Software Parallel CAVLC Encoder Based on Stream Processing

Ju Ren, Yi He, Wei Wu, Mei Wen, Nan Wu, Chunyuan Zhang
Computer School
National University of Defense Technology
Changsha, China
se7en.rj@gmail.com

Abstract—Real-time encoding of high-definition H.264 video is a challenge to current embedded programmable processors. Emerging stream processing methods supported by most GPUs and programmable processors provide a powerful mechanism to achieve surprising high performance in media/signal processing, which bring an opportunity to deal with this challenge. However, traditional serial CAVLC has highly input-dependent execution and precedence constraints, which becomes a bottleneck to implement H.264 encoder efficiently. This paper presents a software parallel CAVLC encoder based on stream processing. Many approaches are explored to solve the restrictions of parallelizing CAVLC caused by data dependency and branch/loop instructions. Experiment results show that our parallel CAVLC encoder on two stream processing platforms of STORM and GPU achieves 3.03x and 2.08x speedup over the original serial CAVLC respectively. Finally, the proposed parallel CAVLC encoder coupled with stream processor enables a real-time encoding of 1080p H.264 video.

Keywords- CAVLC; H.264; parallel; stream

I. INTRODUCTION

CAVLC (Context-adaptive variable-length coding) [1] is a typical entropy coding algorithm widely applied in H.264/MPEG-4 AVC video compression. It is used for coding quantized transform coefficients of the residual images. CAVLC makes a great contribution to high compression rate of H.264.

Along with the vigorous development of media applications such as HDTV and real-time video conference, video encoding requires higher performance embedded processor more and more strictly. Software CAVLC encoder running on embedded programmable processors (e.g. PDA), cannot meet the requirements of real-time performance of HDTV 1080p. Therefore, many dedicated ASIC CAVLC encoders have been developed for real-time video encoding [2][3][4]. However, dedicated ASIC design is time-consuming, expensive, and inflexible. So we attempt to develop a parallel method based on software design, in order to accelerate CAVLC encoding and acquire real-time 1080p H.264 codec performance. However, it is difficult to parallelize CAVLC due to its serial nature of context-based data dependency, branch and loop operations, and variable-length output bit-stream.

Stream processing [5], an emerging parallel processing model supported by most GPUs and embedded programmable processors (e.g. CELL in PS3), can exploit the application’s ILP, DLP, TLP efficiently. It achieves high performance in media/signal processing and scientific computation [6][7]. Previous studies have parallelized many core components of H.264 on stream processing model [8] with satisfactory results (8x8 DCT acquires 6.8x speedup). This may provide a feasible approach to parallelize the CAVLC on stream processing.

This paper presents a software parallel CAVLC encoder based on stream processing. At first, we analyze the characteristic and potential parallelism of CAVLC algorithm. Four restrictions in parallelizing CAVLC are context-based dependency on coefficients computing, context-based dependency on look-up table selection, input-dependent execution, and variable-length bit-stream outputting. To deal with the above restrictions, we adopt four corresponding methods: enlarging data processing granularity, parallel predicting, redundant calculating and parallel stream producing and assembling. Then our parallel CAVLC encoder is implemented on SPI STORM stream processor and NVIDIA Geforce 8800 GTX GPU. Experiment results show that our parallel CAVLC encoder gains 3.03x and 2.08x speedup on the two platforms. Furthermore, compared with the original serial CAVLC encoder executed on dual core x86 and TI C6416 DSP platform, the parallel CAVLC encoder executed on STORM gets 2x and 33x higher performance respectively. A conclusion is drawn that the parallel CAVLC encoder satisfies the real-time encoding requirement of 30 frames per second for 1080p HDTV on STORM.

The rest of this paper is organized as follows. Section II presents stream processing and its advantages on parallel. Potential and restrictions of parallelizing CAVLC are addressed in Section III. Section IV describes the novel parallel CAVLC encoder and approaches to eliminate restrictions of parallel. Evaluation results on different platforms and comparison with other methods are given in Section V. The paper is summarized in Section VI.

