Parallelization and Performance Prediction for HEVC UHD Real-time Software Decoding

Junsoo Jeong, Junchul Choi, Soonhoi Ha
Department of Computer Science and Engineering
Seoul National University
Seoul, Korea
gojun89@iris.snu.ac.kr, hinomk2@iris.snu.ac.kr, sha@snu.ac.kr

Abstract—In this paper, we present a parallelized HEVC decoder by restructuring the reference HEVC decoder, HM, as a task graph with LCU level granularity. Since the task graph reveals the potential parallelism of the algorithm explicitly, we are able to explore the wide design space of parallelization efficiently. Moreover, we propose an analytical method to estimate the throughput performance of the HEVC decoder after parallelization. The proposed method accurately predicts the decoding performance considering sophisticated parallelization techniques, based on which we obtain the required number of cores in a multi-core processor for real-time UHD resolution decoding. Through experiments, we verify the performance increase by the proposed parallelization technique and the accuracy of the performance analysis method.

Keywords— HEVC; codec; decoder; UHD; performance prediction; real-time decoding; parallelization; design space exploration

I. INTRODUCTION

The era of ultra-high definition (UHD) display is at hand owing to incessant demand for higher resolution video service. For UHD TV, HEVC (High Efficient Video Coding) (e.g. [1]) will be the de-facto video compression standard since its coding efficiency is at least twice higher than H.264 at the same level of video quality, which saves the communication bandwidth and storage space proportionally. However the complexity of HEVC decoder is known to be more than 4 times the H.264. Thus extensive efforts have been exerted recently to accomplish real-time HEVC decoding [2][3]. In this paper, we are concerned about CPU-based software HEVC decoding based on HEVC Test Model (HM), the reference implementation of HEVC.

While HM implements the major functionalities of HEVC, the performance is far from real deployment; for DucksTakeOff UHD video, it achieves 1.8 frames/second. On the other hand, it is reported that state-of-the-art techniques for software HEVC decoding achieve 31.7 frames/second [2] by exploiting both task and data parallelism as well as code optimization. The main focus of this paper is not performance but design methodology to parallelize a sequential HEVC decoder fast and efficiently.

In the proposed design methodology, we first restructure the sequential software to construct a task graph where a node represents a function module and an arc represents the dependency between the function blocks. Since we assume multi-core CPU as the target hardware platform, data transfer between function modules is made through shared memory. The task graph reveals the potential parallelism of the algorithm to exploit.

With the given task graph, we can explore the design space of parallel execution easily by varying the degree of parallelism: we vary the number of threads to generate by merging the function modules.

Moreover, we propose an analytical method to estimate the throughput performance of the HEVC decoder after parallelization. Experimental results prove that the proposed analysis method is quite accurate to predict the decoding performance, based on which we compute the required number of cores in a multi-core processor for real-time UHD resolution decoding.

The rest of the paper is organized as follows. In the next section, we give some background knowledge on the HEVC algorithm and the parallelization opportunities. In section III, we review the related work and summarize the contribution of this work. The proposed technique is explained in the subsequent chapters: software restructuring to a task graph in section IV, design space exploration in section V, and the analysis method to estimate the throughput performance in section VI and section VII. Section VIII shows the experimental results and the final section gives concluding remarks.

II. HEVC CODEC

Compared with the predecessor video coding standard, the most prominent feature of HEVC is the use of larger coding units (CUs). Instead of using a uniform size of macro-block, HEVC achieves variable size of coding units by introducing the notion of Coding Tree Unit (CTU) where a CU can be split into a quad-tree with 4 smaller CUs. The Largest CU, called as LCU (Largest Coding Unit), acts like the root node of a quad tree for CUs; the LCU size is set to 64x64 in general while a user can change the size. A frame consists of collection of LCUs: A 4K (3840x2160, or 2160p) frame consists of (60x34) LCUs.

After all CTUs are constructed, each leaf CU can be further split into PU(Prediction Unit) and TU(Transform Unit) to increase the coding efficiency by reducing the size of prediction unit and the transform coding unit. HEVC supports wider choices of PU and TU partitions for each CU. Fig. 1 shows structures of LCU, CU, PU, and TU.
The task-level flow of HEVC decoder is similar to H.264 as shown in Fig. 2. The input streams are first decoded by the CABAC (Context-Adaptive Binary Arithmetic Coding) entropy decoder module. After entropy decoding, the residual information is restored by inverse quantization (IQ) and inverse transform (IT) modules, and added to the predicted frame through motion compensation or intra-prediction. DF (Deblocking Filtering) and SAO (Sample Adaptive Offset) modules are executed to offset the artifacts caused by CU partitioning in the encoding process.

There are several options provided in the HEVC standard to help parallelization of HEVC algorithm by dividing a frame into multiple partitions that can be processed in parallel. Although those options may degrade the compression rate, they provide a way to get a parallelization opportunity in the decoding progress. Alvarez et al. [4] proposed a parallelization technique using options of WPP (Wavefront Parallel Processing) and entropy slice, and achieves maximum speed-up of 7.3 times on a multicore system with 12 cores. Since they forced to assign an entropy slice for each LCU row, entropy decoding for LCU rows can be executed in parallel.

Chi et al. [5] proposed a technique called OWF (Overlapped Wave Front) that is extended from WPP, and they obtained maximum speed-up of 8.88 times on 1600p, 10.0 times on 2160p. OWF enables us to start execution of the next frame if the reference range to the picture buffer associated with the current frame is available, not waiting for the completion of entire frame filtering. They extended their proposed parallelization technique OWF in [2] further by applying several code optimizations, and obtained real-time decoding speed which is more than 30 fps (frames per second) on UHD resolution with 4 threads on a standard multicore CPU.

Those researches based on the parallelization tools provided in HEVC standard have shown remarkable improvement in decoding speed. They have a limitation that their promised performance improvement can be obtained only when the stream is encoded with assumed encoding options.

