Streaming HD H.264 Encoder on Programmable Processors

Nan Wu, Mei Wen, Wei Wu, Ju Ren, Huayou Su, Changqing Xun, Chunyuan Zhang
Computer School, National University of Defense Technology, Changsha, Hunan, P. R. of China, 410073
nanwu@nudt.edu.cn

ABSTRACT
Programmable processors have great advantage over dedicated ASIC design under intense time-to-market pressure. However, real-time encoding of high-definition (HD) H.264 video (up to 1080p) is a challenge to most existing programmable processors. On the other hand, model-based design is widely accepted in developing complex media program. Stream model, an emerging model-based programming method, shows surprising efficiency on many compute-intensive domains especially for media processing. On the basis, this paper proposes a set of streaming techniques for H.264 encoding, and then develops all of the code based on the X264 reference code. Our streaming H.264 encoder is a pure software implementation completely written in high-level language without special hardware/algorithm support. Real execution results show that our encoder achieves significant speedup over the original X264 encoder on various programmable architectures: on X86 Core™2 E8200 the speedup is 1.8x, on MIPS 4KEc the speedup is 3.7x, on TMS320 C6416 DSP the speedup is 5.5x, on stream processor STORM-SPI6 G220 the speedup is 6.1x. Especially, on STORM processor, the streaming encoder achieves the performance of 30.6 frames per second for a 1080P HD sequence, satisfying the real-time requirement. These indicate that streaming is extremely efficient for this kind of media workload. Our work is also applicable for other media processing applications, and provides architecture insights into dedicated ASIC or FPGA HD H.264 encoders.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming – Parallel programming.

General Terms
Algorithms, Performance.

Keywords
Stream, H.264 encoder, real-time, programmable, 1080P HD

1. INTRODUCTION
H.264 standard[1] developed by the Video Coding Experts Group (VCEG) and the ISO/IEC Motion Picture Experts Group (MPEG) is widely adopted in applications from high definition living room entertainment (BluRay/ HD-DVD) to Handheld terminals (DVB-H). It can save 25%-45% and 50%-70% of bitrates when compared with MPEG-4 Advanced Simple Profile (ASP) and MPEG-2, respectively. However, the H.264 coding performance comes at the price of significantly increased computational complexity. Especially, real-time encoding of high-definition H.264 video (up to 1080p) is a challenge to most existing programmable processors.

Stream model, an emerging model-based programming method, shows surprising efficiency on many compute-intensive domains especially for Media/Graphic Processing and Scientific Computing [2][3]. This gives a substituted approach to accelerate applications on programmable processor instead of long-term and expensive dedicated ASIC design.

To meet this goal, we try to implement a streaming high-definition (HD) H.264 encoder. However, the inherent difficulty lies in constructing the initial algorithm representation in the stream model. This paper presents a set of novel streaming techniques for H.264 encoding. Furthermore, the paper implements a streaming full HD (1080p) H.264 encoder, and evaluates it on multiple programmable processors, including desktop processor, embedded processor, DSP and emerging stream processor. Our results show that the streaming encoder achieves significant speedup over the original X264 code on all evaluated processors. Especially, real-time encoding of HD H.264 sequence (up to 1080p) is achieved on the stream processor, while its power consumption and cost also meet the requirement of embedded applications.

This study focuses on program level optimization without any special requirement for algorithm or hardware. Therefore, the streaming techniques presented in this paper can be applicable to other media applications with similar features described in section 3, as well as to multiple programmable processors. The work in this paper is an in-depth study of streaming techniques for a complex multimedia application. Previous studies focused on streaming techniques for a single kernel or benchmark on a dedicated stream processor[4][5]. Also, our approach carefully considered parallelism during streaming and optimized entire software framework, making streaming encoder performs much more optimally than ordinary dataflow implementation[6]. In addition, the streaming HD H.264 encoder also can be the infrastructure of hardware HD H.264 encoder’s implementation.

The paper is organized as follows. Section 2 summarizes the stream model and how to map it to programmable processors. In Section 3, the profiling and the design considerations are described. Then, the streaming implementation of HD H.264 encoding system will be addressed in Section 4. Evaluation results are presented in Section 5 and discussed in Section 6. Finally, Section 7 concludes the paper.

2. Background
2.1 Stream model
In the stream model, the data primitive is a stream, an ordered set of data of an arbitrary data type. Operations in the stream model are expressed as operations on entire streams. These operations include
streamgather and streamscatter, and computation in the form of kernel. The streamgather and streamscatter operations are defined as follow [14]:

\text{streamGather}(s, \text{addrs}); s \text{ is a stream of elements from dereferencing a stream of memory addresses--addrs.}

\text{streamScatter}(s, \text{addrs}); \text{Stores elements of stream } s \text{ to the memory addresses contained in stream addrs.}

The gathers and scatters of data could be from or to sequential, strided, or random locations in an array. Kernels operate on one or more streams as inputs and produce one or more streams as outputs. Moreover, considering execution efficiency on hardware, stream model does not allow global variables, pointers, function calls, or control-flow constructs other than loops appearing in kernels.

