ICICS-PCM 2003
15-18 December 2003
Singapore

Efficient Multithreading Implementation of H.264 Encoder on Intel Hyper-Threading Architectures

Steven Ge[1], Xinmin Tian[2], and Yen-Kuang Chen[3]
[1]Intel China Research Center, Intel Corporation, Beijing, P.R. China
[2]Intel Compiler Laboratory, Software Solution Group, Intel Corporation
[3]Microcomputer Research Laboratories, Intel Corporation
[2,3]3600 Juliette Lane, Santa Clara, CA 95054, USA

Abstract

Exploiting thread-level parallelism is a promising way to improve the performance of multimedia applications running on multithreading general-purpose processors. This paper describes our work in developing the first multithreading implementation of the H.264 encoder. We parallelize the encoder using the OpenMP programming model, which allows us to leverage the advanced compiler technology in the Intel® C++ compiler for Intel Hyper-Threading architectures. We present our design considerations in the parallelization process. We describe an efficient multi-level data partitioning scheme that increases performance of a multithreaded H.264 encoder. Our experiments show parallel speedups ranging from 4.31x to 4.69x on a 4-CPU Intel Xeon™ system with Hyper-Threading Technology.

1. Introduction

H.264 [2] is an emerging video coding standard proposed by the Joint Video Team (JVT). It is aimed at high-quality coding of video contents at very low bit-rates. H.264 uses the same hybrid block-based motion compensation and transform coding model as existing standards such as H.263. However, a number of new features and capabilities have been added in H.264 to improve its coding performance. As a result, the H.264 encoding process is more computationally intensive than existing standards. Hence, we are motivated to improve the speed of the encoder.

In [7], it is demonstrated that using MMX/SSE/SSE2 technology can speed up the H.264 decoder performance by 2-4x. We apply the same technique to the H.264 reference encoder as well. Table 1 shows the speedups for each key module residing in H.264 encoder. Although the encoder is 2-3x faster with SIMD optimization, it is still not fast enough for real-time video processing. One way to accelerate the encoder further is to parallelize it to exploit multiprocessor and Hyper-Threading Technology supported by the Intel architecture.

Recently, both hardware and software support for multithreading has increased. While using multithreading hardware to improve throughput of multiple workloads is straightforward, using it to improve the performance of a single workload requires parallelization. Converting sequential programs into multithreaded programs is often difficult. However, the OpenMP shared-memory programming model [4, 5] provides a rich set of features, which allow the compiler to exploit thread-level parallelism and optimize the performance of applications with a few pragmas. The compiler support enables developers to take full advantage of the state-of-the-art architecture features such as Hyper-Threading Technology [3].

Previously, [6] presented an implementation of a multithreaded H.264 decoder, and there is also some work on exploiting parallelism in MPEG encoders [1]. To the best of our knowledge, we are the first to develop a multithreaded implementation of H.264 encoder. In addition, we study on tradeoffs between video quality and many parallelization schemes. [1] used the most straightforward approach, which parallelizes the encoding process at the slice-level. Our scheme is exploiting both slice-level *and* frame-level parallelism.

This paper describes how to efficiently parallelize an H.264 encoder using the Intel OpenMP compiler and demonstrates a speedup of 4.31x to 4.69x on quad-processor systems with Hyper-Threading Technology. The remainder of this paper is organized as follows. Section 2 presents a brief overview of the Intel compiler and Hyper-Threading architecture. Section 3 presents our implementation for a threaded H.264 encoder. Section 4 shows performance results and discusses the results. Finally, Section 5 concludes the paper.