There are other approaches without restriction of encoding option. Several works have been proposed to accelerate the speed of some part of algorithm. The technique proposed in [6] accelerates DF by splitting the execution of DF at the LCU level and executing them in parallel using GPGPU. SIMD optimization is proposed in [7] for motion compensation, DF, and integer transforms with Intel SSE 128bit length SIMD operations. They reported that average 4.16 speed-up is achieved in various resolutions. There is a work that uses picture level parallelization [3]. In their proposed technique, multiple frames are decoded simultaneously considering the reference to picture buffers of the previous frames.

Recently several corporations and research institutes have announced real-time HEVC decoder on high resolutions. Strongene announced recently a real-time HEVC decoder on PC platforms [8], and reported that they achieved more than 40 fps on UHD resolution. Fraunhofer HHI (Heinrich Hertz Institute) also announced that they developed a real-time 4K 60Hz HEVC decoder [9].

Although the state-of-the-art techniques accomplish the realtime performance on UHD resolution, it is not known how much effort they had to make. On the other hand, we focus on the design methodology to parallelize a sequential HM decoder in a systematic way. The proposed technique is complementary to other optimization techniques to improve the HEVC decoder. Even though we use the HM decoder as the reference HEVC decoder, the proposed technique can be applied to any other decoder. Hence we will not compare the decoder performance with the state-of-the art decoder. We will focus on how much performance improvement can be obtained compared with the reference HM decoder.

There is a related work that converted a sequential H.264 reference encoder to a dataflow graph [10]. To convert a sequential program to a dataflow graph, they had to flatten the function call hierarchy and remove all shared data structures since a dataflow graph does not allow shared variables except arc buffers. The proposed task graph model is different from the dataflow graph in that we allow shared variables and we do not need to flatten the function call hierarchy if not necessary. It took only a week to restructure the original HM decoder to the task graph while several months were taken in their work.

In this section, we describe how the HM decoder is restructured to a task graph. The execution flow of Fig. 2 is first partitioned into five stages that operate at the frame granularity as depicted in Fig. 3. Encoded information of each frame is read from the input bitstream and entropy decoding is performed. After entropy decoding, the frame is decompressed by referring to the current frame and previous frames. The frame is then filtered with the DF and the SAO module sequentially. The decoded frame is saved in the DPB (Decoded Frame Buffer). Note that several modules in the execution flow are clustered to the Decode stage observing that they operate at the CU level of granularity and there exists non-trivial dependency between invocations of modules.
The HEVC decoder can be partitioned into 5 stages that operate at the frame boundary.

### Task Graph Specification of the HEVC Decoder

In the proposed technique, we construct a hierarchical task graph by defining each stage as a separate task at the top level. Thus the task graph consists of five tasks: Read task, Parse task, Decode task, DF task, and SAO task. Each task can be partitioned into subtasks if parallel execution is possible by spawning multiple threads.

Since the input bitstream is from a file or network, the Read task cannot be parallelized. Also the Parse task cannot be parallelized due to the characteristics of CABAC entropy decoding unless a frame is encoded with multiple slices. On the other hand, the other tasks can be divided into subtasks which perform decompression or filtering for a portion of the entire frame.

Since the input bitstream is from a file or network, the Read task cannot be parallelized. Also the Parse task cannot be parallelized due to the characteristics of CABAC entropy decoding unless a frame is encoded with multiple slices. On the other hand, the other tasks can be divided into subtasks which perform decompression or filtering for a portion of the entire frame.

#### Fig. 3.

![HEVC decoder diagram](Image)

**Fig. 3.** The HEVC decoder can be partitioned into 5 stages that operate at the frame boundary.

#### A. Task Graph Specification of the HEVC Decoder

In the proposed technique, we construct a hierarchical task graph by defining each stage as a separate task at the top level. Thus the task graph consists of five tasks: Read task, Parse task, Decode task, DF task, and SAO task. Each task can be partitioned into subtasks if parallel execution is possible by spawning multiple threads.

Since the input bitstream is from a file or network, the Read task cannot be parallelized. Also the Parse task cannot be parallelized due to the characteristics of CABAC entropy decoding unless a frame is encoded with multiple slices. On the other hand, the other tasks can be divided into subtasks which perform decompression or filtering for a portion of the entire frame.

#### Fig. 4.

![Task graph for parallelized HEVC decoder](Image)

**Fig. 4.** Proposed task graph for parallelized HEVC decoder.

There are several possibilities to define the subtask granularity. One is to process at the leaf CU level since the CU is the basic unit of decoding process. Since the size of CU varies dynamically, however, it might be better to use a LCU as the subtask granularity. Then the number of subtasks becomes equal to the number of LCUs that compose a frame, which is 60x34 for a 4K resolution frame. Another possibility is to use the LCU line as the subtask granularity. Then a task will have 34 subtasks inside for a 4K frame. It is what the proposed technique chooses. The resultant task graph after decomposing the Decode, DF, and SAO tasks into associated subtasks is shown in Fig. 4.

In the task graph of Fig. 4, Task$_{name_k}$ indicates the subtask that processes the $k$-th LCU line of the frame for the Decode, DF, and SAO tasks. In the task graph, an arc represents the data dependency between tasks and/or subtasks. Fig. 5 shows the LCU level dependencies between the tasks. A Decode subtask refers to three LCUs that are processed in a predecessor subtask as well as a neighbour LCU that is processed in the same subtask. Hence there are dependency arcs among Decode subtasks in a vertical direction in Fig. 4. For the DF task, dependency relation shown in Fig. 5(b) implies that the $k$-th DF subtask refers to the LCU of the $(k+1)$-th Decode subtask and $(k-1)$-th DF subtask. Note that even though there is a dependency between the $k$-th DF subtask and the $k$-th Decode subtask, we can safely omit this dependency arc because it is absorbed by the dependency arc between $(k+1)$-th Decode subtask and the $k$-th DF subtask. Similarly, we add dependency arcs between the $k$-th SAO subtask and the $(k+1)$-th DF subtask as well as adjacent $(k-1)$-th SAO subtask as implied in Fig. 5(c).