A program followed by stream model – stream program is constructed by chaining stream operations together. The program framework of a stream program is shown in Figure 1(b), while the one of a regular program is shown in Figure 1(a). It can be seen that though kernel resembles function, they differ in two ways: 1. there are only three types of the structure of kernels including pipeline, branch and feedback circle, nested/cross call are not supported. 2. Kernels are also restricted to only operate on local data: a kernel’s stream outputs are functions only of their stream inputs, and kernels may not make arbitrary memory references. In addition, stream is directional and continuous. It cannot be accessed arbitrarily, which is different from data structure in regular program such as global data structure, data array and other variables.

![Figure 1. Comparison of program frameworks](image)

Transforming a regular program to a stream program requires analyzing the dataflow through the desired algorithm and from it, dividing the data in the program into streams and the computation in the program into kernels, which called streaming. The optimal stream program decouples completely computation and memory accesses by boosting the memory reads before the computation, and postponing writes of the live data to memory after the computation. So that it is able to effectively overlap computation and memory.

### 2.2 Mapping stream program to programmable processors

For a general purpose programmable processor, some of these mappings are straightforward such as mapping none-kernel computations to the scalar units, global memory operations to main memory directly. However, mapping the three kinds of stream specific operations (streamgather, streamscatter, and kernel) is more challenging. The mapping focuses on two key parts. First, mapping streams to on-chip memory such as cache and register file. Second, mapping kernels to execution unit, where parallelism need especially be considered if there are multiple execution units or multiple cores. Introduction of mapping stream program to programmable processors has already been discussed by previous studies [7][8][15] in detail.

### 3. Profiling and Analysis

To focus on the target specification, X264 software [9] is used as the reference code. X264 code has already simplified and optimized original H.264 reference implementation, making it very popular on many programmable platforms. We consider the baseline profile mainly targeted for HDTV1080P videos. Figure 2 shows the skeleton of the reference code. It can be seen that the H.264 encoder is composed of four major processes: Analyse, Encode, CAZLC (Context-based Adaptive Variable Length Coding) and Deblocking Filter. In addition, there are 30 functions (e.g. four modes of 16x16 MacroBlocks(MB) luma prediction, nine modes of 4x4 MB luma prediction and Chorma prediction in Intra Prediction, four modes of IME, FME in Inter Prediction, DCT, quantization, zig-zag scan, IDCT and dequantization in transform coding,coefficients computing and residual data writing in CAZLC, filtered samples generating in Deblocking Filter) being defined as key functions, which cover most time-consuming parts of the H.264 encoder. It can be seen that the outside of key functions are three sets of nested loops: the outermost loop for walking over the input frame (frame loop), the median loop for walking over the slices of the frame (slice loop), the innermost loop for walking over the MBs of the slice (MB loop).

![Figure 2. Skeleton of X.264 code](image)

PAPI [10], a software analyzer at the instruction level, is used by this paper to show the program character of the H.264 encoder, and to find the critical parts for streaming implementation. The profile is acquired on an X86-based platform (platform parameters are described in table1). As a result, profile shows that while the H.264 encoder exposes four salient characteristics suitable for stream model as follows: high computation rate, high predictable computation pattern, producer-consumer locality, and abundant data level parallelism, there are also huge challenges for streaming in following three aspects.

#### 3.1 Function vs. Kernel

**Deep Function Call Depth.** As shown in Figure 2, the function call depth of the X264 code is very deep and the key functions reside close to the bottom in the call graph. For example, the call depth of the key function quant4x4() is 7 (3-levels function call and 4-levels
nested loop), besides this function will be called 130560 times per one frame, leading to significant calling overheads. However, on stream model, the function will usually be mapped into kernels, which prohibit the nested function structure to alleviate calling overhead.

**Fine Data Processing Granularity.** Long data stream and coarse data processing granularity are in favor of exploring ILP (Instruction Level Parallelism) and DLP (Data Level Parallelism) in kernel and avoiding short stream effect [3]. However, major processes in the X264 code are MB dependent. So, the length of stream is limited to the size of a MB, which is 256 Bytes. In addition, the MB-level parallel processing in kernel is also difficult, because data processing granularity of X264 code is restricted to a pixel for the same reason.

**Long Sequential Loop Path.** The encoding path of H264 is so long that lots of sequential operations are included in MB loop. Profile shows that the average instruction reference interval, the number of instructions between two iterated references of the same instruction, of X264 code is 4 ×10^4 that may easily exceed the capacity of the I-Cache or register file in current processors. This causes I-Cache missing frequently during each iteration of MB loop. As Cache miss rates for each part of the program shown in Figure 3, I-Cache miss rates are relative high. As not shown in Figure 3, when the cache size is decreased to 16 KB, the I-Cache miss is increased to 4.59% for Analyse. To make matters worse, in those multi-issue processors, kernels are so sensitive to cache misses that even relative low miss rate results in significant increase in execution time of kernel. This is because a large number of issue slots are wasted during each stall cycle due to the deeply exploited parallelisms of the code.

