Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures

Yen-Kuang Chen[1], Xinmin Tian[2], Steven Ge[3], and Milind Girkar[2]
[1]Architecture Research Laboratory, Intel Corporation
[2]Intel Compiler Laboratory, Software Solution Group, Intel Corporation
[1,2]3600 Juliette Lane, Santa Clara, CA 95054, USA
[3]China Research Center, Intel Corporation, Beijing, P.R. China
{Yen-Kuang.Chen, Xinmin.Tian, Steven.Ge, Milind.Girkar}@intel.com

Abstract

Exploiting thread-level parallelism is a promising way to improve the performance of multimedia applications that are running on multithreading general-purpose processors. This paper describes the work in developing our threaded H.264 encoder. We parallelize the H.264 encoder using the OpenMP programming model, which allows us to leverage the advanced compiler technologies in the Intel® C++ compiler for Intel Hyper-Threading architectures. After we present our design considerations in the parallelization process, we describe two efficient methods for multi-level data partitioning, which can improve the performance of our multithreaded H.264 encoder. Furthermore, we exploit different options in the OpenMP programming. While one implementation that uses the task queuing model is slightly slower than the other implementation, it is easier to be read than the other one. The results have shown good speedups ranging from 3.74x to 4.53x over the well-optimized sequential code performance on a system of 4 Intel Xeon™ processors with Hyper-Threading Technology.

Keywords: H.264 standard, Hyper-Threading Technology, thread-level parallelism, OpenMP, multimedia

1. Introduction

H.264 [8] is an emerging video coding standard proposed by the Joint Video Team (JVT). The new standard 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 those existing standards, such as, H.263 and MPEG-4 [7]. Moreover, a number of new features and capabilities have been introduced in H.264 to efficiently improve the coding performance. As the standard becomes more complex, the encoding process requires much more computation powers than most existing standards. Hence, we need a number of mechanisms to improve the speed of the encoder.

One possible mechanism to improve the application speed is to process the task in parallel. In [20], it is demonstrated that using MMX/SSE/SSE2 technology can speedup the H.264 decoder performance by 2-4x. We applied 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, its speed is still not fast enough to meet the expectation of real-time video processing. Furthermore, the optimized sequential code can not take advantage of Hyper-Threading Technology and multiprocessor supported by the Intel architecture. In other words, there are still a lot of rooms for us to continue improving the performance of the H.264 encoder by exploiting thread-level parallelism.

Recently, multithreading with computer architecture and compiler support becomes increasingly common. While using multithreaded hardware to improve throughput of multiple workloads is straight-forward, using it to improve the performance of single-threaded workloads requires parallelization. Converting sequential programs into multithreaded programs is difficult for many applications. However, the explicit parallel programming offered by OpenMP shared-memory programming model [5, 11, 12, 15] provides a rich set of features, which allow a compiler to exploit thread-level parallelism and optimize the performance of applications by adding a very few pragmas. The compiler support enables developers to take advantage of the state-of-the-art architecture features, such as, Hyper-Threading Technology [10].

This paper describes how to efficiently multithread a H.264 encoder using Intel OpenMP compiler and demonstrates speedup on quad-processor systems with Hyper-Threading Technology. The remainder of this paper is organized as follows. The Section 2 presents an overview of the Intel parallelizing compiler. Section 3 gives a brief of the Hyper-Threading architecture. Section 4 presents our design and implementations for parallelizing H.264 encoders. In Section 5, we show our performance results

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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

and conduct discussion on the results. Section 6 discusses related work. Finally, concluding remarks can be found in Section 7.

2. Compiler Overview

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 interprocedural, scalar and loop optimizations such as intra-register vectorization [2, 4] and memory hierarchy oriented optimizations [16, 19] 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 pragma-guided parallelization, as well as a majority of other optimizations, is applicable through a single high-level transformation [15] irrespective of the high-level source language. Throughout the rest of this paper, we refer to the Intel C++ and Fortran95 compilers for Intel architectures collectively as "the Intel compiler". In order to establish the context in which the OpenMP parallelization works, we give a brief overview of the Intel compiler.

Multi-Entry Threading (MET): we have developed and implemented the new compiler technology named *Multi-Entry Threading* (MET). The rationale behind MET is that the compiler does not create a separate compilation unit (or routine) for a parallel region or loop. Instead, the compiler generates a threaded entry and a threaded return for a given parallel region or loop [15, 16].