#### Fig. 5.

![Dependency relation between LCU processing in the Decode, DF, and SAO tasks](Image)

**Fig. 5.** Dependency relation between LCU processing in the Decode, DF, and SAO tasks.

Fig. 4 has an additional dependency arc from $SAO_n$ to $Decode_1$ to enforce explicit synchronization at the frame boundary. Since the Decode task of the current frame may refer to any position of the reconstructed frame of the previous frame for motion compensation, we wait until the whole frame is reconstructed to start the Decode task in the current implementation. It is possible to use LCU-level synchronization between the current frame and the previous frame to reduce the block time of the Decode task for barrier synchronization. Then, the task graph should be modified to specify inter-frame dependency, which make the task graph specification more complicated. Thus we decided to add this dependency arc for simple implementation.

#### Fig. 6.

![Example schedule of subtasks in the HEVC decode task graph](Image)

**Fig. 6.** An example schedule of subtasks in the HEVC decode task graph.

It is worth stressing that a dependency arc between subtasks does not mean that a subtask can start after the completion of
the predecessor subtask. Dependency exists between LCUs while the subtask processes a sequence of LCUs in the associated LCU line. In fact, there exist two LCU processing delays between the start times of two adjacent subtasks. We illustrate a time line representation of subtask schedules in Fig. 6 that shows wave-front parallelism among subtasks.

Note that the task graph may have multiple Parse tasks even though the Parse task is not parallelizable. It specifies the frame-level parallelism unlike the other subtasks that specify LCU-line level parallelism. We may want to exploit frame-level parallelism in a pipelined fashion to satisfy the throughput constraint, since there is no dependency between entropy decoding of two frames. Fig. 7 shows the case where the Parse task becomes the performance bottleneck. In such a case, the throughput performance is limited by the tasks that should be sequentially executed. Although we cannot reduce the decoding latency of one frame, we can increase the throughput performance by pipelined execution of the Parse task.

Since the Read task takes relatively shorter time than the Parse task, multiple frames can be simultaneously entropy-decoded. Whenever the Read task reads the frame information, it sends the information to the next Parse task if available. Fig. 8 shows how entropy decoding of two frames can be overlapped to improve the throughput.

**B. Refactoring of the HM Decoder**

Once the task graph is defined, we refactor the HM decoder to create a multithreaded version associated with the task graph. In the original HM code, the function modules are called sequentially and there is no need of synchronization for LCU processing. The code template for the refactored HM decoder is shown in Fig. 9.

As shown in Fig. 9, we define a main function for each subtask that will be spawned as a thread. `WaitDependency()` function and `SignalDependency()` function accomplish synchronization at the LCU level between subtasks. Each subtask asks which LCU line to process by calling `getCU()` function.

![Diagram of Parse task can be the throughput bottleneck.](image_url)

**Fig. 7.** Parse task can be the throughput bottleneck.

**Fig. 8.** Throughput increase by instantiating multiple Parse tasks to exploit frame-level parallelism.

**Fig. 9.** The code template for the refactored HM decoder.

**V. DESIGN SPACE EXPLORATION OF PARALLELIZATION**

In this section, we use concrete examples to explain the design space exploration step. We use the HM 11.0 [11] reference software for the baseline decoder. The target hardware platform used in our experiments is described in Table I.

**TABLE I. SYSTEM HARDWARE SPECIFICATION**

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>RAM</th>
<th>HDD</th>
<th>OS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual</td>
<td>Xeon E5 2640(2.5GHz, 6 Core, 12 Thread)</td>
<td>32GB DDR3 1,066MHz</td>
<td>SSD 128GB</td>
<td>Windows 7 Enterprise</td>
<td>Parallelized HM 11.0</td>
</tr>
<tr>
<td>Dual</td>
<td>Xeon E5 2640(2.5GHz, 6 Core, 12 Thread)</td>
<td>32GB DDR3 1,066MHz</td>
<td>SSD 128GB</td>
<td>Windows 7 Enterprise</td>
<td>Parallelized HM 11.0</td>
</tr>
</tbody>
</table>

Before applying parallelization technique, we first apply SIMD parallelization for performance improvement, which is commonly used to accelerate the HEVC decoder. This result is described in Table II where we use five 3840x2160 UHD video streams that are obtained from Xiph.org web-site [12]. We measure the time (in seconds) to decode 150 frames for each video stream. From the experimental results, we observe that SIMD optimization achieves about double speed-up, from minimum 1.45 to maximum 2.35. Increasing QP results in higher speed.