Moreover, frequently use of global variable in the X264 code, significantly increasing the average variable live range up to 521330 instructions, leading to great storage and bandwidth pressure. On the contrary, to exposes producer-consumer locality to hardware, efficiently, stream model requires decoupling communication and computation that data can only transfer between kernels via stream explicitly.

**Arbitrary Data Access.** Arbitrary data access including pointer access, large-scale jump access and dynamic address access are very common in the X264 code. They are mainly caused by accessing along different dimensions of a multi-dimensional array, neighbor field searching and table lookup. However stream model does not allow arbitrary accesses to global data structure within kernels. In this context, applications that require data dependent address generation must first generate a set of indices and use those to gather data in to streams that can then be used as inputs to subsequent kernels.

### 3.3 Control-intensive Component vs. Compute-intensive Component

On a typical dual-core X86 processor Core\textsuperscript{TM}2 E8200, the encoding performance of the X264 code is 5.64fps. Our performance measurement shows the time consumption order of the processes: (1) Analyse, (2) Encode, (3) CAVLC, (4) Deblocking Filter, and (5) Others, while the detail fraction of execution time is illustrated in Figure 4. It can be seen that most of the computation load is spent in the top four processes. Comparing with these four parts, the other parts are characterized by control-intensive, because they are mainly constructed by control component such as initialization, branches selection and format adjustment. As shown in Figure 4, control intensive components, take 7.2% of the execution time. According to Amdahl law, if we don’t accelerate this part, to achieve real-time encoding performance (30fps), we have to speedup the top four parts over 8 times. However, without additional special hardware or changing arithmetic algorithm, this degree of improvement is very difficult to be achieved. Therefore, streaming those control-intensive components is also very important.

**Multiplex Modes.** Many parts of H.264 encoder have multiplex modes. For example, there are 17 different modes for Intra Prediction while 259 kinds of partitions for Inter Prediction. Six kinds of 2D transform, 4×4/2×2 DCT/IDCT/Hadamard transform, are involved in reconstruction loops. In the X264 code, function is implemented for each mode, whereas there are many similar code segments between them. Taking 17 modes of Intra Prediction for instance, DC values, horizontal mode prediction, vertical mode prediction and SAD (Sum of Absolute Differences) are almost the same. If the steaming converts these functions to kernels directly, a lot of instructions of the kernels may be fetched and stored on chip redundantly, increasing the instruction bandwidth and storage pressures that are very crucial on programmable processors such as DSP and Embedded processors.

**3.2 Structure Variable vs. Stream**

Numerous Global Variables. A mount of global variables is defined in the X264 code to carry communication among functions implicitly. As a result, functions in the X264 code are tightly coupled with several large global data structures such as x264.t, x264_param_t, which making they closely interdependent. However, on stream model, the function will usually be mapped into kernels, which prohibit the nested function structure to alleviate calling overhead.
nodes are extended gradually one level by one level by taking the callee functions as children of the caller functions, until the following cases occur: 1. The node is a key function itself; 2. The node will be merged into a single key function with other brother nodes. For example, the depth of the sub-tree with leaf node E2−quant4x4() is 5. Finally, a transform tree of the x264 code generated contains 74 nodes (including 41 leaf nodes), and the depth of the transform tree is 5.

Step2: traversing the transform tree at the start of the top-level nodes in breadth-first order gradually one level by one level. Operations performed on each node including:

**Flattening.** If the current node is non-leaf node, all true children of the current node moves up to a level higher and then the current node will be deleted. In other word, the caller function corresponding to the current node may be split into multiple callee sub-functions. As shown in Figure 5, there are seven functions including x264_mb_analyse_inter_p16x16() (D6 node), x264_mb_analyse_inter_p() (D7 node) and etc. called from inside x264_mb_analyse_inter (C2 node), we firstly move these seven functions to the same level as C2, and then deletes the function C2. As a result, the depths of true children are all subtracted 1. Note that some functions and variables' names need to be renamed during flattening to avoid the risk of side-effect.

**Figure 5. Illustration of key function horizontalization**

**Merging.** A caller function may be split into multiple callee sub-functions during flattening. However, when the call dependency is tightly coupled or data dependency between functions is tightly coupled, it is inefficient to divide the function into separate functions. In this case, the tightly coupled functions are merged into a single function to preserve the functionality of the code. For example, the four leaf nodes D1, D2, D3 and D4 were split from a single function to preserve the functionality of the code. For example, the four leaf nodes D1, D2, D3 and D4 were split from a single function to preserve the functionality of the code. For example, the four leaf nodes D1, D2, D3 and D4 were split from a single function to preserve the functionality of the code.