Multi-Level Parallelism (MLP): Intel compiler supports intra-register vectorization for Pentium family processor [2], and software pipelining for Itanium family processor for exploiting instruction-level parallelism (ILP) on top of exploiting thread-level parallelism (TLP). Exploiting MLP (TLP+ILP) ensures the compiler fully utilizes the rich set of performance features of Intel architecture for achieving the highest application performance.

Inter-Procedural Optimization (IPO): this component includes points-to analysis and mod/ref analysis required by many other optimizations. Points-to analysis expands the capabilities of memory disambiguation by determining that which memory locations may be referenced by a memory reference.

High-Level Optimization (HLO): those optimizations in HLO include loop transformations such as loop fusion, loop tiling, loop unroll-and-jam, loop distribution, profile-guided data prefetching, scalar replacement and data optimizations to improve data locality and reduce memory access latency.

Other Scalar Optimization Components: Intel compiler implements an extensive set of scalar optimizations such as branch-merging, strength reduction, constant propagation, dead code elimination, copy propagation, partial dead store elimination, and partial redundancy elimination (PRE) [4].

Task Queuing Model: The Intel compiler supports a task queuing model [16] that can be used to effectively exploit irregular parallelism inherent in applications. This model allows a programmer to parallelize control structures that are beyond the scope of those supported by the standard OpenMP programming model, while still fitting into the framework defined by the OpenMP specification.

Architecture-specific code generation components include instruction scheduling, register allocation, code ordering, advanced instruction selection, and global code scheduling.

3. Hyper-Threading Architecture

Intel's Hyper-Threading technology brings the concept of Simultaneous Multithreading (SMT) to Intel Architecture. However, unlike a proposed research-type SMT processor [17] where most, if not all micro-architectural structures are shared between logical processors, the micro-architectural resources in Intel hyper-threaded processors are managed differently. As detailed in [10], a hyper-threaded processor dynamically operates in one of two modes; in ST (Single Threading) mode, all on-chip resources in Table 2 are given to a single application thread, and in MT (Multi-Threading) mode, resources can be shared, duplicated or partitioned between the two logical processors. As shown in Table 2, structures like caches and execution units are shared between the two logical processors, very much like resource sharing on a research SMT processor [17]. On the

Shared	Trace cache, u-code ROM, execution units, instruction fetch, instruction decode, instruction scheduler, allocator, uop retirement logic, DTLB, L1 D-cache, L2 cache, global history array
Duplicated	Per logical processor architecture state, instruction pointers, renaming logic, ITLB, streaming buffers, return stack buffer, branch history buffer
Partitioned	Re-order buffer, uop queue, memory instruction queue, general instruction queue

Table 2: HW Configuration of Hyper-Threaded Processor

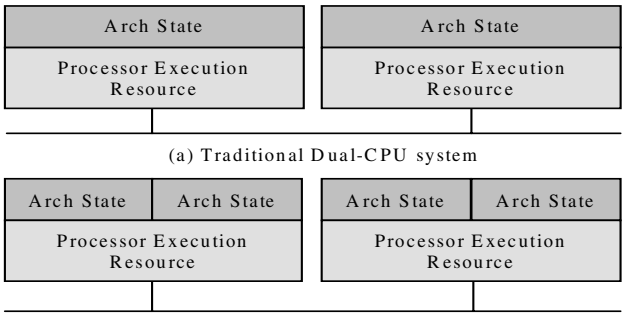


Figure 1: Traditional DP system vs. HT-capable DP system

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

other hand, structures like the reorder buffer are evenly hard-partitioned to prevent one logical processor from taking up the whole resource. In addition, micro-architectural resources like the ITLB and the return stack buffer are replicated for each logical processor.

The Hyper-Threading Technology 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 [10]. 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 duplication of the architectural state on each physical processor, the system appears to have 4 logical processors. From the 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 the micro-architecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources [10].

4. Multithreaded Implementations

There are many potential opportunities in the H.264 encoder for exploiting parallelism at different levels. In order to achieve the best speedup over its well-tuned sequential code on processors with Hyper-Threading Technology, we present our considerations and our design to parallelize the H.264 encode in this section. Section 4.1 describes our criteria of choosing data or task partition. Section 4.2 and Section 4.3 describes our judgments of thread granularity. Section 4.4 depicts our first proposed implementation that uses two slice queues. Section 4.5 shows our second proposed implementation using one task queue.

4.1 Data and Task Decomposition

The H.264 encode process can be divided into multiple threads via data domain decomposition or via functional decomposition naturally.

