and shifts can be packed into VLIW compactly. An additional level of task parallelism can be discerned from the pipelining of kernels.

Locality

Video coding is processed orderly frame by frame and block by block, like a stream of data flowing through every sequential processing module. Kernels encapsulate short-term kernel locality, and allow efficient use of the LRFs. For example, the intermediate results of butterfly transform can be stored in the inner LRFs. At the same time, stream capture long-term producer-consumer locality in the transfer of data form one kernel (e.g. *transform* kernel) to another kernel (e.g. *quantization* kernel) through the SRF without requiring costly memory operations, as well as spatial locality by the nature of stream being a series of data records.

4.2 Implementation

Programming model of Imagine includes two levels: stream level (in StreamC) and kernel level (in KernelC) [19]. According to the relationship of input and output streams, the kernel diagram of transform and quantization is described as Fig.3.

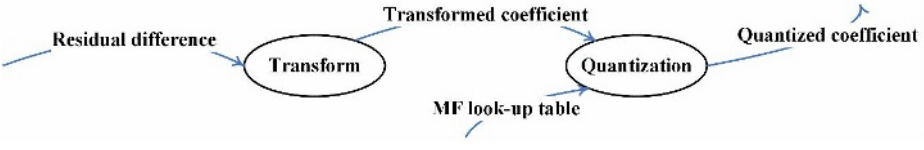

Fig. 3. Kernel diagram. An ellipse represents a computational kernel. And the connection line represents the input or output streams of the kernel.

These two kernels are chained together, where the output stream from the *transform* kernel is fed into the next *quantization* kernel as an input stream. Producer-consumer locality is exploited by consuming the result of one kernel as soon as it is produced. MF look-up table is organized as a constant stream and loaded with transformed coefficient stream into the *quantization* kernel. Note that a kernel can take more than one streams as its input, and the output may be one or more streams for different kernels.

Integer transform $Y=C_fXC_f^T$ performs matrix multiplications twice. Assume that $B=C_fX$, then $Y= C_fXC_f^T= BC_f^T= (C_fB^T)^T$. Based on transpose we keep the block that is to be transformed as right matrix while the left matrix is C_f. The matrix X can be divided into four vectors by column. Inter-column independence makes the butterfly algorithm suited for stream processing. The main loop in the *transform* kernel is illustrated in Fig.4.

```
For x in input residual difference stream:
    Call butterfly transform computation
    Transpose (communication operations)
    Recall butterfly transform computation
    Output transformed coefficient stream
```

Fig. 4. Pseudocode of *transform* kernel

Fig. 5. Stream distribution in eight clusters

The input stream of *transform* kernel consists of 4*4 matrix blocks. Fig.5 illustrates the distribution of each stream record in clusters, where x_{ij} represent one stream record with the relative position in its affiliated block. We choose a simple solution-replicating a 4*4 matrix twice. It brings half waste of cluster resources because the computation of four clusters is redundant. The better data records in the input stream are organized, the better performance stream architecture will get [20]. The way of organizing the data records is explicit for stream programmers. So it requires programmers to understand the algorithm characteristics in order to map it efficiently.