**Enlarging Data Processing Granularity.** Although the key functions are not nested any more, they are still not suitable for converting into kernels directly due to their fine data processing granularity mentioned in section 3.1. This section describes how to enlarge the data processing granularity of the key functions by relaxing MB dependency in H.264 encoding.

In Inter Prediction, the inter Lagrange mode decision takes MBs dependency in Motion Vector (MV) prediction into consideration. There are three modes to calculate MV as shown in Figure 6(a), in any of which the MV of each block is generally medium predicted by neighboring blocks. The cost function can be computed only after prediction modes of neighboring blocks are determined which also causes inevitable sequential processing. To solve these problems, the medium of MVs from left MB are replaced by the MVs from top-left MB in all of these modes. For example, in the first mode, the exact MV cost of the block is the medium of the MV0, MV1, and MV2. As shown in Figure 6(b), the Motion Vector Predictors (MVPs) of blocks are changed to the medium of MV3, MV1, and MV2 in order to facilitate the parallel processing and MB pipelining. Modified code still accord with H.264 standard, Claire QCIF 10Hz series, for example, is used to test PSNR. We investigate that the PSNR of modified Inter Prediction is only decreased at most 0.4DB than original X264 code that does slight influence on image quality, while Inter Prediction can be applied to all MBs in the same row, which can increase the data process granularity of Inter Prediction to 30MB.

**Figure 6. Eliminating MB dependency in Inter Prediction**

In Intra prediction, a prediction block is formed based on planar extrapolation of previously encoded and reconstructed neighboring MBs. The prediction for each MB is still dependent on values of the left, left-top, top and top-right neighboring MBs. Our solution is replacing it to employ original data rather than reconstructed ones to perform prediction tests before choosing the best mode. The residue, however, is evaluated using previously decoded data to avoid drifting. Results show that, for high definition sequences, the quality loss is negligible [11]. We are, thus, able to parallelize the Intra Prediction parts on all MBs in the same slices, which can increase the data process granularity of Intra Prediction to 2MB.
of left and top neighboring MB value. Since these values are produced by function x264_macroblock_encode() that is executed before CAVLC, so all of the dependent values can be precalculated and prefetched as that does in DeBlocking Filter. Second, each encoded MB is linked in final output stream bit by bit, so the MB's position in the output stream dependent on its prior MB's position and size. To make matters worse, the size of each encoded MB is variable. It means that although multiple MBs can be encoded by CAVLC in parallel, it still have to link them one by one. To solve this problem, decouple the VLC process into two stages: 1. Encoding, which can be applied to multiple MBs in parallel; 2. Linking, which includes three sub stages: 1) align it to byte boundary; 2) calculate each MB's shift offset in output bit stream; 3) directly store each MB to its right position by streamscatter operation. By this improvement, the data processing granularity of CAVLC is enlarged to 2MB either.

In DeBlocking Filter, current MB is dependent on residual samples of left and top neighboring MB. Since these values are produced by IDCT that is executed before DeBlocking Filter, so all of the dependent values can be precalculated and prefetched, which makes possibility of parallel processing at MB granularity.

**Encapsulating Function into Kernel.** After MB's dependency being relaxed by previous step in all key functions, each key function can be encapsulated into a kernel to process MBs in parallel by the following steps:

**Step 1:** splitting the MB loop outside of the key functions. As a result, the large loop body is partitioned into several small loop bodies, and each small loop body contains a key function and an innermost loop for walking over all MBs of a slice. In addition, our result shows that the L-Cache miss rates of each part illustrated in Figure 3 are all reduced to less than 1% by this step.

**Step 2:** collecting intermediate result. Before splitting, intermediate result produced by one iteration is transferred between key functions directly, since these functions are tightly coupled in the same loop body. After splitting, all intermediate result produced by a key function during all iterations must be stored so that they can be consumed by the next key function in a lump. Therefore, the storage space of intermediate data has to be enlarged. Usually, lager data arrays are used to replace original intermediate variables. These data arrays are the rudiment of streams. For example, instead of dct4x4[16][4][4] for a single MB, array dct4x4[N][16][4][4] (N is the number of MBs in a slice) is used for collecting all intermediate MB results in a slice that are produced by the DCT transform.

**Step 3:** inserting loop control sentence at the each edge of partitioning. Then, encapsulate each key function and the innermost loop into a kernel.

By now, kernels exploiting MB level parallelism are produced primarily. It can be seen from Figure 7 that the function based code framework consists of a lot of loop nest, shown in Figure 2, is transformed to kernel based code framework that each kernel encapsulates a key function by a MB loop.