**TABLE II. SPEED UP WITH SIMD OPTIMIZATION (TIME IN SECONDS)**

<table>
<thead>
<tr>
<th>Video stream</th>
<th>QP</th>
<th>Base (s)</th>
<th>SIMD (s)</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crowd</td>
<td>QP22</td>
<td>121.837</td>
<td>72.957</td>
<td>1.670</td>
</tr>
<tr>
<td></td>
<td>QP32</td>
<td>77.375</td>
<td>37.767</td>
<td>2.049</td>
</tr>
<tr>
<td></td>
<td>QP37</td>
<td>69.439</td>
<td>32.275</td>
<td>2.151</td>
</tr>
<tr>
<td>DucksTakeOff</td>
<td>QP22</td>
<td>191.243</td>
<td>131.605</td>
<td>1.453</td>
</tr>
<tr>
<td></td>
<td>QP32</td>
<td>83.358</td>
<td>39.079</td>
<td>2.133</td>
</tr>
<tr>
<td></td>
<td>QP37</td>
<td>70.510</td>
<td>31.678</td>
<td>2.226</td>
</tr>
<tr>
<td>InToTree</td>
<td>QP22</td>
<td>128.209</td>
<td>75.760</td>
<td>1.692</td>
</tr>
<tr>
<td></td>
<td>QP32</td>
<td>57.293</td>
<td>25.298</td>
<td>2.290</td>
</tr>
<tr>
<td></td>
<td>QP37</td>
<td>52.623</td>
<td>22.427</td>
<td>2.346</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>QP22</td>
<td>160.239</td>
<td>100.530</td>
<td>1.594</td>
</tr>
<tr>
<td></td>
<td>QP32</td>
<td>52.205</td>
<td>23.162</td>
<td>2.254</td>
</tr>
<tr>
<td></td>
<td>QP37</td>
<td>47.683</td>
<td>20.887</td>
<td>2.283</td>
</tr>
<tr>
<td>ParkJoy</td>
<td>QP22</td>
<td>137.194</td>
<td>80.688</td>
<td>1.700</td>
</tr>
<tr>
<td></td>
<td>QP32</td>
<td>79.781</td>
<td>38.613</td>
<td>2.066</td>
</tr>
<tr>
<td></td>
<td>QP37</td>
<td>69.383</td>
<td>31.959</td>
<td>2.171</td>
</tr>
</tbody>
</table>
The main advantage of using the task graph for parallelization is that we can explore the design space of parallelization easily by changing the mapping between the tasks and cores. The schedule diagram depicted in Fig. 7, for instance, is the case where 4 cores are allocated to each of the Decode, DF, and SAO tasks while the Parse task and the Read task are mapped to other cores. In case N cores are allocated to the Decode task that consists of 34 subtasks, \[
\left\lfloor \frac{34}{N} \right\rfloor \text{ or } \left\lceil \frac{34}{N} \right\rceil \text{ subtasks are mapped to each core. There are two implementation options. One is to generate 34 threads and assign them to cores. The other is to generate as many threads as the number of cores by merging multiple subtasks into a single thread. The former choice should pay more scheduling and context switching overhead in a core while the latter is likely to pay more synchronization overhead to process multiple LCU lines sequentially. Since each subtask is defined separately in the refactored HM decoder, merging multiple subtasks into a single thread can be done easily.}
\]

One axis of the design space is the number of threads generated for each task. In the target hardware, we can use 24 logical cores (12 physical cores \(\times 2\)). Fig. 10 shows how much speed up we can gain by varying the number of Decode threads. The number of DF and SAO threads is set to the half of the Decode threads. In this experiment, we do not specify the mapping of threads to cores: the OS will map the threads dynamically. As shown in the figure, we can get a linear speed-up up to 16 Decoder threads, regardless of the video streams. And the maximum speed-up is about 15.

Another axis of the design space is the number of Parse task, which accounts for frame-level parallelism. As explained in the previous section, entropy decoding can be the performance bottleneck and parallel entropy decoding for multiple frames may be required. However, if we assign too many cores for the Parse task, the system may be underutilized. Hence a proper number of Parse task should be explored in the system design. Fig. 11 shows the execution time for each sequence changing the number of Parse task from 1 to 4, while we generate the maximum number of threads for the other tasks. The result shows that throughput performance is almost converged after the number of Parse task is more than 2.

The last axis we explore is the mapping of threads to cores. In the Xeon 2 CPU system used in our experiments, the mapping of the subtask affects the performance because the memory access delay is not uniform. If two tasks that access the same memory address frequently are mapped to different CPUs, then memory access delay degrades the overall performance. When designing such a system, design space exploration for task mapping should be considered.

We performed an experiment to evaluate the performance variation depending on the mapping information. For this experiment, we define four different mapping configurations as summarized in Table III.

**TABLE III. MAPPING CONFIGURATION**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration 1</td>
<td>Threads are not fixed on specific core</td>
<td></td>
</tr>
<tr>
<td>Configuration 2</td>
<td>Read, Parse, Decode</td>
<td>DF, SAO</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>Read, DF, SAO</td>
<td>Parse, Decode</td>
</tr>
<tr>
<td>Configuration 4</td>
<td>Read, Parse, DF, SAO</td>
<td>Decode</td>
</tr>
</tbody>
</table>

Fig. 12 shows the execution time for each mapping configuration while the number of Parse task is 4 and the numbers of Decode task, DF task, and SAO task are 34 each. It shows that configuration 2 shows the worst performance since adjacent sub-tasks are mapped to different CPUs.
VI. BASELINE PERFORMANCE ANALYSIS METHOD

In this section, we explain the proposed performance analysis method to determine how many cores to assign to each task for real-time processing assuming that the proposed parallelization technique is used. At first we introduce the baseline performance estimation formulas assuming a simple scheduling model of tasks. In the next section, we improve the analysis formula by introducing compensation factors that bridge the gap between the baseline model and the real situation.

To analyse the throughput performance of the HEVC decoder after parallelization, we first obtain the execution profile of each task. While the proposed technique is generally applicable, we explain the technique assuming a specific target hardware platform and a specific throughput requirement, 30 frames per second. Remind that the target hardware platform used in our experiments is summarized in Table I.

For performance profiling, we use five 3840x2160 UHD video streams that are encoded with QP 32 in the random access profile as shown in the first column of Table IV. Since the Parse task is a sequential task, we measure the average execution time of the Parse task as a whole during 150 frame decoding. On the other hand, we use the average execution time of a LCU for other tasks in 33 msec on average. Since the execution time of the Read task is very short and one thread is sufficient for real-time decoding, we omit the Read task for brevity of explanation.

We denote the number of cores assigned to task \( k \) as \( N_k \) and the profiled execution time associated with task \( k \) as \( E_k \). Then we can compute the minimum number of processors that the Parse task requires for real-time processing as follows:

\[
\frac{E_{\text{Parse}}}{N_{\text{Parse}}} \leq 33.33
\]  

(1)

From this inequality, we determine that two copies of the Parse task are necessary for three out of five video streams of Table IV while only one thread works for the other two video streams.

