Encoding H.264 by Thread Level Parallelism

Under the guidance of Dr. K.R Rao

Submitted by:
Shaheer Ahmed
ID: 1000836047
MSEE, University of Texas at Arlington
**Objective:** To perform thread level parallelism on H.264 encoder and to decrease its encoding time.

**Introduction:**
H.264 [1] is an industry standard for video compression, the process of converting digital video into a format that takes up less capacity when it is stored or transmitted. Video compression (or video coding) is an essential technology for applications such as digital television, DVD-Video, mobile TV, videoconferencing and internet video streaming [1]. In order to meet the needs of the industry H.264 was developed by ITU-I and ISO/IEC. The H.264/AVC standard was first published in 2003. It builds on the concepts of earlier standards such as MPEG-2 and MPEG-4 Visual and offers the potential for better compression efficiency (i.e. better-quality compressed video) and greater flexibility in compressing, transmitting and storing video [1]. It is based on modules of block motion compensation, transform, quantization and entropy coding. The H.264 baseline encoder is estimated to be 5x to 10x more complex than the H.263 encoder [2] while the decoder is 2x to 2.5x more than H.263 baseline decoder [3].

H.264 comprises of the following profiles
1. Baseline
2. Extended
3. Main
4. High

Fig.1 describes the profiles comprised in H.264

![H.264 profiles](image_url)

- Baseline profile is used for real-time conversational services such as video conferencing and video calling. [4] [11]
- Extended profile is for the multimedia services over the internet. [4] [11]
- Main profile is designed for digital storage media and television broadcasting. [4] [11]
- High profile is used in the fidelity range extensions for applications such as content contribution, content distribution and studio editing. [5]

The H.264 comprises of both encoder and decoder i.e. it is a codec. The encoder compresses the video into .264 formats using JM 18.4 software and the decoder converts its back to the uncompressed format. This decoder is lossy.

The H.264/AVC encoder takes raw video data as input, compresses them by reducing spatial, temporal, and statistical redundancy, and then generated a compressed bitstream. For redundancy reduction, motion estimation (ME) and intra prediction (IP) together calculate the encoding cost of possible encoding modes, and then perform DCT and quantization operations for each prediction mode. [18] Now, an encoding mode is selected
based on its cost and copies of quantized DCT coefficients of the selected mode to the buffer in entropy coding (EC). The video bitstream output is then generated by compressing quantized DCT coefficients in the buffer using statistical redundancy. [18]

The Inverse quantization and inverse transform block reconstruct the images which is fed back to the intra prediction block for the next iteration, and also delivered to the motion estimation block after passing through the deblocking filter. [18]
**Prediction Modes:**
Prediction exploits the spatial or the temporal redundancy of a video sequence so that only the difference between actual and predicted image is encoded instead of encoding the entire image. There are two types of prediction
   1. Intra prediction
   2. Inter prediction

-Intra prediction is done on the intra frame (I frame). Since there exists a high similarity among neighboring blocks in a video frame, a block can be predicted and reconstructed using the neighboring pixels. [7] Intra macro block is a coded reference to the data only in the current slice. In an intra macro block the luma component can be selected in 3 ways, namely 16x16, 8x8 or 4x4. [16]. A single prediction block is generated for each chroma component as shown in table 1.

<table>
<thead>
<tr>
<th>Intra Prediction Block Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16x16 (luma)</td>
<td>A single 16x16 prediction block P is generated. Four possible prediction modes</td>
</tr>
<tr>
<td>8x8 (luma)</td>
<td>An 8x8 prediction block P is generated for each 8x8 luma block. Nine possible prediction modes. ‘High’ profiles only</td>
</tr>
<tr>
<td>4x4 (luma)</td>
<td>A 4x4 prediction block P is generated for each 4x4 luma block. Nine possible prediction modes</td>
</tr>
<tr>
<td>Chroma</td>
<td>One prediction block P is generated for each chroma component. Four possible prediction modes. The same prediction mode is used for both chroma components</td>
</tr>
</tbody>
</table>

Table.1: Intra Prediction block sizes and notes [17]  
A 16x16 Mb has the 4 modes of prediction as shown in Table 2.