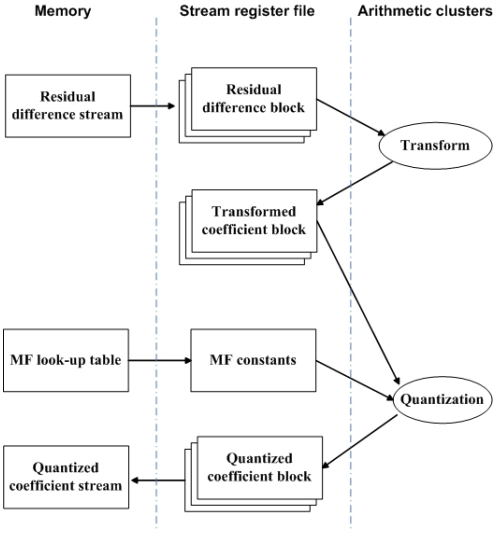

Fig. 6. Three-level memory hierarchy diagram

The *transform* kernel reads each data element from memory (In the whole encoder, the input of *transform* kernel is also fed by its prior kernel. So, it is reasonable to suppose the input stream exists in SRF not memory.). After kernel execution, the transformed coefficient stream produced by *transform* is sent to the following *quantization* kernel directly. MF is loaded by a series of communication operations and multiplies Y by index. Finally, the output quantized coefficient stream can be written back to memory or stored in SRF as an intermediate stream for other kernels. The three-level bandwidth hierarchy corresponds to the three columns of Fig.6.

5 Simulations and Results

5.1 Experimental Results

We run our streaming implementation of transform and quantization on ISim, a cycle-accurate simulator ISim (500MHz), which is provided by the Imagine Project of Stanford University [21]. The simulator can accurately model all aspects of stream processing and stream memory system. And the execution cycles obtained by ISim is convictive enough to evaluate the performance of a streaming implementation. The correctness of our implemented transform and quantization is also checked. This is done by passing different input sequences to our stream program and comparing the experimental result and the mathematical value.

Simulated results show that $3.485*10^5$ cycles is needed for a CIF image. Thus, dealing with the transform and quantization process of a 4*4 block requires 6.875ns, thereinto 5.79ns for integer transform. It means that our streaming implementation is able to process 145.5 millions of inputs per second. The performance may be optimized by some advanced techniques such as loop unrolling and software pipelining [20]. The processing rate is higher than that defined to HDTV video sequence (HDTV must process 124.5 millions of inputs per second [9]).

We compare our streaming implementation with other different improvements, mentioned in Section 1. Take the time of 4*4 integer transform as the criterion, shown in Fig.7. Obviously, Imagine obtains comparative performance with special-purpose hardware designed for transform and quantization application, much better than MMX and DSP improvements. However, Imagine is more flexible than special-purpose hardware. Thus, it has good adaptability and scalability.

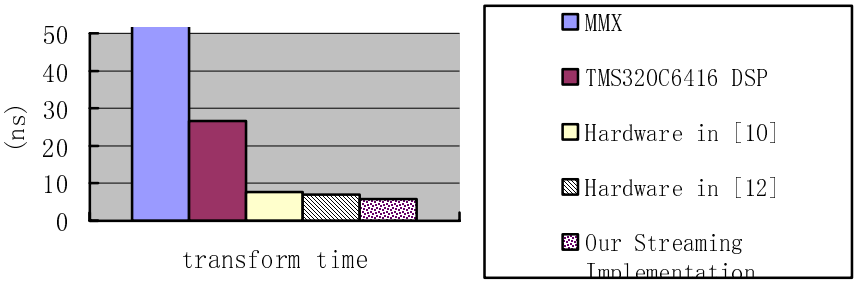

Fig. 7. Comparison among different implementations, where the data of processing time refer to [4,6,7,9] resprectively

Our streaming implementation has the shortest processing time of five cases in Fig.7. Imagine can achieve high performance for three reasons. First, stream computation is efficient when operated on homogeneous data elements. Stream processing mechanism ensures to overlap between kernel computation and memory access, hiding the latency of memory operations. Second, Imagine performs in the SIMD manner. Large data parallelism and little global data reuse may explore the powerful

computing capability of Imagine. Third, kernel locality and producer-consumer locality are captured in LRF and SRF of Imagine. The three-level memory hierarchy can afford the bandwidth requirement well.

5.2 Discussions

For an actual image, the residual difference input stream is a very large data set. Processing each element in a single computation is impractical because the size of data set may greatly exceed the size of on-chip storage. Instead, most Imagine applications use the common technique of *stripmining* [18]. In our implementation, residual difference pixel-blocks are divided into input batches, stream operations are applied to an entire input batch at a time. The size of a batch in our implementation is almost 14000 pixels. Fig.8 gives the utilization of SRF in the execution of transform and quantization.