This work was supported in part by the China NSFC No. 60673148, 60703073, 863 Project No. 2007AA01Z286, and SRFDP No. 20069998025.
II. STREAM PROCESSING

Figure 1 illustrates the stream processing model. Different from traditional programming patterns, stream processing model decouples an application’s memory accesses and computation. Data movements are described by streams and computations are described by kernels. Stream, organized by operated data, is an ordered set of homomorphous records of an arbitrary data type. Streamgather, streamscatter, and computations in kernel are three operations on streams in stream processing model. Streamgather is used to gather data into stream, whereas streamscatter scatters stream into data. Kernel, operating on one or more streams, is an individual computation process. As shown in figure 1, application is decomposed by a collection of streams passing through a series of kernels in stream processing model.

Stream processing is widely supported by GPUs and Stream Processors, such as NVIDIA or ATI GPU, Imagine, Merrimac, STORM, CELL, TRIPS, and RAW. This paper selects NVIDIA GPU GeForce8800 GTX and SPI STORM as target stream processing platforms. Figure 2 is the abstract architecture of those two processors. The architecture detail of GPU and STORM is described in [9] and [10] respectively. In this paper, we name them stream processor. Stream processor is composed of PPU (parallel process unit) array with inter-PPU switch, PPU controller, shared on-chip memory, and off-chip memory system. During execution, streamgather gathers data from off-chip memory into streams in the on-chip memory. All PPUs fetch stream records from on-chip memory into local register files. And stream processor issues only one instruction of a kernel at a time for all PPUs. The output stream records are stored in the on-chip memory and then are scattered to off-chip memory. In the whole executing procedure, all operations in kernels are executed in parallel. Therefore, the architecture brings parallel into full play in stream processing, which is elaborated in the following multiple levels of parallelism.

1. Data-Level Parallelism is acquired by SIMD fashion. Each PPU processes one stream records in parallel, so multiple stream records are processed by the same instruction simultaneously in PPU array.

2. Since there are multiple Function Units in a PPU, Instruction-Level Parallelism can be exploited by executing multiple independent operations on a stream record. ILP is supported by VLIW in STORM and pipeline in GPU.

3. Task-level Parallelism can be exploited effectively by overlapping computation on PPUs and memory access between on-chip and off-chip memory system. We can reorder the kernel computation and streamgather/streamscatter without changing the dependency of these stream operations.

III. CAVLC PARALLELISM ANALYSIS

A. CAVLC Profile

This paper takes CAVLC in x264 [11] as reference code. Figure 3 illustrates the skeleton of the CAVLC encoder. It shows that CAVLC encoding is processed orderly frame by frame, slice by slice, and Macro-block (MB) by Macro-block, which is a typical serial procedure. In one MB, a maximum of 27 blocks, including 1 Luma DC block, 16 Luma AC blocks, 2 Chroma DC blocks, and 8 Chroma AC blocks, should be scanned in zigzag order firstly. Then all of the blocks are encoded sequentially in one MB. CAVLC encoding procedure, which is similarly in each block, is partitioned into two phases of reverse scanning phases and coefficients encoding phases. In the two phases, five coefficients listed in the right part of figure 3 should be calculated, encoded through variable-length coding look-up tables, and be output into bit-stream. When encoding a 1920x1080 frame, Block CAVLC_encode in reference code will execute more than 200000 times totally in serial mode according to figure 3.

In our previous studies, components of H.264 such as analyse, encode and filter have been parallelized based on stream processing [12]. Figure 4 shows the breakdown of different components in H.264 encoder executing on STORM processor before and after parallelizing. In figure 4(a), the whole H.264 encoder is executed in serial mode. Occupancy rate of CAVLC is 7%. In figure 4(b), the components in H.264 are executed in parallel mode except CAVLC still executed in serial mode. Occupancy rate of CAVLC is up to 34%, which proves it becomes a bottleneck hindering efficient implementation of H.264 encoder.
Before parallelizing CAVLC on stream processing model, it is necessary to analyze the inherent characteristics in parallelism. We found that large data parallelism exists in CAVLC, including block parallelism, MB parallelism, and other parallelisms.