<table>
<thead>
<tr>
<th>Mode 0 (vertical)</th>
<th>Extrapolation from upper samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 1 (horizontal)</td>
<td>Extrapolation from left samples</td>
</tr>
<tr>
<td>Mode (DC)</td>
<td>Mean of upper and left hand sample</td>
</tr>
<tr>
<td>Mode 4 (Plane)</td>
<td>A linear ‘plane’ function is fitted to the upper and left hand samples H and V. This works well in areas of smoothly-varying luminance</td>
</tr>
</tbody>
</table>

Table. 2: 16x16 luma prediction modes [17]

Fig 4: 16x16 luma prediction modes [17]
The possible modes in 4x4 intra macro block are given in Table 3

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 0 (Vertical)</td>
<td>The upper samples A,B,C,D are extrapolated vertically</td>
</tr>
<tr>
<td>Mode 1 (Horizontal)</td>
<td>The left sample I,J,K,L are extrapolated horizontally</td>
</tr>
<tr>
<td>Mode 2 (DC)</td>
<td>All samples in P are predicted by mean of samples A..D and I..L</td>
</tr>
<tr>
<td>Mode 3 (Diagonal down-left)</td>
<td>The samples are interpolated at a 45° angle between lower left and upper right</td>
</tr>
<tr>
<td>Mode 4 (Diagonal down-right)</td>
<td>The samples are extrapolated at a 45° angle down and to the right</td>
</tr>
<tr>
<td>Mode 5 (Vertical left)</td>
<td>Extrapolation at an angle of approximately 26.6° below the horizontal</td>
</tr>
<tr>
<td>Mode 6 (Horizontal down)</td>
<td>Extrapolation at an angle of approximately 26.6° below horizontal</td>
</tr>
<tr>
<td>Mode 7 (Vertical right)</td>
<td>Extrapolation or interpolation at an angle of approximately 26.6° to the right of vertical</td>
</tr>
<tr>
<td>Mode 8 (Horizontal up)</td>
<td>Interpolation at an angle of approximately 26.6° above horizontal</td>
</tr>
</tbody>
</table>

Table. 3: 4x4 luma prediction modes [17]
Inter prediction is done on the predictive frame (P frame) and bi-directional frame (B frame). Since the frame rate of a video sequence is 30 to 60 fps (nominal), the probability of having two successive frames to be similar is very high. The goal of inter prediction is to utilize this temporal redundancy and reduce the data to be encoded. [7] Inter prediction is the process of predicting a block of luma and chroma samples in a current frame from samples already coded and transmitted from another frame or a reference frame. [16] The macro blocks are split into four types as shown in figure 6a and figure 6b.

![Fig 6.a 16x16, 8x8, 16x8, 8x8 macro block partitions [17]](image)

![Fig 6.b 8x8, 4x8, 8x4, 4x4 macro block partitions [17]](image)

**Need for Parallel programming to implement H.264**

Gordon Moore [8] had stated that the transistors available on a semiconductor would double approximately every 18 to 24 months. This law has guided the computer designer for the past 40 years, but many people mistakenly think of Moore’s law as a predictor of CPU frequency which is not true. Over the years the CPU frequency has tended to follow Moore’s law but since past few years the CPU clock speeds have flattened out. This is because the higher speeds cause excessive power consumption, heat dissipation and current leakage. Thus in order to make the most efficient use of the processor resources, computer architects have developed ways to implement many instructions in parallel during the same clock cycle by either using multi cores or instruction level parallelism. [8]

Although the H.264 encoder achieves high compression efficiency [18] when compared to previous versions i.e. H.263, H.262, it increases the computational complexity, which prevents this codec from being widely used in embedded systems without the assistance of a special hardware. [18]. Now that the encoder cannot rely on CPU clock speeds to improve the encoding time, parallelization of the computations is required at the software level. This can be done by exploiting thread-level parallelism at different levels.

- **Task Level Parallelism:**
  The codec is decomposed into functional blocks and these blocks are grouped together and implemented as a thread. The implementation can be either static or dynamic. Static means that
functional blocks are statically allocated to the processor cores so that a video stream flows through the cores, each running one or more functional blocks [19] Dynamic implementation means that the functional block threads dynamically float over a set of processor cores based on the thread scheduling decision made by the schedulers of the operating system (OS). The recent multi-core processor, such as Intel Quad Core perform dynamic implementation. [18]

- **Data Domain parallelism:**
  Data level parallelism which is inherent in H.264/AVC codec can also be used. The H.264/AVC encoder treats a video sequence as many group of pictures (GOP). Each GOP includes a number of frames. Each frame is divided into slices. Each slice is an encoding unit and is independent of other slices in the same frame. The slice can be further decomposed into a macro block (MB), which is the unit of motion estimation and entropy coding. These regions i.e. frames, slices and macro blocks are placed to parallelize the encoder [9]. The effect of parallelization can vary depending on the degree of parallelism to explore and the amount of resources to use for implementation.