To analyse the Decode, DF, and SAO tasks, we have to consider parallelization of those tasks and complex dependency between threads. Remind that the parallelization unit of those tasks are LCU lines that are assigned to each core in the round robin fashion. An UHD frame has 34 lines and each line has 60 LCUs. Suppose that \( N_{\text{decode}} \) cores are assigned to the Decode task. Then the total execution time to process all lines will be as follows if there is no dependency between lines:

\[
\left[ \frac{34}{N_{\text{Decode}}} \right] \times 60E_{\text{Decode}}
\]  

(2)

Since there is a dependency between two adjacent lines, however, we have to account for the delay caused by this dependency, which is equal to two LCU delays between LCU lines. Equation (2) represents the execution time that the first core will take. The other cores will experience the dependency delay if they have extra lines to process after the first core. The number of extra lines after the first core processes the last LCU line assigned becomes.

\[
\left( 33 - \left[ \frac{33}{N_{\text{Decode}}} \right] \times N_{\text{Decode}} \right)
\]  

(3)

Combining two formulas (2) and (3) for the Decode task, we can obtain the total execution time as follows:

\[
\left[ \frac{34}{N_{\text{Decode}}} \right] \times 60E_{\text{Decode}} + \left( 33 - \left[ \frac{33}{N_{\text{Decode}}} \right] \times \right) \times \frac{N_{\text{Decode}}}{2E_{\text{Decode}}}
\]  

(4)

For the DF and SAO tasks, we have to consider the overlapped execution with the Decode task as illustrated in Fig. 6. From the profiled information of Table IV, we observe that the combined execution time of the DF and SAO tasks is smaller than the Decoder execution time. Therefore the execution time of the DF and the SAO tasks can be completely hidden except a few LCUs at the tail of the execution schedule: what we need to compute is the execution time required for the DF task after the Decode task finishes the last LCU line. Consulting the dependency between the Decode task and the DF task, shown in Fig. 5 (b), we infer that the DF task consumes two LCU delays after the last Decoder execution. Similarly the

TABLE IV. PROFILE INFORMATION OF TASKS (AVERAGE EXECUTION TIME IN MS)

<table>
<thead>
<tr>
<th>Example</th>
<th>Parse</th>
<th>Decode</th>
<th>DF</th>
<th>SAO</th>
</tr>
</thead>
<tbody>
<tr>
<td>DucksTakeOff</td>
<td>41.199</td>
<td>0.0701</td>
<td>0.0202</td>
<td>0.0072</td>
</tr>
<tr>
<td>CrowdRun</td>
<td>38.289</td>
<td>0.0674</td>
<td>0.0211</td>
<td>0.0066</td>
</tr>
<tr>
<td>InToTree</td>
<td>24.401</td>
<td>0.0439</td>
<td>0.0137</td>
<td>0.0061</td>
</tr>
<tr>
<td>ParkJoy</td>
<td>43.605</td>
<td>0.0670</td>
<td>0.0203</td>
<td>0.0072</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>23.368</td>
<td>0.0390</td>
<td>0.0126</td>
<td>0.0059</td>
</tr>
</tbody>
</table>

The profiled information reveals that the Parse task cannot satisfy the throughput requirement if executed sequentially, being the performance bottleneck. But the parse task cannot be accelerated by parallel processing. Instead we are able to pipeline the execution since entropy decoding has no dependencies between frames.

![Fig. 13. Simple run-time schedule model of tasks.](image)

The proposed analysis method is first to determine how many cores are needed for each task, based on the simple model of run time task schedule. The same number of threads as the number of assigned cores is generated for each task. Fig. 13 shows an example of the simple schedule model assuming three copies of the Parse tasks are executed in a pipelined fashion. To
dependency between the DF task and the SAO task, only 4 LCU delays are added after the completion of the DF task until the SAO task finishes. Fig. 14 illustrates the modified schedule diagram based on the aforementioned inference.

![Diagram](Image)

Fig. 14. Latency from filter tasks after the completion of decode task.

In short, we can get the total execution time of the Decode, DF, and SAO tasks by just adding $2E_{DF} + 4E_{SAO}$ to the total execution time of the Decode task as follows, which should be smaller than $33.3$ ms to satisfy the throughput constraint.

$$\left\lceil \frac{34}{N_{Decode}} \right\rceil 60E_{Decode} + \left(33 - \left\lceil \frac{33}{N_{Decode}} \right\rceil \right) 2E_{Decode} + 2E_{DF} + 4E_{SAO} \leq 33.33 \quad (5)$$

Now we figure out how many cores are required for the DF and the SAO tasks. Since the combined execution time of the DF and SAO tasks is smaller than half of the Decoder execution time, it may use the smaller number of cores proportionally. Thus the number of cores assigned to the DF and the SAO tasks combined, denoted as $N_{Filtering}^{\text{DF,SAO}}$, is set to half of the cores assigned to the Decode task. It means that

$$N_{Filtering} = \frac{N_{Decode}}{2} \quad (6)$$

VII. COMPENSATION FACTORS

Since the schedule model assumed in the baseline analysis formulas ignores all overheads and additional delays caused by data dependency, there is a significant error between the estimated performance and the measured performance. The total execution time estimated from the baseline model is highly underestimated. In this section, we improve the performance model by introducing a few compensation factors to consider the extra delays at the very high level.

A. Execution time variation factor ($\alpha$)

In the baseline performance model, we assume that all LCU processing times are the same to the average execution time. In reality, however, LCU processing delay varies widely depending on the scene characteristics. If the previous line takes long LCU processing time, the current line should wait due to dependency relation. There are several synchronization methods to realize dependency relation between LCUs while we use semaphores in this work. Such a non-uniform distribution of LCU processing delays is a major source of extra delay.

In addition, concurrent accesses to the shared memory incur arbitration delay. Since all cores access the same shared memory in a CPU, the actual LCU processing delay becomes larger than the profiled information that is measured from sequential execution of the HEVC decoder.