- Data domain decomposition: As shown Figure 2, in H.264, a sequence of video is consisted of many groups of pictures (GOP). 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.
- Functional decomposition: Each frame should experience a number of functional steps: motion estimation, motion compensation, integral transformation, quantization and entropy coding. The reference frames also need inverse qualification, inverse integral transformation and filter. It is also possible to explore the parallelism amount the functions.

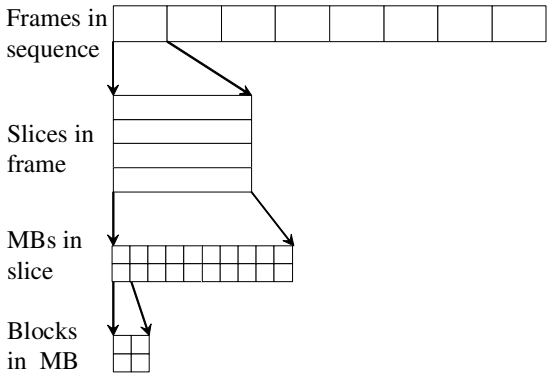


Figure 2: Hierarchy of data domain decomposition in H.264

To choose the best data or task partition scheme, we list the advantages and disadvantages of two schemes below:

- **Scalability:** In the data-domain decomposition, to increase the number of threads, we can decrease the size of the processing unit of each thread. Because of the hierarchical structure in GOPs, frames, slice, MBs, and blocks of H.264 encoder, there are many choices to select the size of processing unit. Thus, it seems easy to achieve good scalability. In functional decomposition, each thread has difference function. In order to increase the number of threads, we must select partition a function into two or more threads. It is a difficult task when the function is unbreakable.
- **Load balance:** In the data domain decomposition, each thread processes the same operation on different data block that has the same dimension. In theory (without cache misses or other non-deterministic factors), all threads should have the same process time. On the other hand, it is difficult to achieve good load balance among functions, as the execution time of each function is determined by the algorithm. Furthermore, how to functionally decompose the video encoder with good load balance highly depends on algorithms. As the standard keeps improving, the algorithms will change over time.

Considering these factors we discussed above, we decided to use the data-domain decomposition as our multithreading scheme.

4.2 Slice-Level Parallelism

After deciding the thread partition scheme, 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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

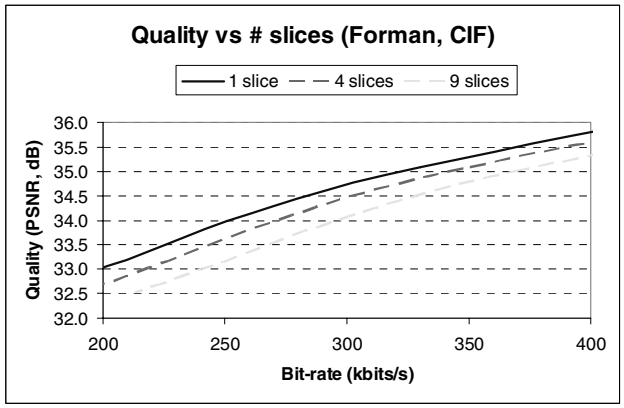


Figure 3: Encoded picture quality vs the # of slices in a picture

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. 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 can not 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.

4.3 Frame-Level Parallelism

Another possible scheme of exploiting parallelism is to identify independent frames. Normally, we encode a sequence of frames using an IBBPBBP… structure.[1] 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. 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.

[1] (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. No frame depends on B-frames.

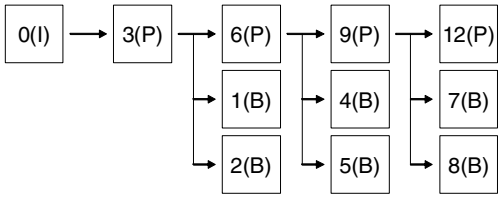


Figure 4: Data dependence among frames. The numbers are the display order[2] of the video frames

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).