![](image1)

**Fig. 7 Hierarchy of data domain decomposition [9]**

- **Frame Level Decomposition:**
  At frame-level, the input video stream is divided into GOPs. Since GOPs are usually made independent of each other, it is possible to develop a parallel architecture where a controller is in charge of distributing the GOPs among the available cores. [12]
  The number of frames that can be coded in parallel is determined by the sequence of frame types in the video. A typical sequence of frames is $I_1 B_2 B_3 P_4 B_5 B_6 B_7 B_8 B_9 P_{10}$, (where the subscript of the frame type indicates the frame's serial order). In this sequence, only three frames can be processed concurrently, with the following order of processing:
  $$\{I_1\} \rightarrow P_4 \rightarrow B_2, B_3, B_7 \rightarrow \{B_5, B_6, P_{10}\}. \quad [10]$$
  By using the following steps frame level parallelization can be done
  1. Separate the total number of frames to encode into 2 equal sets.
  2. Perform the parallel intra coding on two frames in both partitions.
  3. Perform inter coding on frame 2 and frame 17 by incorporating changes in the encoding algorithm using Open MP. Repeat for frame 3 and frame 18 and so on till all the frames are encoded, as given in the figure 8.
**Slice Level Decomposition:**
Partitioning of a frame to multiple independent slices enables parallel processing of slices. However, slicing the image and compressing each slice independently reduce the amount of spatial redundancy that can be exploited. Therefore, more slices will increase the bitrate required for a compressed video, assuming a desirable fixed quality of the compressed video. If the bit rate is kept fixed and the allowed degradation in the compressed video quality is less than 0.3dB (in terms of signal-to-noise ratio), then each picture can be divided into 4-8 slices only, resulting in a limited number of threads. [10]

**Fig.9 Simultaneous slice level and macro block level parallelism [12]**
Parallelization Implementations:

Table 4 summarizes some of the implementations performed on the H.264/AVC encoder by different approaches.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>[18]</th>
<th>[19]</th>
<th>[20]</th>
<th>[21]</th>
<th>[22]</th>
<th>[12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploited parallelism</td>
<td>TLP/DLP</td>
<td>TLP/DLP</td>
<td>DLP</td>
<td>DLP</td>
<td>TLP/DLP</td>
<td>DLP</td>
</tr>
<tr>
<td>System domain</td>
<td>Embedd ed system</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>Server</td>
</tr>
<tr>
<td>Parallelization granularity</td>
<td>MB</td>
<td>Frame/sli ce</td>
<td>Frame/M B</td>
<td>MB</td>
<td>Frame/slice/M B</td>
<td>Slice/M B</td>
</tr>
<tr>
<td>Evaluation method</td>
<td>HW/SW co-simulation</td>
<td>HW</td>
<td>Simulatio n</td>
<td>Simulatio n</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC</td>
<td>Proprietary</td>
<td>N/A</td>
<td>N/A</td>
<td>Proprietary</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of threads</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Evaluation platform</td>
<td>ARM MPCore</td>
<td>Intel Quad Xeon with HTa</td>
<td>Intel Pentium 4</td>
<td>Intel Pentium M</td>
<td>Intel Quad Xeon with HT</td>
<td>AMD 8384 quad-core</td>
</tr>
<tr>
<td>Main memory size</td>
<td>512 MB</td>
<td>2 GB</td>
<td>512 MB</td>
<td>512 MB</td>
<td>N/A</td>
<td>8 GB</td>
</tr>
<tr>
<td>Software</td>
<td>JM 13.2</td>
<td>N/A</td>
<td>JM 9.0</td>
<td>JM 10.2</td>
<td>N/A</td>
<td>JM 14.0</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.36</td>
<td>3.7 - 4.5</td>
<td>3.17</td>
<td>3.3</td>
<td>3.8 - 4.6</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Table 4: Comparison of parallelization approaches for H.264/AVC encoder [18]

The parallelization approaches by the authors [19], [20], [21], [22] improve speedup by 3-4 times for the encoder. The authors must have performed extensive research on parallelization of H.264/AVC using OpenMP to achieve speedup of around 3.7 times.