There may be other causes to make the average execution time per LCU larger than the profiled one. We group these overheads as the execution time variation overhead as a whole since it is impossible to measure how much delay is attributed to each overhead individually. Instead we account for the gross overhead by defining the execution time variation factor ($\alpha$) that is multiplied to the total execution time as follows;

$$Time_{Pred} = \left(\left\lceil \frac{34}{N_{Decode}} \right\rceil 60E_{Decode} + \left(33 - \left\lceil \frac{33}{N_{Decode}} \right\rceil \right) \times \frac{33}{N_{Decode}} \right) 2E_{Decode} + 2E_{DF} + 4E_{SAO} \times \alpha \quad (7)$$

Note that we define a single factor even though the execution time variation effect will be different among Decode, DF, and SAO tasks to minimize the number of compensation factors. The compensation factors are all determined empirically to minimize the gap between the estimated performance and the actual performance.

B. Additive factor ($\beta$)

In the baseline performance formula, the total execution time is proportional to the execution time of computation tasks. But there are extra delays that do not depend on the execution time of computation tasks. For instance, we need to consider the operating system overhead to schedule the threads at run-time. Since the synchronization is accomplished by semaphores, we should pay extra overhead for blocking and waking-up threads. This overhead is not dependent on the computation time but on the synchronization count.

We also have to consider the thread scheduling overhead of the multi-core OS, which increases as the total number of threads increases. In our parallelization scheme, the DF task and the SAO task share the cores. It means that we need to consider the context switching overhead between a DF thread and a SAO thread. In case the context switching overhead is not negligible, the total sum of the DF and the SAO processing delay for a LCU may be larger than the Decoder processing delay for a LCU.

To sum up this kind of overhead that is not proportional to the computation time, we introduce an additive compensation factor ($\beta$) as follows:

$$Time_{Pred} = \left(\left\lceil \frac{34}{N_{Decode}} \right\rceil 60E_{Decode} + \left(33 - \left\lceil \frac{33}{N_{Decode}} \right\rceil \right) \times \frac{33}{N_{Decode}} \right) 2E_{Decode} + 2E_{DF} + 4E_{SAO} + \beta \quad (8)$$

As discussed above, the additive factor depends on the number of threads, the number of synchronization activities, and the context switching overhead. To take into account of this complex dependency, we define a reference factor, $\beta_0$, and formulate the additive factor as the relative value to the reference factor as follows:

$$\beta = \left(\frac{\left\lceil \frac{34}{N_{base, Deco}} \right\rceil 60E_{base, Deco}}{\left\lceil \frac{34}{N_{target, Deco}} \right\rceil 60E_{target, Deco}} \right) \beta_0 \quad (9)$$
The reference factor is associated with a specific implementation, which will be discussed in the next section. As this formula shows, the additive factor is proportional to the number of threads and inverse proportional to the Decoder execution time per LCU. This formula is confirmed empirically as discussed in the next section.

C. Remote memory access overhead ($\gamma$)

In case multiple CPUs are used in the target hardware platform, we need to consider the communication overhead between CPUs during the computation. The Xeon platform used in our experiments consists of two CPUs that have six hyper-threaded cores inside as shown in Table I. If the total number of cores assigned to the tasks is greater than 12, we have to use both CPUs and take into account the data transfer delay between CPUs. In the Xeon system used in our experiments, data transfer between two CPUs is performed through QPI (Quick Path Interconnect) as shown Fig. 15.

![Data transfer diagram of dual CPU](image)

To estimate the time overhead incurred from inter-CPU communication, we ran a preliminary experiment in which all tasks run sequentially on a single core except the Decode task. We measured the total execution time for the following two cases: the Decoder thread is assigned to the same CPU ($T_{shared}$) or to the other CPU ($T_{dist}$). The difference between these two, $T_{dist} - T_{shared}$, is regarded as the overall remote memory access overhead, which is the bidirectional communication between two CPUs. Since this overhead should be considered as the one-way communication overhead from the Parse task to the Decode task in our performance formula, we divide the overall overhead by 2, the number of frames, and the number of LCUs in a frame to obtain the remote memory access overhead per LCU, denoted as $\gamma_0$, as follows:

$$\gamma_0 = \frac{\left(\frac{T_{dist} - T_{shared}}{2}\right)}{\text{The number of LCUs in frame}}$$

(10)

For 150 frames of UHD (3840x2160) resolution video streams with 64x64 LCU size, $\gamma$ is equal to

$$\gamma_0 = \frac{\left(\frac{T_{dist} - T_{shared}}{2}\right)}{60 \times 34}$$

(11)

Note that this overhead depends on the volume of data to be communicated. Since the Parse task time is roughly proportional to the data volume, we adjust the remote memory access overhead with the ratio of the Parse task time between the target video sequence $E_{target}^\text{Parse}$ and the reference sequence $E_{base}^\text{Parse}$ as follows:

$$\gamma = \frac{E_{target}^\text{Parse}}{E_{base}^\text{Parse}} \gamma_0$$

(12)

Since the remote memory access overhead is applied to the Decode task only, the modified execution time becomes

$$Time_{pred} = \left(\frac{34}{N_{Decode}}\right) 60 \left(E_{Decode} + \frac{\gamma}{\alpha}\right) + \left(33 - \left\lfloor\frac{33}{N_{Decode}}\right\rfloor\right) \times N_{Decode} \times 2 \left(E_{Decode} + \frac{\gamma}{\alpha}\right) + 2E_{DF} + 4E_{SAO}$$

(13)

D. Final formula for the estimated performance.

Considering all three compensation factors to the baseline performance, we obtain the following formula for the estimated performance after parallelization:

$$Time_{pred} = \left(\frac{34}{N_{Decode}}\right) 60 \left(E_{Decode} + \frac{\gamma}{\alpha}\right) + \left(33 - \left\lfloor\frac{33}{N_{Decode}}\right\rfloor\right) \times N_{Decode} \times 2 \left(E_{Decode} + \frac{\gamma}{\alpha}\right) + 2E_{DF} + 4E_{SAO} + \beta$$