Block Parallelism. Taking the Luma AC encoding for instance, CAVLC encoder processes one block of 16 luma AC blocks in a MB each time. The encoding processes in all 16 blocks are exactly the same. Hence, there is potential to have the blocks process in parallel.

MB Parallelism. Similar with the block parallelism, all MBs in a slice or in a frame have the same encoding processes. Slice parallelism and frame parallelism work as the way block parallelism does. But large scale parallel may cause pressures in bandwidth requirements. In addition, [8] provides pixel parallel for streaming transform coding of video compression. An appropriate parallel granularity is discussed in section IV. B.

C. Restrictions of Parallel CAVLC

Though CAVLC has large data parallelism, its serial nature leads to highly input-dependent execution and precedence constraints, which restrains CAVLC parallelizing. Various restrictions of parallel are discussed in the rest of this section.

1) Restriction of parallel calculations in kernel

a) Input-dependent executions

Branch and loop operations are frequently presented in CAVLC encoder, which restrain parallel on stream processing. There are two reasons responsible for it. First, stream processing works in SIMD manner, which forces multiple PPUs to execute the same instructions simultaneously. Second, control flow in CAVLC has input-dependency. For example, loop count of a code phase for coefficients calculating depends on the input residual data. Because input residual data have different characteristic in different PPU, the code phase may execute diverse times on different PPUs, which is against the SIMD rules. In the same way, encoding path may enter into distinct branches. Moreover, branch and loop operations are nesting in each other.

b) Context-based dependency.

There are two context-based dependencies in CAVLC encoder.

Context-based data dependency intra-block. In levels encoding phase of a block, the value of a parameter named suffix_Length is updated according to its current value and encoded levels. This data dependency only needs data inner a block, thus called context-based data dependency intra-block.

Context-based data dependency inter-blocks/inter-MBs. This dependency occurs when encoding nonzero coefficients. Four variable-length tables and one fixed-length table are available for nonzero coefficients encoding. Choosing a table among the five depends on a parameter called nC, an average number of nonzero coefficients of the neighboring left and upper blocks named nA and nB respectively. Thus, encoding a block depends on its neighboring left and upper blocks.
Fortunately, a process called predict_nC can eliminate data dependency inter-blocks. After zigzag ordering, predict_nC calculates nC parameters needed by each block in one MB and stores it in an array. Then CAVLC starts the nonzero coefficients encoding without any inter-blocks dependency in a MB by getting nC from the array. However, different MBs still cannot be executed in parallel because of data dependency inter-MBs. Figure 5 illustrates this dependency. It shows that gray blocks in MB121 depend on blocks in the left MB120 and upper MB1, which makes MBs in the figure cannot be executed in parallel.

2) Restriction of parallel stream producing and assembling

As shown in figure 3, five coefficients in a block are encoded and outputted into bit-stream one by one. Because each is encoded to unpredictable length of bits, different blocks have variable-length output bit-streams. Similarly, made up by assembling blocks’ output bit-streams, different MBs have variable-length output bit-streams.

Furthermore, each encoded MB is connected in final output stream one by one, so the MB’s output position in the bit-stream depend on its prior MB’s output position and length. It implies that although different MBs can be encoded in parallel, the output streams still have to be assembled in serial. Even worse is that bit-stream is connected bit by bit but not byte by byte. So, two neighboring MBs may connect their output stream within a byte. Figure 6 illustrates an instance of assembling three MB’s output bit-stream. They produce output bit-stream in different lengths (18 bit, 12 bit, and 14 bit respectively). As a result, bit-streams of MB0 and MB1 have to be connected in the third byte of final bit-stream, which proves that the MB1’s output stream must be produced after the MB0’s output stream.