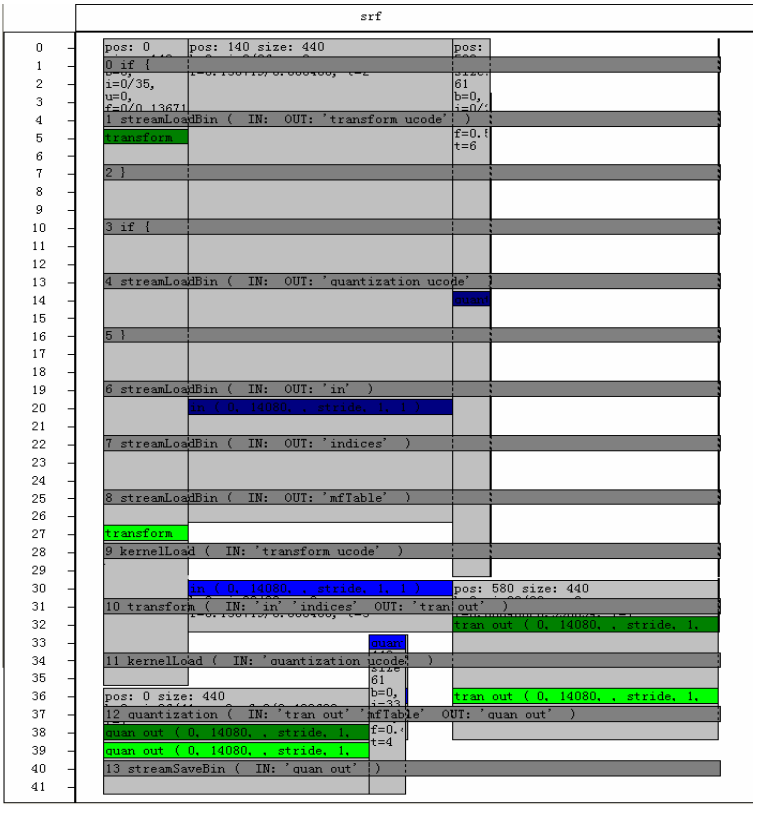

Fig. 8. SRF utilization, where the horizontal axis represents SRF size and colorful bars represent usage of SRF. Vertical axis represents stream execution time. The inputs in a batch are 14080 residual difference pixels in this figure. But blue bar indicates a read or write that requires a memory access when SRF is spilled over.

The utilization of functional units is given in Table 1. The result is matched with the algorithm characteristics: large amount of additions and shifts make great use of adders only expect for some stalls before initial operands are prepared. While multipliers are only used for quantization, so the utilization ratio is not very high.

By taking advantage of unique three-level memory hierarchy and large numbers of functional units, Imagine can achieve so high performance. Combined with the accelerated implementation of motion estimation on Imagine [17], we can infer that the whole H.264 encoder will get better performance and meet real-time requirement of current video applications.

Table 1. Arithmetic unit utilization

	Adders	Multipliers
Utilization ratio	83%	26%

6 Conclusion

In this paper we have presented a streaming implementation of transform and quantization in H.264. Experimental results show that processing transform and quantization for a 4*4 block needs 6.875ns. As a result, our implementation is able to process 145.5 millions of inputs per second. It can satisfy the real-time requirement of video applications. And it proves that the programming model including memory hierarchy of stream architecture is helpful for large numbers of data to repeat the same or similar operations. But some issues are still needed to be researched deeply, such as slice partition granularity and stream algorithm optimization. We will pay more attention to its further improvement.

Acknowledgements. We thank Imagine project group of Stanford University for providing the Imagine simulator. We also thank the reviewers for their insightful comments. This work was sponsored by National Natural Science Foundation of China under Grant 60573103.

References

1. JVT, Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264| ISO/IEC 14496-10 AVC). May 2003.
2. Henrique S. Malvar, Antti Hallapuro, Marta Karczewicz, Louis Kerofsky: Low-Complexity Transform and Quantization in H.264/AVC. IEEE Transactions on Circuits and Systems for Video Technology, Vol.13, No.7, July 2003.
3. Cui Yansong, Duan Dagao, Deng Zhongliang: The Analysis of Transform and Quantization in H.264. Modern Cable Transmission, 2004.5, pp 71-74
4. Wei Fang, Li Xueming: SIMD Optimization of Transform and Quantization in H.264. Computer Engineering and Applications, 2004.17, pp 24-27
5. Liu Baolan, Liu Guizhong, Su Rui: Implementation and Optimization of Pixel-Compression Module in H.264 Based on DSP System. Microelectronics, Vol. 22, 2005, No.6, pp200-205

6. Shen Haitao, Fan Yangyu, Wang Fengqin, Hao Chongyang: An Implementation of Transform Encoding on DSP in H.264. 2004
7. Liu Ling-zhi, Qiu Lin, Rong Meng-tian, Jiang Li: A 2-D Forward/Inverse Integer Transform Processor of H.264 Based on Highly-parallel Architecture. In Proceedings of the 4th IEEE International Workshop on System-on-chip for Real-Time Applications, 2004
8. Ihab Amer, Wael Badawy, and Graham Jullien: Hardware Prototyping for the H.264 4*4 Transformation. ICASSP 2004.
9. Roger Endrigo Carvalho Porto, Marcelo Schiavon Porto, Thaisa Leal da Silva, Leandro Zanetti Paiva da Rosa, Jose Luis Almada Guntzel, Luciano Volcan Agostini: An Integer 2-D DCT Architecture for H.264/AVC Video Coding Standard. XX SIM-South Symposium on Microeletronics.
10. Young-hun Lim, Yong-jin Jeong: Hardware Implementation of Integer Transform and Quantization for H.264. December 2003.
11. Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, Brucek Khailany: The Imagine Stream Processor. Appears in the Proceedings of the 2002 International Conference on Computer Design, September 2002.
12. "H.264/MPEG-4 Part 10: Transform&Quantization"www.vcodex.com
13. Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang: Imagine: Media Processing with Streams. IEEE Micro, March-April 2001.
14. Mei Wen, Nan Wu, Haiyan Li, Chunyuan Zhang: Research and Evaluation of Imagine Stream Architecture. Advances on Computer Architecture, ACA'04
15. Mei Wen, Nan Wu, Haiyan Li, Li Li, Chunyuan Zhang: Multiple-morghs Adaptive Stream Architecture. Journal of Computer Science and Technologgy, 2005
16. Mei Wen, Nan wu, Haiyan Li, Li Li, Chunyuan Zhang: A Parallel Reed-solomon Decoder on the Imagine Stream Processor. Second International symposium on Parallel and Distributed Processing and Applications, Hongkong, 2004.12
17. Haiyan Li, Mei Wen, Chunyuan Zhang, Nan Wu, Li Li, Changqing Xun: Accelerated Motion Estimation of H.264 on Imagine Stream Processor. International Conference on Image Analysis and Recognition 2005
18. John D. Owens, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Brian Towles, Ben Serebrin, William J. Dally: Media Processing Applications on the Imagine Stream Processor. In the Proceedings of the 2002 International Conference on Computer Design, 2002
19. Abhishek Das, Peter Mattson, Ujval Kapasi, John Owens, Scott Rixner, Nuwan Jayasena: Imagine Programming System User's Guide 2.0, June 2004
20. Haiyan Li, Chunyuan Zhang, Li Li, Ming Pang: Stream Algorithm of 4*4 Integer Transform. Conference on Virtual Reality and Vision 2006.
21. The Imagine Project, Stanford University, http://cva.stanford.edu/imagine/