4.4 First Implementation Using Two Slice Queues

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

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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

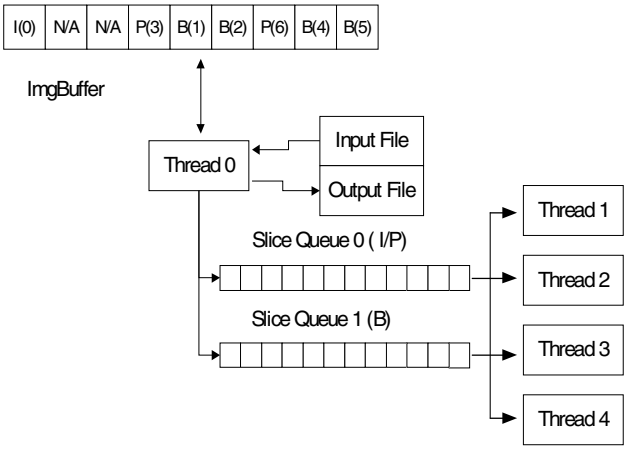


Figure 5: Implementation with image and slice buffers

```
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 using two slice queues

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.

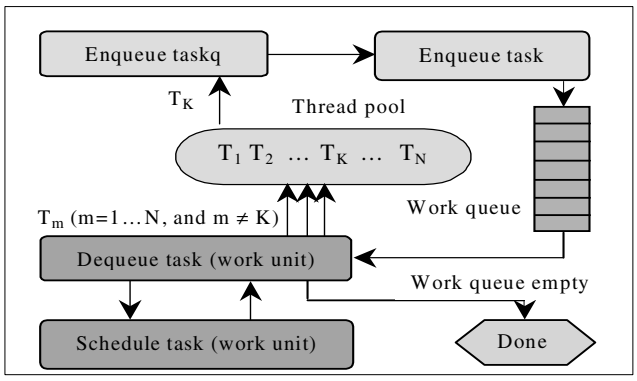


Figure 7: Task queuing Execution Model