(14)

We determine the actual compensation factors by regression analysis. For regression analysis, we select a set of example video streams, $S_{video}$. And we measure the actual execution times by varying the number of cores assigned to the Decode task. The number of cores assigned to the DF and the SAO tasks is determined by (6). Then we explore all possible values of $\alpha$, $\beta_0$, and $\gamma_0$ to search for the optimal values that minimize the worst-case difference ratio between the estimated performance by (14) and the measured performance. It can be summarized as follows:

$$\max_{N \in S_{Decode} \times S_{video}} \left\{\frac{FPS_{pred}(N,S,\alpha,\beta_0,\gamma_0) - FPS_{real}(N,S)}{FPS_{real}(N,S)}\right\}$$

(15)

where $S_{Decode}$ means the set of number of cores assigned to the Decode task, and $FPS_{pred}$ and $FPS_{real}$ represent the predicted and measured throughput performances respectively in terms of frames per second. Note that we can use another objective such as minimizing the maximum error for the absolute frame rate.

VIII. EXPERIMENTS

The first set of experiments was conducted to determine the compensation factors in the proposed performance formula. Among five video streams listed in Table IV, we select the following three for regression analysis: $S_{video} = \{\text{DucksTakeOff, ParkJoy, OldTownCross}\}$. Note that the
decoding performance of the original HM decoder varies significantly for those training video streams. We use the other two video streams, CrowdRun and InToTree, for verifying the accuracy of the proposed analysis formula. In addition, five 4K video streams are down-sampled to full HD (1080p) resolutions and down-sampled video streams are also used in our experiments.

For regression analysis, we vary the number of cores assigned to the Decode task from 4 to 12 as shown in the second column of Table VII: \( N_{\text{Decode}} \in \{4,6,8,10,12\} \). The number of cores assigned to the DF and the SAO tasks is half of the cores assigned to the Decode task. And the cores assigned to the Parse task is determined to be the minimum number satisfying (1). One core is assigned to the Read task. We assign the Decode task to a different CPU in case the number of cores assigned to Decode task is greater than 4. It means that we do not account for the remote memory access factor when \( N_{\text{Decode}} \) is 4. As discussed in section VII.B and VII.C, we have to select specific implementations to determine the reference additive factor \( \beta \) and the reference factor \( \gamma \). The selected implementations for each resolution are as follows:

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Reference stream for ( \beta )</th>
<th>( N_{\text{Decode}} )</th>
<th>Reference stream for ( \gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2160p</td>
<td>ParkJoy</td>
<td>4</td>
<td>OldTownCross</td>
</tr>
<tr>
<td>1080p</td>
<td>DucksTakeOff</td>
<td>6</td>
<td>OldTownCross</td>
</tr>
</tbody>
</table>

Since the performance formula (14) is not linear, we perform exhaustive search over the three dimensional space of three factors after confining the range of those factors. For instance, the execution time variation factor will be larger than 1 and smaller than a reasonable upper bound, say 3.

Table VI shows the resultant compensation factors. It shows that the execution time of the Decode, DF, and SAO task will be lengthened by about 1.9 times due to execution time variation.

**TABLE VI. COMPENSATION FACTORS FOUND BY REGRESSION ANALYSIS**

<table>
<thead>
<tr>
<th>Compensation factor</th>
<th>( \alpha )</th>
<th>( \beta_0 )</th>
<th>( \gamma_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2160p</td>
<td>1.895</td>
<td>1.384</td>
<td>0.00142</td>
</tr>
<tr>
<td>1080p</td>
<td>1.869</td>
<td>0.528</td>
<td>0.00796</td>
</tr>
</tbody>
</table>

Table VII displays the difference between the measured performance and the estimated performance based on the compensation factors obtained through the regression analysis, in terms of the throughput performance, frames per second. From Table VII, the worst-case FPS error between the predicted performance and the measured performance is less than 6% for 4K video streams and 12% for 1080p video streams over all configurations; the error for 1080p is about twice larger than that for 4K video streams. To verify the accuracy of the proposed performance formula, we compare the performance with the other two video streams, CrowdRun and InToTree.

The verification result is displayed in Table VIII that shows the similar accuracy level over all variations of \( N_{\text{Decode}} \). The prediction error is quite random; even for the same video stream, the error fluctuates randomly. The result shows no trend or dependency on any condition of experiments, except the case of full HD resolution. For both regression result and verification result of full HD resolution, our prediction method shows relatively high FPS errors when \( N_{\text{Decode}} \) is 4 or 8. In those cases, only one core is assigned one more LCU row than the other cores since the number of LCU rows in full HD resolution is 17. It means that the predicted FPS is overly compensated by \( \alpha \) factor. We could improve the correctness by a few percent more by decreasing the influence of \( \alpha \) factor for those cases. Without this complicated adjustment, the results prove that the proposed performance analysis formula is quite robust despite its simplicity.

**TABLE VII. DIFFERENCES BETWEEN PREDICTED FPS AND ACTUAL FPS OF EXAMPLES FOR COMPENSATION FACTOR COMPUTATION**

<table>
<thead>
<tr>
<th>Example</th>
<th>( N_{\text{Decode}} )</th>
<th>( \text{FPS}_{\text{real}} )</th>
<th>( \text{FPS}_{\text{pred}} )</th>
<th>( \text{FPS} ) Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>DucksTakeOff</td>
<td>4</td>
<td>13.9</td>
<td>38.2</td>
<td>2.4%</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>4</td>
<td>22.2</td>
<td>65.2</td>
<td>5.8%</td>
</tr>
<tr>
<td>ParkJoy</td>
<td>6</td>
<td>19.6</td>
<td>48.4</td>
<td>1.2%</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>6</td>
<td>31.0</td>
<td>84.3</td>
<td>3.2%</td>
</tr>
<tr>
<td>DucksTakeOff</td>
<td>6</td>
<td>23.8</td>
<td>55.4</td>
<td>3.2%</td>
</tr>
<tr>
<td>ParkJoy</td>
<td>8</td>
<td>36.0</td>
<td>96.2</td>
<td>2.4%</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>10</td>
<td>29.4</td>
<td>52.9</td>
<td>3.3%</td>
</tr>
<tr>
<td>DucksTakeOff</td>
<td>10</td>
<td>40.6</td>
<td>102.2</td>
<td>2.6%</td>
</tr>
<tr>
<td>ParkJoy</td>
<td>12</td>
<td>32.1</td>
<td>64.3</td>
<td>0.8%</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>12</td>
<td>43.0</td>
<td>110.8</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

**TABLE VIII. DIFFERENCES BETWEEN PREDICTED FPS AND ACTUAL FPS OF EXAMPLES FOR VERIFICATION**

<table>
<thead>
<tr>
<th>Example</th>
<th>( N_{\text{Decode}} )</th>
<th>( \text{FPS}_{\text{real}} )</th>
<th>( \text{FPS}_{\text{pred}} )</th>
<th>( \text{FPS} ) Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrowdRun</td>
<td>4</td>
<td>14.3</td>
<td>34.0</td>
<td>1.2%</td>
</tr>
<tr>
<td>InToTree</td>
<td>4</td>
<td>20.4</td>
<td>61.9</td>
<td>3.6%</td>
</tr>
<tr>
<td>CrowdRun</td>
<td>6</td>
<td>20.0</td>
<td>43.7</td>
<td>0.3%</td>
</tr>
<tr>
<td>InToTree</td>
<td>6</td>
<td>28.8</td>
<td>80.6</td>
<td>1.4%</td>
</tr>
<tr>
<td>CrowdRun</td>
<td>8</td>
<td>24.8</td>
<td>50.8</td>
<td>3.7%</td>
</tr>
<tr>
<td>InToTree</td>
<td>8</td>
<td>34.1</td>
<td>92.5</td>
<td>0.4%</td>
</tr>
<tr>
<td>CrowdRun</td>
<td>10</td>
<td>30.0</td>
<td>55.1</td>
<td>5.4%</td>
</tr>
<tr>
<td>InToTree</td>
<td>10</td>
<td>39.2</td>
<td>98.3</td>
<td>0.7%</td>
</tr>
<tr>
<td>CrowdRun</td>
<td>12</td>
<td>34.0</td>
<td>58.9</td>
<td>2.1%</td>
</tr>
<tr>
<td>InToTree</td>
<td>12</td>
<td>41.7</td>
<td>108.2</td>
<td>3.3%</td>
</tr>
</tbody>
</table>

The next set of experiments is to find out the minimum number of cores assigned to each task to achieve real-time 4K and full HD 30Hz HEVC decoding. Table IX shows the result that is obtained from the proposed performance formula. The last column confirms that the real performance is truly greater than the throughput constraint, 30 fps.
TABLE IX. PREDICTED MINIMUM NUMBER OF CORES TO DECODE EACH EXAMPLE OVER 30 FPS

<table>
<thead>
<tr>
<th>Example</th>
<th>(N_{\text{Parse}})</th>
<th>(N_{\text{Decode}})</th>
<th>(N_{\text{Filtering}})</th>
<th>(\text{FPS}_{\text{pred}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>DucksTakeOff</td>
<td>2</td>
<td>12</td>
<td>6</td>
<td>32.4</td>
</tr>
<tr>
<td>CrowdRun</td>
<td>1080</td>
<td>1</td>
<td>4</td>
<td>30.0</td>
</tr>
<tr>
<td>InToTree</td>
<td>2160</td>
<td>2</td>
<td>12</td>
<td>33.3</td>
</tr>
<tr>
<td>ParkJoy</td>
<td>1080</td>
<td>1</td>
<td>4</td>
<td>32.2</td>
</tr>
<tr>
<td>OldTownCross</td>
<td>2160</td>
<td>1</td>
<td>6</td>
<td>32.0</td>
</tr>
</tbody>
</table>

IX. CONCLUSION

In this paper, we propose a design methodology to parallelize a sequential reference HEVC decoder. In the proposed methodology, we first restructure the sequential decoder to make a task graph. At the top level of the task graph, there are five tasks that operate at the frame boundary. For the tasks that can be parallelized at the LCU level, we make a set of subtasks that process a single LCU line. In the resultant task graph, dependency between subtasks is specified by arcs. We also specify the frame-level parallelism for the Parse task to achieve pipelining to satisfy the throughput constraint. We could explore the design space of parallelization easily by changing the mapping of threads to cores.

In addition, we devise an analytical method to estimate the performance after parallelization. The baseline model is established by drawing a simple and ideal schedule diagram of task execution. On top of the baseline performance model, we add compensation factors to account for extra delays caused by execution delay variation, OS-related overhead, remote memory access overhead, and so on. The proposed performance formula has three compensation factors. These compensation factors are determined by regression analysis with three video streams. And the performance formula is verified with the other factors are determined by regression analysis with three video streams. Experimental results show that the proposed performance formula is accurate enough to find out the number of cores necessary to achieve real-time HEVC decoding.

Note that we changed the structure of the HM decoder without manual optimization except some well-known SIMD optimization. It took us only a few weeks to restructure the HM decoder to achieve real-time processing once the proposed design methodology is established. To achieve more speed up, we need more elaborate optimization to the HM source code itself, which is left as a future work.

ACKNOWLEDGMENT

This research was supported by the MSIP(Ministry of Science, ICT & Future Planning), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2014-H0301-14-1018) and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2013R1A2A2A01067907). The ICT at Seoul National University provides research facilities for this study.

REFERENCES