2. Compiler and Architecture

Intel Compiler: The Intel OpenMP implementation in the compiler strives to: (a) generate multithreaded code which gains a true speedup over well-optimized sequential code, (b) integrate parallelization tightly with advanced interprocedure, scalar and loop optimizations such as intra-register vectorization and memory hierarchy oriented optimizations to achieve better cache locality and efficiently exploit multi-level parallelism, and (c) minimize the overhead of data-sharing among threads. The Intel compiler has a single common intermediate representation named IL0 for the C++/C and Fortran95 languages. Hence, OpenMP parallelization, as well as a majority of other optimizations, is applicable through a single high-level transformation irrespective of the high-

0-7803-8185-8/03/$17.00 © 2003 IEEE

Module	Speedup
SAD Calculation	3.5x
Hadamard Transform	1.6x
Sub-Pel Search	1.3x
Integer Transform and Quantization	1.3x
¼ Pel Interpolation	2.0x

Table 1: Speedups of the key modules in H.264 encoder using SIMD-optimization only

level source language [5]. Throughout the rest of this paper, we refer to the Intel C++ and Fortran compilers for Intel architectures collectively as "the Intel compiler". In order to establish the context in which the OpenMP parallelization is enabled.

Architecture: Hyper-Threading (HT) technology brings the concept of Simultaneous Multithreading (SMT) to Intel Architecture. HT makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors [3]. From a software or architecture perspective, this means operating systems and user programs can schedule threads to logical CPUs as they would on multiple physical CPUs. From a microarchitecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources [3].

Figure 1(a) shows a system with two physical processors that are not Hyper-Threading Technology-capable. Figure 1(b) shows a system with two physical processors that are Hyper-Threading Technology-capable. In Figure 1(b), with a duplicated copy of the architectural state on each physical processor, the system appears to have four logical processors. Each logical processor contains a complete set of the architecture state.

With HT technology, the majority of execution resources are shared by two architecture states (or two logical processors). Rapid execution engine process instructions from both threads simultaneously. The Fetch and Deliver engine and Reorder and Retire block partition some of the resources to alternate between the two intra-threads. In short, HT technology improves performance of threaded programs by increasing the processor utilization of the on-chip resources available in the Intel NetBurst™ micro-architecture.

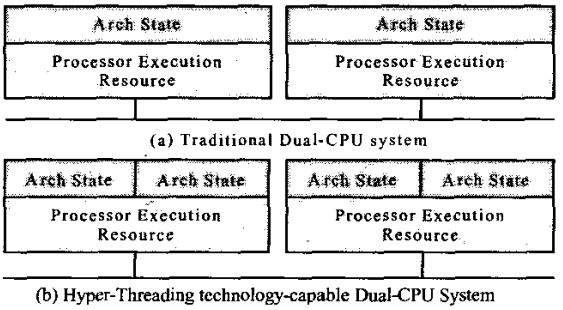


Figure 1: Traditional DP system vs. HT-capable DP system

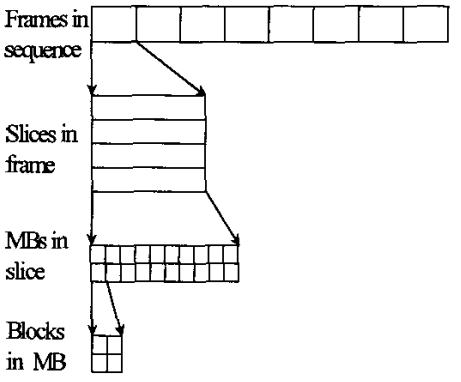


Figure 2: Hierarchy of data domain decomposition in H.264

3. Multithreaded Implementation

There are many opportunities in the H.264 encoder for exploiting parallelism at different levels. In order to achieve the best speedup over its well-tuned serial code on processors with Hyper-Threading Technology, our design is to divide the H.264 encode process into multiple threads via data domain decomposition. A sequence of video is consisted of many groups of pictures (GOP). As shown Figure 2, each GOP includes a number of frames. Each frame is divided into slices, which is the self-content encoding unit and is independent of other slices in the same frame. The slice can be further decomposed into macroblock, which is the unit of motion estimation and entropy coding. Finally, the macroblock can be separated into block and sub-block. These are all possible places to parallelize an H.264 encoder. Section 3.1 describes our judgments of thread granularity. Section 3.2 depicts our proposed implementation.