```
#pragma intel omp parallel taskq
{
    while ( there is frame to encode )
    {
        if ( there is no free entry in image buffer )
            commit the encoded frame, release the entry;
        load the original picture to memory, prepare for encoding;
        for (all slice in this frame)
        {
        #pragma intel omp task
            {
                encoder one slice;
            }
        }
    }
}
```

Figure 8: Pseudo code of the multithreaded H.264 encoder using the task queuing model

4.5 Second Implementation Using the Task Queuing Model

While our first implementation uses OpenMP pragma, the structure of the parallel code is very different from that of a sequential code. Therefore, in this section, we demonstrate our second proposed implementation that uses the task queuing model [16] supported by Intel OpenMP compiler.

Essentially, given a program with task queuing constructs, a team of threads is created, when a parallel region is encountered. As shown in Figure 7, with the task queuing execution model, from among all threads that encounter a taskq pragma, one thread (T_K) is chosen to execute it initially. All the other threads (T_m, where m=1, …, N and m≠K) wait for work to be enqueued on the work queue. Conceptually, the taskq pragma causes an empty queue to be created by the chosen thread T_K, enqueues each task it encounters, and then the code inside the taskq block is executed single-threaded by the T_K. The task pragma specifies a unit of work, potentially executed by a different thread. When a task pragma is encountered lexically within a taskq block, the code inside the task block is enqueued on the queue associated with the taskq. The conceptual queue is disbanded when all work enqueued on it finishes, and when the end of the taskq block is reached.

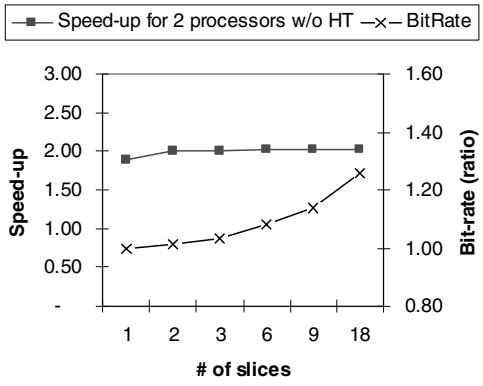
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Our first proposed multi-threaded H.264 scheme uses two FIFO buffers: (1) image buffer and (2) slice buffer. The main thread is in charge of (1) moving raw images into the image buffer *when the image buffer has space*, (2) moving slices of the image buffer into slice buffers *when the slice buffer has space and the image is not yet dispatched*, (3) moving the encoded images out the image buffer *when the image is encoded*. The working threads are in charge of encoding new slices *when there is a slice waiting to be encoded in the slice buffer*. All these operations are synchronized through the image buffers. Hence, it is very natural to use the task queuing model supported by Intel OpenMP compiler.

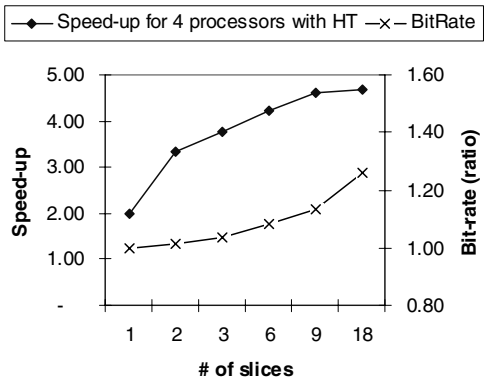
Figure 8 shows the pseudo code of the multi-threading of H.264 encoder using the task queuing model. The new multi-threaded source code is closer to the single-thread code. The only difference is the pragma---which is one of the goals of OpenMP. Furthermore, in this scheme, there is no more control thread. There are only n working threads in total.

5. 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) SHAST server, built with quad Intel Xeon processors (8 logical processors) running at 2.8GHz with Hyper-Threading enabled, 512K L2 Cache (no L3 Cache), 2GB memory. Unless specified otherwise, the resolution of the input video is CIF-resolution (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.



(a)



(b)

Figure 9: Speedup and bit rate vs the # of slices in a frame

5.1 Tradeoff between Speedup and Compression Efficiency

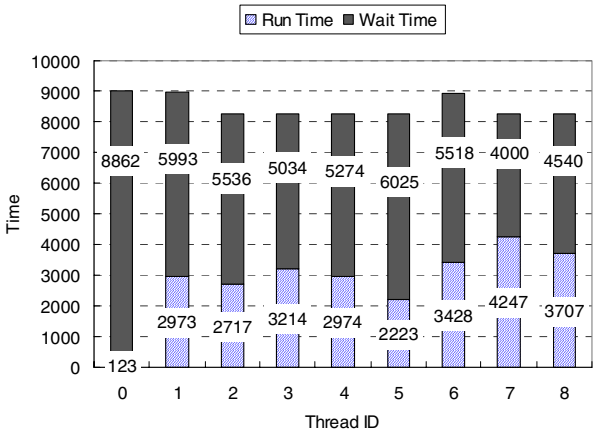
A frame can be partitioned up to 18 slices. Taking a slice as the base encoding unit for a thread can reduce the synchronization overhead because there is no data dependency among slices in a single frame for performing encoding. As we mentioned earlier, partitioning the frame into multiple slices can 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 9 shows the speedup of encoding and the bit rate with variation of the number of slices for each frame in two machine configurations.[3] In Figure 9(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 9(b) shows that we need more than 3 slices to keep 8 logical processors busy on the SHAST platform. Essentially, we need 9 threads to achieve an optimal performance for 4 physical processors with Hyper-Threading enabled.

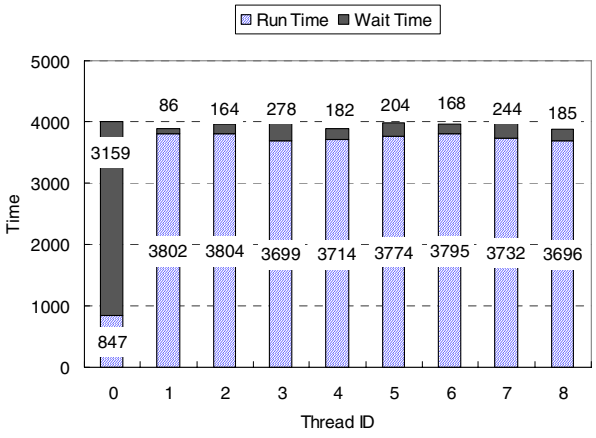