Parallel Programming Using OpenMP:

OpenMP API is an implementation of multithreading, a method of parallelization whereby the master thread forks a specified number of slave threads and a task is divided among them. [13] The threads then run concurrently, with the runtime environment allocating threads to different processors. The programming model of OpenMP is based on cooperating threads running simultaneously on multiple processors or cores. Thus, the OpenMP program begins with a main thread or master thread. Slave threads in the program are created and destroyed in a fork-Join pattern as shown in Figure 10. When the parallel construct is encountered, the initial thread, as a master thread creates a team of threads consisting of a certain number of new threads and the initial thread itself. This fork operation is performed implicitly. The program code inside the parallel construct is called as a parallel region and is executed in parallel by all threads of the team. At the end of a parallel region, there is implicit barrier synchronization, and only the master thread continues to execute after this region (implicit join operation). There can be nested parallel regions as well inside the code. [13] All OpenMP threads of a program have access to the same shared memory. Synchronization primitives have to be employed in the code in order to avoid conflicts, deadlocks, race conditions etc. The OpenMP provides various library routines for avoiding deadlocks, conflicts and race conditions. [13]
At compile time, multithreaded program code is generated based on the compiler directives. The OpenMP API supports multithreaded parallel programming in C, C++, and FORTRAN for shared memory multiprocessor [18] and various compilers have a support for OpenMP standard.
Open MP Directives:

Table 4 summarizes the important OpenMP directives.

<table>
<thead>
<tr>
<th>Pragma Directives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td>Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.</td>
</tr>
<tr>
<td>#pragma omp for</td>
<td>Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.</td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td>Shortcut combination of <strong>omp parallel</strong> and <strong>omp for</strong> pragma directives, used to define a parallel region containing a single for directive.</td>
</tr>
<tr>
<td>#pragma omp parallel</td>
<td>Work-sharing construct identifying a structured block of code that must be executed in sequential order.</td>
</tr>
<tr>
<td>#pragma omp section,</td>
<td>Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.</td>
</tr>
<tr>
<td>#pragma omp sections</td>
<td></td>
</tr>
<tr>
<td>#pragma omp parallel</td>
<td>Shortcut combination of <strong>omp parallel</strong> and <strong>omp sections</strong> pragma directives, used to define a parallel region containing a single sections directive.</td>
</tr>
<tr>
<td>#pragma omp single</td>
<td>Work-sharing construct identifying a section of code that must be run by a single available thread.</td>
</tr>
<tr>
<td>#pragma omp master</td>
<td>Synchronization construct identifying a section of code that must be run only by the master thread.</td>
</tr>
<tr>
<td>#pragma omp critical</td>
<td>Synchronization construct identifying a statement block that must be executed by a single thread at a time.</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td>Synchronizes all the threads in a parallel region</td>
</tr>
<tr>
<td>#pragma omp flush</td>
<td>Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.</td>
</tr>
<tr>
<td>#pragma omp threadprivate</td>
<td>Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.</td>
</tr>
</tbody>
</table>

Table 4 Open MP Directives [14]

Figure 11 illustrates an execution flow which is supported by the OpenMP model. The for loop is parallelized using **parallel for** directive. [18]
INTERIM RESULTS:

Codec Used: JM 18.4 [15]
Test Video Sequence: foreman_part_qcif.yuv
Format: 4:2:0 | Width: 176 | Height: 144 | Total number of frames: 300 | Number of frames encoded: 3
PSNR = 38.398 dB
Bit Rate: 333.04 Kbits/sec
Encoding Time: 3.554 sec

After applying parallelism to the code
Format: 4:2:0 | Width: 176 | Height: 144 | Total number of frames: 300 | Number of frames encoded: 3
PSNR = 38.398 dB
Bit Rate: 333.04 Kbits/sec
Encoding Time: 3.376 sec
Codec Used: JM 18.4 [15]
Test Video Sequence: news_qcif.yuv
Format: 4:2:0 | Width: 176 | Height: 144 | Total no of frames: 300 | No of frames encoded: 3
PSNR = 34.46 dB
Bit Rate: 333.24 Kbits/sec
Encoding Time: 3.037 sec

After applying parallelism to the code
Format: 4:2:0 | Width: 176 | Height: 144 | Total no of frames: 300 | No of frames encoded: 3
PSNR = 34.46 dB
Bit Rate: 333.24 Kbits/sec
Encoding Time: 3.037 sec

**Future Work:**
Frame Level Parallelism will be implemented with emphasis on decreasing encoding time for a minimum deprecation in video quality i.e PSNR and Bit Rate
REFERENCES


JM 18.4, Fraunhofer Heinrich Hertz Institute: http://iphome.hhi.de/suehring/tml/download/

V. Gajula, “Optimizing Baseline Profile in H.264/AVC video coding by parallel programming and fast intra and inter predictions”, Website: http://www-ee.uta.edu/Dip/Courses/EE5359/index.html


Websites Referred:
2. EE 5359 Projects : http://www-ee.uta.edu/Dip/Courses/EE5359/index.html
7. Open MP tutorial : https://computing.llnl.gov/tutorials/openMP/#Introduction
8. Test Sequences : http://media.xiph.org/video/derf/