3.1 Slice-Level vs Frame-Level Parallelism

First, we should decide the thread granularity. One possible scheme of decomposition is to divide a frame into small slices. The advantage of parallelizing among slices is that the slices in a frame are independent. Thus, we can simultaneously encode all slices in any order. On the other hand, the disadvantage is that it will increase the bit rate. Figure 3 shows the video encoder performance (rate-distortion) when a frame is divided into different numbers of slices.[1] When a frame is divided into 9 slices, the bit-rate at the same quality is about 15~20% higher. This is because slices break the dependence between macroblocks. When a macroblock in one slice cannot exploit another macroblock in another slice for compression, the compression efficiency decreases. In order not to increase the bit-rate at the same video quality of the parallelized encoder, we should exploit other parallelism in the video encoder.

Another possible scheme of exploiting parallelism is to identify independent frames. Normally, we encode a

[1] In H.264, a slice can be as large as a frame. Breaking a frame into multiple slices is not required.

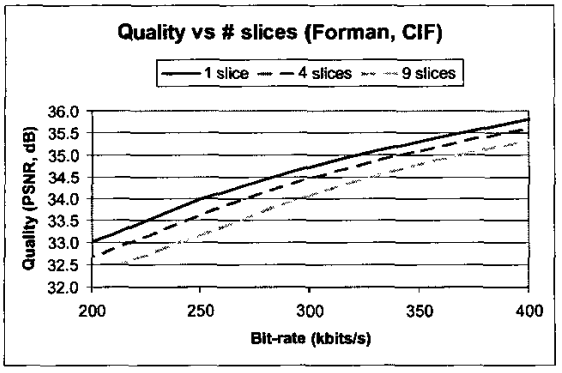


Figure 3: Encoded picture quality vs the # of slices in a picture

sequence of frames using an IBBPBBP... structure.[2] There are two B frames between P frames. While P frames are reference frames (which other P or B frames depend on), B frames are not necessary. In our implementation of H.264 encoder, we treat B frames as non reference frames to explore more parallelism. In the rest of this paper, we will assume this simplification by default. The dependence among the frames is showed in Figure 4. In this PBB encoding structure, the completion of encoding a P frame will make the subsequent one P frame and two B frames ready for encoding. The more frames encoded simultaneous, the more parallelism we can explore. Therefore, *P frames are on the critical point in the encoder.* Accelerating P-frame encoding will bring more frames ready for encoding, and avoid the idle of threads. In our implementation, we will encode I or P frames first, then B frames.

Unlike dividing a frame into slices, utilizing parallelism among frames will not increase the bit rate. However, the dependence among them will limit the threads scalability. The trade-off is to combine the above two approaches into one implementation. *We first explore the parallelism among frames; we can gain performance from it without bit rate increase. After we reach the upper limit of the thread number can be reached by the frame-level parallelism, we will explore the parallel among slices subsequently.* As a result, we utilize processor resources as much as possible and keep the compression ratio as high as possible (the bit-rate as low as possible). (More details will be given in Section 4.1.)

3.2 Implementation

We divide the encoder into three parts: input pre-processing, encoding, and output post-processing. The

[2] (1) I-frame in video codecs stands for intra frames, which can be encoded or decoded independently. Normally, there is an I-frame per 15~60 frames. (2) P-frame stands for predicted frames, each of which is predicted from a previously encoded I-frame or P-frame. Because a P-frame is predicted from the previously encoded I/P-frame, the dependency makes it harder to encode two P-frame simultaneously. (3) B-frame stands for bi-directional predicted frames, which are predicted from a two previously encoded I/P-frames.

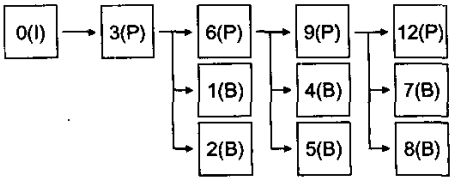


Figure 4: Data dependence among frames. The numbers are the display order[3] of the video frames

input processing will read uncompressed images, perform some preprocesses, and then issue the images to encoding threads. The preprocessed images are put in a buffer, called image buffer. The output processing will check the encoding status of each frame and commit the encoded result to the output bit-stream sequentially. After that, the entries in the image buffer are reused to prepare the image for encoding. Although the input and output processes of the encoder must be sequential due to the natural of the H.264 encoder, the computation complexity of input and output processes are insignificant compared to the encode process. Therefore, we use one thread to handle the input and output processes. This thread is the master thread in charge of checking all the data dependency.

We use another buffer, called slice buffer, to explore the parallelism among slices. After each image is preprocessed, the slices of the image will put into the slice buffer. The slices in the slice buffer are independent and ready for encoding (the readiness of reference frames is checked during the input process). In this case, we can encode these slices out of order. To distinguish the priority differences between the slices of B frames and the slices of I or P frames, we use two separate slice queues to handle them.

Figure 5 depicts the final multithreading implementation. Figure 6 shows the pseudo code. We use one thread to process input and output in order and use other threads to encode slices out of order.

4. Performance Results and Analysis

We conduct the performance measurements of our multithreaded H.264 encoder on (1) Dell 530 MT workstation, built with dual Intel Xeon processors (4 logical processors) running at 2.0GHz with Hyper-Threading enabled, 512K L2 Cache, 1GB memory; (2) IBM 360 Server, built with quad Intel Xeon processors (8 logical processors) running at 1.5GHz with Hyper-Threading enabled, 256K L2 Cache, 512K L3 Cache, 2GB memory. Unless specified otherwise, the resolution of the input video is 352x288 in pixels or 22x18 in MBs. It is guaranteed that there are enough slices for eight threads, when we take slice as the basic encoding unit for a thread.

[3] In video codec, there are two orders. One is the display order; the other one is the encoding order. While the display order in a GOP is IBBPBBP, the encoding order is actually IPBBPBB.

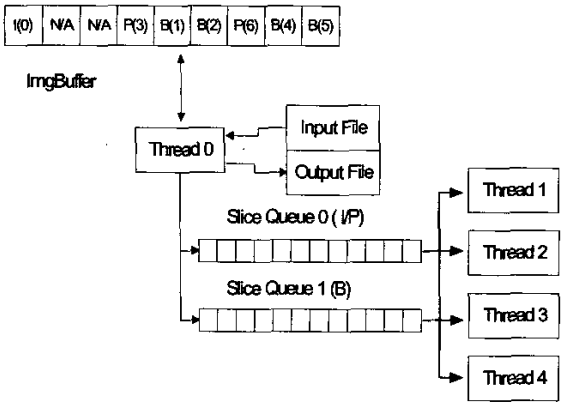


Figure 5: Implementation with image and slice buffers

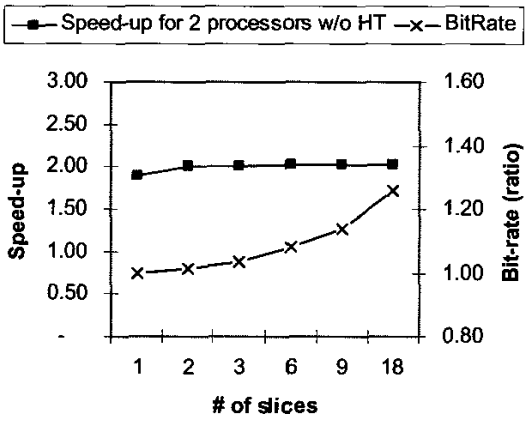
```
omp_set_nested( # of encoding thread + 1)
#pragma omp parallel sections
{
#pragma omp section
  {
    while ( there is frame to encode )
    {
      if ( there is free entry in image buffer )
        issue new frame to image buffer
      else if ( there are frame encoded in image buffer )
        commit the encoded frame, release the entry
      else                              // dependency are handled here
        wait;
    }
  }

#pragma omp section
  {
  #pragma omp parallel num_threads(# of encoding thread)
    {
      while ( 1 )
      {
        if ( there is slice in slice queue 0)
          encode one slice              // higher priority for I/P-frames
        else if ( there is slice in slice queue 1)
          encode one slice              // lower priority for B-frames
        else if ( all frames are encoded )
          exit;
        else
          wait;                 // wait for the main thread to put more slices
      }
    }
  }
}
```

Figure 6: Pseudo code of the multithreaded H.264 encoder

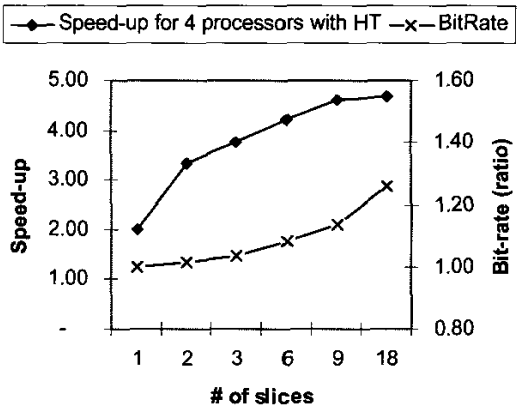
The profile of encoder is configured as following: (1) all intersearch types are enabled; (2) only the nearest previous frame is used for inter motion search; (3) maximum search range is 16; (4) 1/4-pel motion vector resolution is used; (5) hadamard transform is enabled; (6) quant parameter is set to 16 for all frames; and (7) rd-optimization without restrictions and losses is used.

4.1 Tradeoff Between Speedup and Compression Efficiency

A frame can be partitioned up to 18 slices. Taking a slice as the base encoding unit for a thread can reduce the synchronization overhead because there is no data dependency among slices in a single frame for performing



(a)



(b)

Figure 7: Speedup and bit rate vs the # of slices in a frame

encoding. As we mentioned earlier, partitioning the frame into multiple slices can increase the degree of parallelism, but, it also increases the bit-rate. One of challenges is that we aim at achieving a higher speedup with a lower bit-rate without sacrificing any image quality. Therefore, we should choose the slicing threshold carefully.

Figure 7 shows the speedup of encoding and the bit rate with variation of the number of slices for each frame. In Figure 7(a), the number of slices ranges from 1 to 18 with a constant quality of encoded frames. There is a good speedup when the number of slices for a frame is 1 to 2 on the DELL 530 platform, and the speedup is almost flat while the number of slices changing from 2 to 18. Meanwhile, the bit-rate increasing is smaller if the number of slices is less than 3, but it starts going up from 3 slices to 18 slices. One important observation is that partitioning a frame to 2 or 3 slices delivers the best tradeoff that achieves a higher speedup and a lower bit rate. Figure 7(b) shows that we need more than 3 slices to keep 8 logical processors busy on the IBM 360 platform. Essentially, we need 9 threads to achieve an optimal performance for 4 physical processors with Hyper-Threading enabled. Our heuristic is to keep the number of

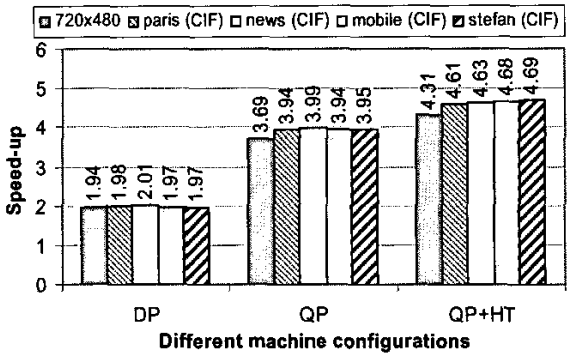


Figure 8: Encoder speedups on different video sequences after multithreading

	Without HT	With HT
Retired 1 instruction	20.03%	25.67%
Retired 2 instructions	16.52%	18.62%
Retired 3 instructions	7.79%	8.55%

Table 2: Instructions retired breakdown

slices roughly same as the number of logical processors. This is a simple yet and efficient way to achieve a good performance and a good image quality with an optimal tradeoff while generating enough slices to keep threads busy for encoding.