This can be explained from the profile of threads. Figure 10(a) shows the profile when there is only 1 slice in a frame. Figure 10(b) shows the profile when there are 9 slices in a frame. In Figure 10(b), the 8 encoder threads are all busy except the setup time. In this case, almost all processor resources are used---only 3.70% execution time is waiting. On the other hand, in Figure 10(a), about 61.19% execution time of encoder thread is waiting. This is because there is not enough parallelism. Therefore, *during the process of doing trade-off, we should carefully choice the best point.*

[3] In order to contrast the speedup vs the number of slices in a frame over different numbers of processors, we use a 2-processor system and an 8-logical-processor system.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

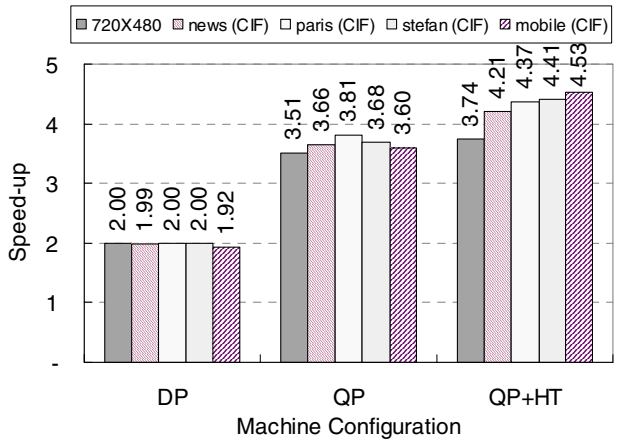


(a)

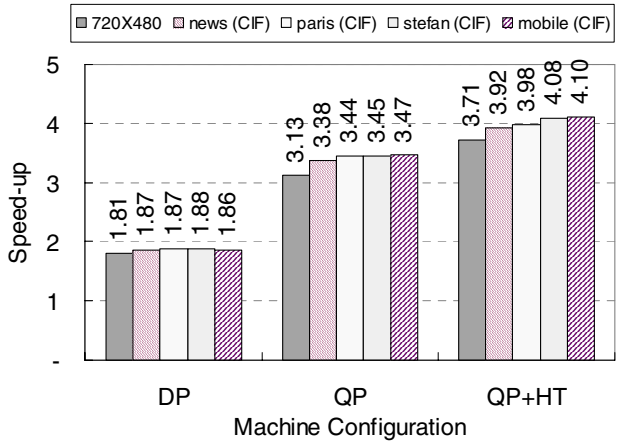


(b)

Figure 10: The execution time profile of the first implementation using two slice queues when (a) there is only one slice in a frame and (b) there are 18 slices in a frame (both on our SHAST system with HT).



(a)



(b)

Figure 11: Encoder speedups on different video sequences after multithreading. (a) uses two slices queues. (b) uses one task queue.

The criterion is keep the slices number in the low level while provide enough slices to let all encoder threads busy. If the slices number is smaller than the threads number, the speedup will decrease. (Figure 13 also shows that the execution time on QP+HT is longer than that on QP if there are only a small number of threads.)

Our heuristic is to keep the number of 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.

5.2 Performance on Multiprocessor with HT and Microarchitecture Metrics

Figure 11 shows the speedup of our multithreaded H.264 encoder on the SHAST quad-processor system with Hyper-Threading Technology. In our implementation, a picture frame was partitioned into 9 slices. In general, our multithreaded H.264 encoders achieved a speedup ranging from 1.8x to 2.0x on 2 processors, a speedup ranging from 3.1x to 3.7x on 4 processors, and a speedup ranging from 3.7x to 4.5x on 4 processors with Hyper-Threading enabled for five different input video sequences.

	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 3: Instructions retired breakdown

To explain the 1.2x speedup with Hyper-Threading Technology enabled, let's take a look the microarchitecture metrics.

First, Table 3 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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

	DELL 530				SHAST			
	UP	UP+HT	DP	DP+HT	UP	DP	QP	QP+HT
Instruction per cycle	0.79	0.90	1.57	1.81	0.76	1.50	2.86	3.20
uops per cycle	1.11	1.26	2.17	2.48	1.112	2.139	4.036	4.365
Trace cache deliver mode %	80.80%	71.13%	80.39%	69.06%	83.73%	82.88%	83.71%	71.98%
Trace cache build mode %	17.59%	25.15%	17.27%	25.42%	16.74%	17.31%	17.08%	22.23%
1st level cache load misses rate	6.24%	9.19%	6.42%	9.02%	5.95%	6.24%	5.87%	8.87%
2nd level cache load misses rate	0.45%	0.56%	0.54%	0.54%	0.15%	0.17%	0.20%	0.28%
Front-side-bus utilization rate	0.65%	1.51%	1.57%	3.74%	0.96%	2.93%	8.53%	13.09%

Table 4: uArch metrics on DELL 530 and SHAST

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.

Second, as shown in Table 4, about 80% of the time the trace cache is under the deliver mode (good for performance) while 18% is under the build mode (bad for performance) without Hyper-Threading Technology. However, when Hyper-Threading Technology is enabled, the deliver mode percentage drops to 70% while the build mode percentage increases to 25%. This indicates the front end of Hyper-Threading system cannot provide enough uops to execution unit. Similarly, the first level cache load miss rate also show the same issue. The number of first level cache misses increase about 50% when Hyper-Threading enabled (miss rate increased from 6% to 9%). This is because the two logic processors in one physical package share the only first-level cache of 8KBytes. In short, our performance gains on Hyper-Threading Technology are limited by the trace cache and the L1 cache for our multithreaded H.264 encoder.

There is no notable impact on other microarchitecture metrics except front-side-bus utilization rate. The number of bus activities does not increase significantly along with the increasing of number of threads. The execution time is reduced due to the better use of processor resources by exploiting enough thread-level parallelism. It results in the increased front-side-bus utilization rate.