IV. PARALLEL CAVLC ENCODER

The stream processing model of our proposed parallel CAVLC encoder is shown in Figure 7. CAVLC encoding in one MB is divided into several kernels: zigzag_scanning, coefficients_encoding, and VLC_packing. Among them coefficients_encoding is a kernel group, which is composed of 5 different kernels, ue_encoding, Luma_DC_encoding, Luma_AC_encoding, Chroma_DC_encoding and Chroma_AC_encoding. One or more functions in the CAVLC reference code may be merged into one kernel according to the dependency of these functions. The residual coefficients, VLC look-up tables are gathered into input streams which turn to be consumed whereas output streams are produced by kernels. In the end, bit-stream produced by the kernel VLC_packing is scattered onto off-chip memory. An efficient CAVLC encoder on this parallel model needs to overcome the restrictions mentioned in section III.C. Approaches to eliminate these restrictions are described in the sections IV.A and IV.B. At last, the implementation of the proposed CAVLC encoder is shown in IV.C.

A. Parallel Calculation in Kernels

To counter the restrictions of parallel calculation, we adopt three corresponding methods, which are enlarging data processing granularity, parallel predicting and redundant calculating.

1) Enlarging data processing granularity and parallel predicting
In order to execute CAVLC in parallel, an appropriate data processing granularity should be chosen first. Different processing granularity including pixel, block, MB and slice/frame are discussed as follows. Figure 8 shows the data distributing in pixel, block and MB parallelism with the assumption that the number of PPU is 16.

a) **Pixel parallelism**

In this manner, data processing granularity is one pixel in a PPU, and an iteration execution processes a block’s 16 pixels. Noting that the input data in CAVLC is residual data in fact, we call it pixel for convenience. The condition of paralleling in pixel granularity is that 16 pixel in a block must execute the same calculations. Unfortunately, CAVLC does not process a pixel in the same way but calculate statistics information by scanning each pixel in a block. Thus, pixel distributed in different PPU makes communications overhead increase in zigzag ordering and coefficients encoding to acquire pixel in other PPU. Furthermore, calculations in a block have to execute mainly in one PPU while other PPUs are left unused.

b) **Block parallelism**

Data processing granularity is one block in each PPU, and 16 PPUs process 16 blocks in one iteration. All residual data in a block are processed by the block’s host PPU, so data dependency intra-block can be eliminated. As shown in figure 3, luma ac encoding and chroma ac encoding process 16 and 8 blocks respectively, which enable block parallelism to execute effectively. But only one or two PPU needs to process luma dc and chroma dc encoding. In block parallel manner, data processing granularity is 256byte when encoding can be applied in 16 blocks of a MB.

c) **MB parallelism**

In this manner, data processing granularity is one MB in each PPU. However, context-based data dependency inter-MBs restrain different MBs from executing in parallel. Approach called parallel predicting is applied to eliminate the data dependency. It includes three steps. First, nonzero parameters of all blocks in a MB are calculated by the MB’s host PPU. Second, precedence MB sends nonzero parameters to constrain MB by executing communication operations. Third, each PPU calculates nC parameters of all blocks in its own MB. These three steps can execute in parallel because there is no data dependency between MBs in each step.

MB parallelism is superior to the Block parallelism in that:
1. DC encoding in different MBs could be processed in parallel.
2. By eliminating the data dependency inter-MBs, zigzag ordering and coefficients calculating phase can be parallelized on all MBs in the same slices. Certainly, PPU numbers decide parallel Macro Blocks number in one executing iteration, and on-chip memory decides stream length.

d) **Slice/Frame parallelism**

In the same way of parallelizing MB level processing, multiple slice/frames can be processed in parallel. But large scale processing granularity leads to long input residual data stream. For example, a 1920x1080 frame needs more than 2MB on-chip memory to store the input stream only.

This paper applied MB parallelism onto the CAVLC encoder according to its flexibility and stream processors hardware capacity.

2) **Redundant calculating**

Redundant calculating, which is shown in figure 9, is used to relax the restriction caused by branch and loop operations. In the figure, each PPU executes all different branches in sequential manner, and renames variables in order to save the results in different branches. After executing the last branch, each PPU chooses correct results from different branches according to the branch condition. Similarly, each PPU executes loop body up to the maximum loop counts. When a loop is scheduled to be finished according to the loop conditions before loop count achieve maximum number, PPU
Figure 10. Parallel output stream producing and assembling

early saves the results and continue running the rest iterations without writing values into the result variable. Redundant calculating enables each PPU to execute the same instructions simultaneously. On the other hand, the calculations pressure is increased by executing redundant branches or iterations. However, overhead brought by redundant calculating can be ignored because CAVLC encoding is not a computing-intensive process and arithmetic units are really abundant in PPU.

B. Parallel Stream producing and assembling

Restriction of parallel stream producing and assembling comes from precedence constraints of VLC packing. Variable-length output bit-stream leads to strict data dependency between precedence MB and constrain MB. Considering the instance in figure 6, three output bit-streams must be produced sequentially in byte 2 and byte 3. We present a method named parallel shifting for parallel stream assembling, which is shown in figure 10. There are 3 steps as follows.

Step 1. Encode MBs in parallel in PPUs. Each MB’s output bit stream is written into a byte-aligned stream space. Therefore, the beginning address of output stream in each MB is byte-aligned whereas the ending address is not. For example, the last byte of MB0’s output stream has only 2 bits. In this phase, the length (expressed in bit but not in byte) of the output stream in each MB has been calculated.

Step 2. Shift output streams in parallel. In this step, if the last byte of output stream does not have 8 bits, all bits in this byte should be shifted to the head of the next MB’s stream simultaneously. Surely, the next stream should shift in the same offset in order to connect with the new bits from previous MB’s stream. In order to accomplish this step in parallel, the shifting offset of each output stream must be calculated firstly. Offsets can be easily acquired by using the length of output stream in step 1. The length of output stream is recalculated after shifting. For example, the last 2 bits of the MB0’s output stream is shifted to the first 2 bits of MB1’s output stream. The MB1’s original output stream is shifted 2 bits backwards.

Step 3. Merge output streams. After step 2, output streams are aligned in byte boundary, except the last MB. The starting address of each output stream is calculated by using the stream length parameter. All outputs streams are merged into one stream according to the starting address.

C. Implementation of Parallel CAVLC

By using the approaches presented above, parallel CAVLC encoder is mapped on stream processing. As shown in figure 11, we implement the encoder on Geforce 8800GTX GPU and STORM SP16-G220 for encoding 1080p HD frames sequence. Kernels are executed on stream multiprocessors of GPU and Data Parallel Unit of STORM which are represented by PPU. Streams are presented in the shared memory/cache of GPU and Lane Register Files of STORM represented by on-chip memory. The key points of implementations are provided as follows.

1) Input data organization

Based on the capacity of on-chip memory, residual data of 120 Macro Blocks are merged into one input stream. Since a 1920x1080 frame has 120x68 Macro Blocks, kernels should be executed 68 times to accomplish CAVLC encoding. Before each iteration execution, data of the next 120 Macro Blocks are organized into a new input stream. Because data in the input stream are consumed by PPUs in a round robin manner, we have to organize the elements in stream according to the order they would be used. Besides, all the five VLC look-up tables in the reference code are represented by only one table stream in order to reduce stream number (STORM’s stream number is limited in 8).

2) Kernel split and kernel parameterization

Encoding of Luma AC/DC and Chroma AC/DC modes take really similar processing procedure. Only one function implements the different encoding modes in the reference code.
At the beginning of the function, a series of branch operations are executed to decide which mode should be executed. We make two different implementations in the GPU and STORM. STORM is sensitive to control flow. Hence we split one function into 4 kernels for reducing the branch operations in the kernel. We use a reusable parameterized kernel to represent different modes on GPU. In the kernel, different executing phases are optional by parameters.

V. PERFORMANCE ANALYSIS AND COMPARISON

A. Performance Analysis

Our target application is 1080p HDTV on baseline profile. The evaluation is performed on four platforms with different architecture as shown in table I. On STORM SP16 G220 and Geforce 8800 GTX GPU, both of reference and parallel streaming code are executed to encode 200 frames of a high definition video sequence BlueSky 1080p. On X86 Core2 E8200 and TI C6416, only reference code is executed. Table II lists the CAVLC encoder performances on the four platforms. The results indicate that our parallel CAVLC encoder takes only 1.02 second to encode 200 frames 1080p. Compared with the original serial CAVLC encoder executed on Core2 E8200 and TI C6416 DSP platform, our CAVLC encoder executed on STORM gets 2x and 32x higher performance respectively.

Does high performance on stream processors mainly depend on the hardware computing capacity? The answer is No. As shown in table II, the execution time of serial reference code on Core2 E8200, STORM, and Geforce 8800 GTX are very close. That means the hardware advantage of stream processors does not work when running a serial CAVLC code. As shown in the speedup column, our parallel CAVLC encoder gains 3.03x and 2.08x higher performance on STORM and GPU respectively. It proves that the high performance on stream processors benefits from stream processing model.
scanning). Table III lists the execution time and percentage of each component in our H.264 encoder. The execution time occupancy rate of CAVLC in the entire H.264 encoder decreases from 34% (as shown in figure 4(b)) to 16% by using our parallel methods. The result shows that the overall performance of encoding 200 frames Blue-sky 1080p is 6.49 seconds, which means encoding performance is 30.6FPS. It proves that our parallel CAVLC encoder coupled with stream processor enables a real time encoding of 1080p/30 H.264 video.

B. Comparing with other Parallelization Methods

Apart from streaming, there are many parallelization methods to accelerate CAVLC, such as frame/slice parallelism, MB/block pipeline, component parallelism within MB, and fine granularity parallelism within block. Most of them have been used in hardware and software CAVLC encoder.

Frame, slice parallelism and our MB parallelism belong to spatial parallelism. Compared with our MB parallelism, frame/slice level parallelism eliminates data dependency between MBs, but it will increase the requirement of bandwidth due to the large parallel granularity.

Reference [2] [3] [4] [13] [14] and [15] design hardware or software CAVLC encoders which adopts MB-level or block-level pipeline parallelism. The encoding procedure is divided into several stages. However, the performance of the pipelining design is limited by the maximum execution time required for one of the stages. For example, [14] proposes a software CAVLC encoder implemented on a fine-grained many-core system named AsAP. This technique divides CAVLC encoder into 15 tasks and each task is executed on one core of AsAP. Compared with our streaming methods, the technique focuses on pipeline parallelism and TLP with the risk of unbalanced workloads on different cores. For instance, the zigzag reorder costs 377 cycles, whereas the CAVLC scanning costs 600 cycles when processing one 4x4 block. It implies that computing capacity of some cores in AsAP is wasted. On the contrary, our parallel encoder keeps workloads on each PPU balanced.

The performance of two software parallel methods (AsAP [14] and our streaming) and their appropriate platform configurations are shown in table IV. AsAP encodes two 720p frame and STORM encodes 200 1080p frames. Parallel CAVLC encoder based on stream processing achieve 5.1ms per frame, which is 5x higher than AsAP.

VI. CONCLUSION

This paper presents a parallel CAVLC encoder based on stream processing. In order to eliminate restrictions of parallel in encoding and bit-stream assembling, several methods are applied: (1) Enlarging parallel processing granularity to MB. (2) Parallel predicting. (3) Redundant calculating. (4) Parallel stream producing and assembling. The experiment results show that our CAVLC encoder achieves significant speedup and its performance satisfies the real-time requirements of 1080p H.264 encoding. The high performance is achieved by transforming the original serial code to a parallel code on stream processing model. Without additional requirement for any special dedicate ASIC design or algorithm, this parallel method can be applied in many programmable processors supporting stream processing.

REFERENCES