4.2 Performance on Multiprocessor with HT

Figure 8 shows the speedup of our multithreaded H.264 encoder on the IBM 360 quad-processor system with Hyper-Threading Technology. In our implementation, a picture frame is partitioned into 9 slices. For five different input video sequences, our multithreaded H.264 encoder achieves a speedup ranging from 1.94x to 2.01x on 2 processors, a speedup ranging from 3.69x to 3.99x on 4 processors, and a speedup ranging from 4.31x to 4.69x on 4 processors with Hyper-Threading enabled.

With Hyper-Threading Technology enabled, we have an additional 1.2x speedup. This speedup can be explained by taking a look the microarchitecture metrics. Table 2 shows the distribution of the number of instructions retired per cycle. The data is collected on the Dell 530 dual-processor system with the second processor disabled. Although there is no instruction retired for almost half of the execution time, the probability of retiring more instructions is higher with Hyper-Threading Technology. This indicates that higher processor utilization is achieved with Hyper-Threading Technology.

5. Conclusions

As the emerging codec standard becomes more complex, the encoding and decoding processes require much more computation power than most existing standards. To the best of our knowledge, this paper presents the very first and efficient multithreaded implementation of the H.264 video encoder, which exploits multiple levels of parallelism. Tradeoffs of using different parallelism in video codec and the final implementation scheme have been illustrated in detail. We are the first one who considers compression efficiency degradation as well as parallel speedup. Thus, the proposed scheme not only provides good execution speedup, but also keeps the video degradation as minimal as possible.

Our multithreaded implementation based on OpenMP programming model also demonstrates that it is very simple yet and efficient to exploit parallelism through adding a few pragmas in the serial code. The programmers can rely on the parallelizing compiler to convert the serial code to multithreaded code automatically. The performance results have shown that the code generated by the Intel OpenMP compiler delivers an optimal speedup truly over the well-optimized sequential code on the Intel Hyper-Threading architecture. Our work demonstrates that Hyper-Threading Technology can gain us ~20% performance, which is a performance gain beyond the multiprocessor performance without limited additional cost. The performance speedups ranging from 4.31x to 4.69x supports the merit of our implementation and the efficiency of multithreaded code generated by the Intel OpenMP compiler. The techniques demonstrated in this work can be applied not only to H.264, but also to other video/image coding/decoding applications on personal computers.

Acknowledgements

The authors thank the Intel compiler group for developing the Intel C++ high-performance compiler. We also acknowledge the efforts of Eric Q. Li and Xiaosong Zhou at Intel China Research Center in developing the SIMD-optimized encoder.

References

[1] Y.-K. Chen, M. Holliman, E. Debes, S. Zheltov, A. Knyazev, S. Bratanov, R. Belenov, and I. Santos, "Media Applications on Hyper-Threading Technology," Intel Technology Journal, pp. 47-57, Feb. 2002.

[2] ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC, Document JVT-D157, 4th Meeting: Klagenfurt, Austria, July 2002.

[3] D. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Microarchitecture and Architecture," *Intel Technology Journal*, Vol. 6, Q1, 2002.

[4] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," Version 2.0, March 2002, http://www.openmp.org

[5] X. Tian, Y.-K. Chen, M. Girkar, S. Ge, R. Lienhart, S. Shah, "Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel OpenMP Compiler", in Proc. of *Int'l Parallel & Distributed Processing Symposium*, April 2003.

[6] E. B. van der Tol, E. G. T. Jaspers, and R. H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture", in Proc. of SPIE Conf. on Image and Video Communications and Processing, Jan. 2003.

[7] X. Zhou, E. Q. Li, and Y.-K. Chen, "Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions," in Proc. of SPIE Conf. on Image and Video Communications and Processing, Jan. 2003.