5.3 Performance Comparison between 2-Slice-Queue and 1-Task-Queue Schemes

As shown in Figure 11, there are some performance difference between the first implementation with two slices queues and the second implementation with only one task queue. The performance gap is larger when there are more processors. Because the implementation uses two queues to accelerate the encoding of I or P frames, it can provide more slices ready for encoding, especially when there are a large number of processors. On the other hand, the task queuing model in OpenMP maintains only one queue. In this case, all slices are treaded equal. Therefore, there is more idle time in the execution threads when there are a lot of processors, as we will see more details at the end of this section.

Figure 12 shows the execution time profile of the second implementation using one task queue. As mentioned earlier, because the task queuing model in OpenMP only maintains one queue, all slices are treaded equal. Therefore, there may not be enough ready-to-encode slices, as we can see from the amount of idle time in the execution threads. Compared to Figure 10(b), Figure 12 shows that the processors are utilized less efficiently.

5.4 Threading Overhead

In our previous discussion, we mentioned that the number of threads equal to the number of logical processors delivers a good tradeoff between speedup and parallelism. In this section, we study the performance in the case of the number of threads that is greater or less than the number of logical processors. Figure 13 shows that the speedup (of the first implementation using two slice queues) changes along with the number of threads. The speedup grows up along with increasing of the number of threads, it gets to the peak performance when the number of threads equals to the number of logical processors.

An interesting observation is that the speedup is pretty much flat or it drops only slightly when the number of

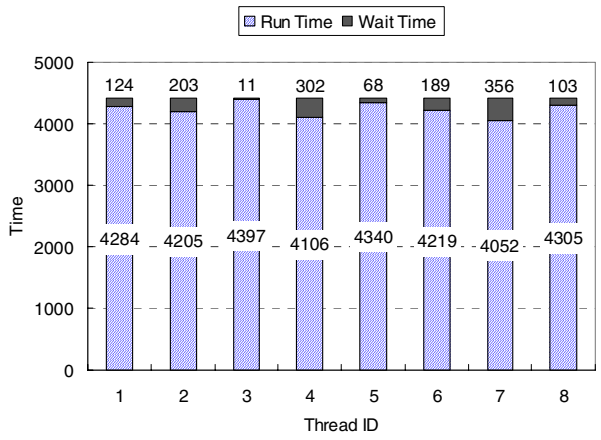


Figure 12: The execution time profile of the second implementation using one task queue (on our SHAST system with HT).

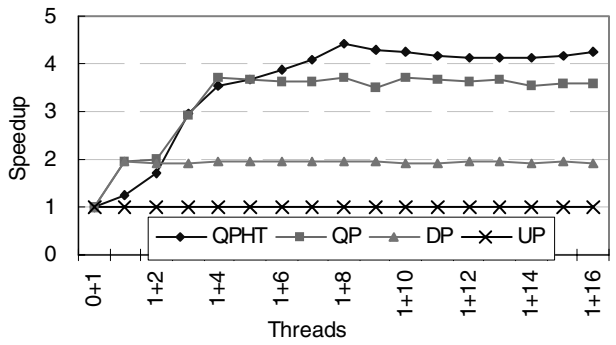


Figure 13: Speedups vs the number of threads

threads is greater than the number of logical processors. It indicates that the overhead due to threading is minor. In other words, the multithreaded code generated by the compiler is efficient on exploiting effective parallelism, and the overhead of the multithreaded runtime library is small. Furthermore, our multithreaded H.264 encoder should have good scalability for large-scale multiprocessor systems because the performance is not sensitive to the number of threads.

6. Related and Future Work

Previously, [18] presented an implementation of multithreading H.264 decoder, and there are also some works on exploiting parallelism in MPEG encoders [1][3][13][14]. To the best of our knowledge, we are the first one who developed the multithreaded implementation of H.264 encoder on the multithreading architecture [6]. In addition, we have done an in-depth study on different tradeoffs in video quality and parallelization. [1][3][13] used the most straightforward approach to encoding the video sequences either by pictures or by slices. Our scheme is slightly more complicated in exploiting both the slice-level and frame-level parallelism.

In the future, we will further analyze the performance impact from different image resolutions. While the resolution of source image can scale from QCIF, CIF, SD to HDTV, most of our current analysis focused on the CIF resolution. Figure 11 shows that the speedup of SD (720x480) format is slightly less than that of CIF (352x288) format. While the speedup is determined by several factors (such as, synchronization and degree of parallelism), our experimental results show that the number of synchronizations per second during encoding SD video is only 1/3 of that during encoding CIF video. Furthermore, SD has a higher degree of parallelism. It will be great to understand the reasons why the speedup of encoding higher-resolution video is less than that of lower-resolution video.

7. Conclusions