**Kernel Parameterization** In this step, we use few reusable parameterized kernels to replace dedicated kernels for multiplex modes. Parameterized kernel extracts common segments from different kernels to as a mode sharing part, and then residual different ones are also integrated in the kernel but are optional by some parameters.

![Figure 8. Example for kernel Parameterization (mode sharing parts are highlighted in gray box)](image)

To explain how to implement parameterized kernel, Intra Prediction of 16x16 MB luma and Chorma are taken as an example in this section. Because chroma variety in a image is relatively smooth, the Intra Prediction of Chorma and 16x16 MB luma are very similar with each other that both of them consists of DC/horizontal/vertical/plane mode prediction. Hence the two kernels (16x16 MB luma and Chorma) are replaced by a single parameterized kernel, as shown in Figure 8. The procedures of four prediction modes are shared to support both chroma and luma Intra Prediction. The input and control flow are configured by two sets of parameters. First, a set of mode parameters are used to select the input stream of predictors extended from the block boundaries (8 pixels for chroma, 16 pixels for luma) and process by mode sharing procedures but with different loop iteration numbers. Second, another set of parameters are used to assert constant values in predictor generator for current MB.

**4.2 Structure Variables to Stream**

**Global Variable Localization** The purpose of this step is to convert global variables as many as possible to local variables. By identifying the true live ranges among kernels, variables in the X264 code can be classified into three types of variables as follows:

- **Local variable**, as demonstrated as variable A in Figure 9, only produced and then consumed inside a kernel. **Pseudo global variable**, variable B for example, is a global variable that can be accessed by multiple kernels, but the value of B inside one kernel does not affect the outcome of the other kernel. Global constants are
classified as this class. True Global variable also may be accessed by multiple kernels, but there are write after read dependency between the accesses, as demonstrated as variable C in Figure 9.

![Figure 9. Classification and localization of variables.](image)

Profile shows that, most members of global structures in the X264 code belong to pseudo global variable. These pseudo global variables can be firstly stripped from the structures, and then be substituted by multiple renamed copies in each kernel, as an example demonstrates in Figure 9 where the pseudo global variable B is renamed to B0 in kernel1. In addition, there are still some variables belongs to true global variable. For these variables, we split their live rang into several epochs — regions contiguous in time. In order to simplify the analysis, we define epoch at the granularity of entire kernels. An example is shown in Figure 9. One of the variable C’s epochs—epoch2 lies inside kernel2 with write-after-read dependency, so we rename or rematerialize C to C0 in that kernel. Otherwise, if an epochs lies between kernels but without write-after-read dependency like epoch4, we eliminate its corresponding variable’s live rang directly. As a result, the live rang of a true global variable is split into several short epochs, each of which lies between a pair of kernels with write-after-read dependency only.

Converting Structure Variables to Stream. Although previous step localized the variables, all input/output and intermediate variables between kernels are still defined as global structures. Therefore, this step firstly converts all structures into streams or parameters for kernel according to their data granularity: large-scale variables are defined as stream, while the small-scale ones are defined as parameters. Then, assembles the elements of a stream sequentially, and appends attributes such as index, length and direction on it.

![Figure 10. The dataflow graph of the H.264 encoder](image)
By now, the H.264 encoder is completely transformed from the X264 code to streaming style, which consists of decoupled streams, kernels and few necessary controls logic. The dataflow graph of transformed and original X264 encoders is shown in Figure 10. In the dataflow graph of stream code, those global structures are decomposed to multiple short live range variables by localization step, and expose their producer-consumer relationships between kernels to programmers. For example, all members of x264_t, the largest global structure in the X264 code, have been stripped from x264_t and then localized. Then, these members have been assembled into streams such as fenc, fdec, mv, dct4x4. The similar transform also occurs on x264_param_t, another key global structure, except that, some members of it such as i_width, i_height and i_frame total are defined as parameters instead of stream. Meanwhile, since dependency between MBs has been relaxed, much longer streams containing all MBs in a slice are assembled, such as mb_type_strm, none_zero_strm, mode_strm shown in Figure 10(b).

However, there are still some unfavorable factors in the dataflow graph of our encoder so far. First, not all streams produced by its producer kernel are directly consumed by its consumer kernel. Some reordering and controlling operations, highlighted by dash border rectangle box in Figure 10(b) are inserted between kernels. Second, producer-consumer stream reuse is not fully exploited neither between kernels nor inside a kernel. The following two approaches are used to optimize dataflow graph more efficiently.

**Stream Regularization** In H.264 encoder, some producer-consumer locality is expressed in the irregular form. In other words, some streams are consumed by one or more later kernels in a different order than the ones they are generated in. These streams are called *irregular stream*. The irregular stream cannot be directly used by the later kernel due to stream model does not allow arbitrary accesses to global memory within kernels. Therefore, some reordering operations, such as transpose, indexed streamgather/streamscatter and conditional selection, are required to insert between kernels. However, these operations will increase the waiting time of kernels[13].

![Figure 11. Intra Predication example for stream regularization](image1)

Stream regularization intends to make the produced order and consumed order of stream consistent by adjusting either of it. An example is shown in Figure 11. The former Intra Predication kernel generates an intermediate result stream in row-major order. The later DCT kernel needs to apply the algorithm along a neighborhood of 4x4 pixels, requiring a reordering of the stream. To avoid inserting reordering operations, we adjust the calculation order of Intra Predication kernels in consistent with that of DCT. Thereby, the result stream form Intra Predication can be directly use by DCT kernels. Through regularization, most reordering operations for irregular streams, which are highlighted by dash border rectangle box in Figure 10(b), are eliminated. It means each stream produced by its producer kernel can be directly consumed by its consumer kernel.

**Stream reuse** Reusing as many streams as possible by kernels is another crucial factor for efficiently exploiting producer-consumer locality. In our streaming H.264 encoder, streams reuses are majorly enhanced by following two levels:

1. **Inter kernel stream reuse.** Streams read from memory or generated by kernel, are consumed by one or more later kernels at the granularity of the entire stream. A trivial example is shown in Figure 12. It shows field search which are commonly used in Motion Estimation. For an object MB in frame1, Motion Estimation searching a 32x32 MBs range around corresponding position of the target MB in frame0. As shown in Figure 12 (a) , the left-most 32x32 searching MBs for previous target MB can be reused by the current target MB search range. In stream, these parts of data are expressed as a sequence of sub-stream (indicated as gray blocks in the Figure 12(a)).

2. **Intra kernel stream reuse.** This reuse is something different from traditional data reuse. Because stream just can be accessed forward along one direction, prohibiting any rollback, so the reuse can only occur on most recent used date that are buffered in the kernel. To exploit Intra kernel stream reuse, we define a FIFO buffer for each stream in a kernel. Each time, buffer popping the data from the head, and then push the same number of data from the input stream into the tail, or vice versa for output stream. Thereby, a fixed number of stream data are always available in the kernel for reusing. In example of Motion Estimation, each time the kernel1 calculate the SAD for all 3x3 pixel filed in a MB to estimate whether it is the most matching one, then a 1x3 pixels are input from stream and attach to the right of original field, constructing a new filed for next SAD calculation. Overlapping parts of two neighboring 3x3 field (indicated as gray blocks in the Figure 12(b)) are reused in kernel.

![Figure 12. Motion Estimation example for stream reuse](image2)

4.3 Control-intensive Component to Compute-intensive Component

As shown in Figure 10 (b) with dash border ellipse box, there are still some control-intensive components in the streaming encoder, such as glue and branch operations between kernels. It brings lots of non-kernel computing overhead and unpredictability.
According to analysis in section 3.3, those control-intensive ones also need streaming.

This paper proposes two solutions. The first is creating a new kernel for those glue computations or merging them to the neighbor kernels. Taking function macroblock_cache_load() that is used to initialize loading image samples for example, this step should encapsulate it into a dedicated kernel. The second is converting fine granularity branches to redundant computations by executing all branches, and then selecting the right one from the results. For instance, functions x264_mb_encode_i4x4() and x264_mb_encode_i16x16(), corresponding to 16x16 and 4x4 mode transform coding respectively, are completely different in top level. The former has a additional Hadamard transform procedure for DC values than the latter. Therefore, we synthesize the processing procedures of two modes into one by adding Hadamard transform to 4x4 mode, and adds a mode-based selection between Hadamard-transformed and non-Hadamard-transformed results at the last stage of transform coding. This solution is especially efficient for those processors with multi-cores/lanes, since they can provide abundant execution units for redundant computations. Through this step, most glue and branch operations, highlighted by dash border ellipse box in Figure 10(b), are encapsulated into kernels and most branch operations are eliminated.

4.4 Optional Parallelism Optimization

As is well known, most modern programmable processors are designed to benefit from either or all three kinds of parallelisms (ILP, DLP, TLP) exposed by parallelized program. On the other hand, our streaming implementation also exposes multiple levels of parallelism inherent in H.264 encoding. This section will discuss parallelism optimizations of the streaming H.264 encoder, which are optional for various architectures.

Instruction-Level Parallelism is widely supported by modern programmable processors, such as superscalar (X86), VLIW (DSP, STORM), or the structure of the deep pipeline(X86, MIPS). Our work is exploiting ILP within our streaming encoder by executing multiple independent operations among those that must be applied to a single stream element (i.e. within a single iteration of the kernel loop body) in parallel. This may be achieved via unrolling and/or software pipelining main loop bodies in each kernel.

Data-Level Parallelism may be expressed since the kernel loop body may be applied to many records of an input stream in SIMD fashion. This requiring the processor the stream program is being executed on has sufficient compute resources, for example, multiple cores (X86 Core2D2 has two cores) or multiple lanes (STORM-220IP has sixteen SIMD lanes). For instance, the Sub16x16_DCT kernel can be applied to all of the elements of its input fdec stream in parallel.

Task-level Parallelism is exploited by overlapping kernel execution and memory operation. Without changing the dependency among kernels and streams, we adjust the execution order of memory and kernel operations in the program to exploit as many TLP (Task Level Parallelism) as it can. The adjustment is based on the high predictability that there is little data-dependent variation in the control flow for stream code. Figure 13 shows an example. The left part of Figure13 illustrates the original execution order of kernel and memory operations, where all memory accesses occur between a pair of sequential kernels.

Thereby, waiting for the intervening memory load/store completed, the latter kernel cannot start immediately after the former kernel, although there is no dependency between them. We adjust the code order as shown in the right part of Figure13 by moving data load for quant table behind the kernel load for DCT, so that these two operations can executed with kernel Analyse and kernel DCT in parallel. Similar adjustment may be performed for other stream operations. In addition, TLP may also be exploited through executing kernels on different cores or computational lanes.

5. Results

Table 1. Experimental platform configurations summary

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Processor</th>
<th>Desktop Processor</th>
<th>Embedded Processor</th>
<th>DSP Processor</th>
<th>Stream Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core No.</td>
<td>X86 E8200</td>
<td>MIPS 4KE</td>
<td>TMS320 C6416</td>
<td>STORM-SP16 G220</td>
<td></td>
</tr>
</tbody>
</table>
| CPU/Frequency      | 2.67GHz            | 330MHz            | 600MHz             | System MIPS 330MHz  
| 1/2                 |
| Peak Power         | 95W                | <1W               | 1.6W               | 10W            |
| Memory Bandwidth   | 10.7GB/s           | 8.3 GB/s          | 2.4 GB/s           | 8.3 GB/s       |
| SRAM size          | N/A                | N/A               | N/A                | 256KB          |
| D-Cache size       | 2x32KB             | 16KB              | 16KB               | 16KB on System MIPS  
| Streaming Metrics |                   |                   |                   |                 |

The evaluation is performed on four typical programmable processors with different architecture as shown in table 1. On each platform, both of X264 and streaming code encode 200 frames of a high definition video sequence – BlueSky 1080P 25Hz, meanwhile a standard h.264 decoder decodes the encoded bit streams directly to verify the correctness. Comparison of two decoded results shows that, the output bit streams of our streaming encoder is correct. Furthermore, the compression rate of BlueSky sequence of our encoder is 50:1 that is consistent with X264 code. Then, the following comparisons focus on performance of the encoder.

Streaming Metrics In section 3, we have presented a set of metrics to measure the suitability of program for stream model. Table 2 shows values of these metrics before and after streaming of H.264 encoder’s code. It can be seen that the X264 code is transformed efficiently to stream code.
Table 2. Code’s features before and after streaming

<table>
<thead>
<tr>
<th>Streaming Metrics</th>
<th>X264 code</th>
<th>Stream code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data processing granularity</td>
<td>256 Bytes</td>
<td>30 KB~2MB</td>
</tr>
<tr>
<td>Average Call depth of key function</td>
<td>5.82</td>
<td>3</td>
</tr>
<tr>
<td>Average instruction reference interval</td>
<td>4*10^6 instr.</td>
<td>7*10^6 instr.</td>
</tr>
<tr>
<td>Average variable live range</td>
<td>521330 instr.</td>
<td>7635 instr.</td>
</tr>
<tr>
<td>Average l-Cache miss rate (on x86)</td>
<td>2.1%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Control-intensive overhead(on x86)</td>
<td>7.2%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

Table 3. Performances on processors for 200 frames of a high definition video sequence – BlueSky 1080P

<table>
<thead>
<tr>
<th>Platform</th>
<th>X264 code</th>
<th>Stream code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Frame/s</td>
</tr>
<tr>
<td>P4 E8200</td>
<td>35.4s</td>
<td>5.6fps</td>
</tr>
<tr>
<td>MIPS 4KEc</td>
<td>1052.6s</td>
<td>0.19fps</td>
</tr>
<tr>
<td>TMS320 C64</td>
<td>699.8s</td>
<td>0.29fps</td>
</tr>
<tr>
<td>STORM-SP16</td>
<td>39.9s</td>
<td>5.0fps</td>
</tr>
</tbody>
</table>

Performance Table 3 lists the overall execution time, frame rate and speedup of both X264 code and stream code. It shows that the stream code achieves significant speedup over the X264 code on all four platforms. It’s quite excellent for streaming, a program level optimization technique without any special hardware and algorithm support, to obtain such degree of speedup on various different architectures. The experimental results also show that our encoder is capable of real-time 1080p H.264 video encoding on a completely programmable processor—STORM.

The effects of streaming are various on different parts of the H.264 encoder. Figure 14 shows the execution time breakdown of the encoder on processors before/after streaming. Taking STORM platform as example, for Analyse, stream code achieves 7.9x maximum speedup over X264 reference code, while its fraction of execution time decrease from 64% to 49%. Meanwhile, for CAVLC, the speedup is the minimum, only 2.7x. As a result, its fraction rises from 7% to 16%. This result indicates that streaming is more efficient for parts with higher computation intensity.

The effects of streaming are also various on different processor architectures. As shown in table 3, our encoder achieves lower improvement on typical scalar architectures such as X86 and MIPS, this is because these processors have relative fewer process unites, while the large amount of die area are occupied by complex control structures that are useless for highly predictable stream model. In addition, caches in these processors are not optimized to directly take advantage of producer-consumer locality to increase available on-chip bandwidth, but rather are optimized to exploit temporal and spatial locality to reduce average memory latency that is not very important for bulk asynchronous memory gathers/scatters in stream pattern. Whereas, DSP and emerging stream processor achieve higher speedup because of their simple control hardware but multi-way VLIW or multi-lane SIMD units, meanwhile their software managed Cache and register files are also very important to implement streamgather and streamscatter operations efficiently.

6. Discussion

Comparing with other Parallelization Techniques In essence, streaming is a model-based parallelization technique. However, there are many general parallelization techniques for accelerating H.264 encoding, such as frame/slice level parallelism, wavelet/MB pipeline and fine granularity block parallelism[12]. Compared to streaming, these techniques only focus on data dependency of some level with the risk of side-effect. For example, frame level parallelism will increase the requirement for bandwidth, while slice level parallelism will reduce the compression rate. Furthermore, the streaming techniques can be used combining with other parallelization techniques. Our measurement on X86 shows that slice level parallelism (2 slices) accelerates HD H.264 encoding to 6.38fps for BlueSky 1080P sequence, while the streaming accelerates it to 10.2fps, and the combination of these two techniques can accelerates it to 12.3fps.

Note that the further speedup of our software H.264 encoder may be achieved by algorithm level optimization beyond program level optimization. For example, K-L or wavelet transform can be used to replace DCT transform in transform coding, CABAC can also be used to replace CAVLC in entropy coding. However, this leads to two-sided effect, because it may bind improvement with some special algorithm that will limit the applicable domain of streaming technique.

Implication For future higher resolutions of H.264 codec, we consider that our streaming code is also suitable for mapping on modern GPUs or using it as template to design dedicate hardware. One reason is that stream code flattens the nested function structure, and omits most global communications and unpredictable control flows, which reducing the requirement for complex control logic. Another reason is that stream code exposes abundant compute intensity, parallelism and producer-consumer locality that are well suited to be accelerated by special PE arrays and pipelines of GPU or some other highly parallelized hardware. However, there are also some most critical differences that a developer would face. We assume that memory access latency, global synchronization with standard processor and unsuited stream and kernel size would be critical differences. The stream code should be further improved for these platforms.

7. Conclusion

This paper proposes a set of novel model-based technique called streaming for H.264 encoding, and implements a streaming HD H.264 encoder that is a software encoder programmed in high-level language. Our experiments show that the encoder can run on multiple programmable processors, and acquires significant
performance accelerating. Especially, real-time encoding of high-definition H.264 video (up to 1080p) is achieved on a programmable stream processor. The performance speedup is achieved by transforming code to a special model-stream model without additional requirement for any special hardware or algorithm. Hence, it makes streaming technique more attractive. After all, for this type of media processing whose standards are various and often varied, programmable processor is a more wiser choice to save time and cost, on condition that the performance must be comparable to ASIC.

The stream code achieves this level of performance because it matches the capabilities of modern processor architecture to the demands of H.264 standard in four ways. First, it organizing data into stream to reduce the demand for memory bandwidth, since streaming exploits the producer-consumer locality so that data generated by one kernel is staged in local storage, the Cache or register files, and then used by the next kernel without ever being sent to main memory. Furthermore, Streaming bulk memory access is particularly optimized by modern memory systems. Second, encapsulating arithmetic operations into kernel provides the much higher arithmetic density needed to exploit the parallelism inherent in H.264 encoders, and reduces the negative influence by deep function nesting. Thirdly, by overlapping stream operations, each of which moves or transforms hundreds of data records, hardware is able to exploit the latency tolerance inherent in H.264 encoder to further increase throughput. Finally, streaming expose more optimizing opportunities for those parallel architectures. In short, streaming achieves its performance using the same methods that have traditionally been exploited in special hardware, but completely implemented in software.

Acknowledgments
This work was supported by National Nature Science Foundation of China No. 60673148 and 60703073, 863 Project of China No.2007AA01Z286.

REFERENCES
[12] Ishfaq Ahmad , Yong He , Ming L. Liou, “Video Compression with Parallel Processing,” Parallel Computing, v.28 n.7-8, p.1039-1078, August 2002