As the emerging codec standard becomes more complex, the encoding and decoding processes require much more computation power than most existing standards. H.264 standard includes a number of new features and requires much more computation than most existing standards, such as MPEG-2 and MPEG-4. Even after media instruction optimization, the H.264 encoder at CIF resolution is still not fast enough to meet the expectation of real-time video processing. Hence, we exploit the parallelism to improve the performance of H.264 encoders.

To the best of our knowledge, this paper presented 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 demonstrated 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. We also demonstrated the tradeoff between the source code complexity and the performance using application-managed queues and the task queuing model supported by Intel OpenMP compiler.

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 demonstrated that Hyper-Threading Technology can gain us ~20% performance, which is a performance gain beyond the multiprocessor performance with very little additional cost. The performance speedup ranging from 3.74x to 4.53x have supported the merit of our implementation and the efficiency of multithreaded code generated by the Intel OpenMP compiler.

Acknowledgements

The authors thank all members of the Intel compiler team for their contribution in developing the Intel C++/Fortran high-performance compiler. We also acknowledge the great efforts of Eric Q. Li and Xiaosong Zhou at Intel China Research Center in developing the SIMD-optimized encoder.

References

[1] D. M. Barbosa, J. P. Kitajima, and W. Meira Jr., “Real-Time MPEG Encoding in Shared-Memory Multiprocessors,” in Int’l Conf. on Parallel Computing Systems, 1999.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

[2] A. Bik, M. Girkar, P. Grey, and X. Tian, "Automatic Intra-Register Vectorization for the Intel Architecture," in Int'l Journal of Parallel Programming, April 2002.

[3] 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.

[4] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu, "A New Algorithm for Partial Redundancy Elimination Based on SSA Form," in ACM Conf. on Programming Language Design and Implementation, June 1997, pp. 273-286.

[5] J.-H. Chow, L. E. Lyon, and V. Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors," in Proc. of CASCON, pp. 76-89, Nov. 1996.

[6] S. Ge, X. Tian, and Y.-K. Chen, "Efficient Multithreading Implementation of H.264 Encoder on Intel Hyper-Threading Architectures," in IEEE Pacific-Rim Conf. on Multimedia, Dec 2003.

[7] International Standard Organization, "Information Technology-Coding of Audio-Visual Objects, Part2---Visual," ISO/IEC 14496-2.

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

[9] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-Complexity Transform and Quantization with 16-Bit Arithmetic for H.26L," Int'l Conf. on Image Processing, vol. 2, pp. 489-492, Oct. 2002.

[10] 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.

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

[12] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," Version 2.0, November 2000, http://www.openmp.org

[13] K. Shen, L. Rowe, and E. Delp, "A Parallel Implementation of an MPEG-1 Encoder: Faster than Real-Time," in SPIE Conf. on Digital Video Compression: Algorithms and Techniques, 1995.

[14] H. H. Taylor et al. "An MPEG Encoder Implementation on the Princeton Engine Video Supercomputer," in Proc. of Data Compression Conference, pp. 420–429, 1993.

[15] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, E. Su, "Intel® OpenMP* C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance," Intel Technology Journal, Vol. 6, Q1, 2002

[16] 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 Int'l Parallel & Distributed Processing Symposium, pp. 36-43, Apr. 2003.

[17] D. M. Tullsen, S. J. Eggers, H. M. Levy. "Simultaneous Multithreading: On-Chip Parallelism," in Int'l Symposium on Computer Architecture, June 1995.

[18] E. B. van der Tol, E. G. T. Jaspers, and R. H. Gelderblom, "Mapping of H.264 Decoding on a Multiprocessor Architecture," in SPIE Conf. on Image and Video Communications and Processing, Jan. 2003.

[19] M. J. Wolfe, High Performance Compilers for Parallel Computers, Addison-Wesley Publishing Company, Redwood City, California, 1996.

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

* Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit http://www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE