J. Vis. Commun. Image R. 23 (2012) 742–752

Contents lists available at SciVerse ScienceDirect

J. Vis. Commun. Image R.

journal homepage: www.elsevier.com/locate/jvci

Batch-pipelining for multicore H.264 decoding

Tang-Hsun Tu, Chih-Wen Hsueh *, Ja-Ling Wu

Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei 10617, Taiwan, ROC

ARTICLE INFO

Article history:
Received 1 September 2011
Accepted 26 March 2012
Available online 9 April 2012

Keywords:
Parallelization
Pipelining
Batch
H.264
HEVC
Multicore
Synchronization
Optimization

ABSTRACT

Pipelining has been applied in many area to improve system performance by overlapping executions of hardware or software computing stages. However, direct pipelining for H.264 decoding is difficult because video bitstreams are encoded with lots of dependencies and little parallelism is left to be explored. Fortunately, pure software pipelining can still be applied to H.264 decoding at macroblock level with reasonable performance gain. However, the pipeline stages might need to synchronize with each other and incur lots of extra overhead. For optimized decoders, the overhead is relatively more significant and software pipelining might lead to negative performance gain. We first group multiple stages into larger batches and execute these batches concurrently, called *batch-pipelining*, to explore more parallelism on multicore systems. Experimental results show that it can speed the decoding up to 89% and achieve up to 259 and 69 frames per second for resolution 720P and 1080P, respectively, on a 4-core ×86 machine over an optimized H.264 decoder. Because of its flexibility, batch-pipelining can be applied to not only H.264 but also many similar applications, such as the next-generation video coding: high efficiency video coding. Therefore, we believe the batch-pipelining mechanism creates a new effective direction for software codec development.

1. Introduction

H.264 is a very popular standard for high-quality video compression of low bitrates [1,2]. However, its high complexity requires large computing power for encoding and decoding. In H.264, video stream contains a series of groups of pictures (GOPs), each GOP may contain more than one frames, each frame may contain more than one slices, and each slice may contain thousands of macroblocks (MBs). Decoding an MB might rely on the information from many other MBs in the same frame or other frames. Because of the on-line decoding requirement, tight dependencies of function components, and complex data structures, it is difficult to explore the decoding parallelism in H.264. Moreover, considering the overhead of parallelization, synchronization and merging of partial decoding results, pursuing parallelism might not always lead to positive performance gain. Fortunately, we find that there exists a systematic software approach to parallelize H.264 decoding on multicore systems.

To parallelize and speed up H.264 decoding, one common idea is to execute independent computations concurrently, where a computation might consist of executing multiple decoding components. Data and function (or task) decompositions are two frequently used approaches to construct independent computations. Since GOP-level parallelism requires a lot of memory and results in high latency and frame-level parallelism needs the support of corresponding encoders [3], we do not consider using them in this paper. Because a slice in H.264 might be an independent data unit that can be decoded on itself, slice-level parallelism [4] is intuitive. However, for less bitrate, there is usually only one slice encoded in a frame and MB-level parallelism can be applied directly with multiple slices. Therefore, we also do not further discuss slice-level parallelism and focus on MB-level parallelism.

For data decomposition, at MB level, wavefront [5,6] is a technique to decode several MBs concurrently by rearranging the independent MBs in a 2-dimensional manner. However, even if the MBs can be decoded concurrently, it might incur a lot of overhead due to the small granularity of synchronization among MBs [7]. Actually, because of the dependencies of a bitstream itself, the entropy coding component needs to be decoded sequentially in practice even using the wavefront technique. Actually, the wavefront technique is still not fully scalable for decoding. Meenderinck et al. [3] also extend the 2D wavefront to 3D-Wave for further parallelism. However, there were no detail implementation results and only the analysis of highly scalable parallelism was provided. Actually, the synchronization overhead might be very significant in real implementation and negative performance gain is possible especially applied on an already optimized decoder, such as that the decoding time of a MB is in microsecond level. Wang et al. [7] proposed a scalable parallelizing mechanism for deblocking filter on four cores, called *PD* in this paper. The PD mechanism might only

* Corresponding author.
E-mail addresses: d98944004@ntu.edu.tw (T.-H. Tu), cwhsueh@csie.ntu.edu.tw (C.-W. Hsueh), wjl@cmlab.csie.ntu.edu.tw (J.-L. Wu).

1047-3203/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jvcir.2012.03.009

be suitable for deblocking filter and is optimized on frame level. However, it can be used together with batch-pipelining.

For function decomposition, Sihn et al. [8] assigned motion compensation and deblocking filter to different synergistic processor elements on Cell Broadband Engine [9–11]. However, it is difficult to port to other multicore systems and might have bottleneck of scalability on specific hardware, such as the power processor element. Seitner et al. [12] divided several functions into two parts on dual cores respectively and developed a simulator for H.264 onto various hardware architectures. Since the number of functions in H.264 is limited and function decomposition is difficult in general, function decomposition usually requires special hardware assistance.

Traditionally, it is intuitive to parallelize the H.264 decoding using pipelining at MB level or on functions. Schöffmann et al. [13] developed a parallel approach using MB pipelining on different decoding components. It compared the pipelining approach with other multi-threading and multi-slicing approaches on different multicore systems and concluded that functional partitioning and macroblock pipelining is a good alternative for the evaluated videos. However, in spite of the large variation on performance of different videos, the synchronization overhead among MBs might become relatively large, especially when the pipelining stages are executed much faster with better hardware support or software optimization.

Previous works mostly focused on one of the two decompositions mentioned above and had big synchronization overhead or required special hardware support on multicore systems [13]. In our observation, some components in MB decoding process still can be done concurrently, such as motion compensation, inverse discrete cosine transform, and inverse quantization. However, the dependencies in entropy decoding is inevitable. Therefore, we adopt both function decomposition and data decomposition at the same time to explore more parallelism.

We modify the JM reference software [14] (an unoptimized decoder) to verify our batch-pipelining mechanism as function decomposition and apply the mechanism with different optimizations to an optimized H.264 decoder with PD [15,16] to measure performance gain. Since the decoding is multi-threading, we also apply a UDispatch [17] mechanism to verify the flexibility of batch-pipelining and further improve performance in terms of operating system issues, where the UDispatch can dispatch threads to specific cores by users in user mode to optimize multi-threading. With the optimizations above, although the resultant decoder is much faster than the real-time requirement, our mechanism is flexible and can be easily assumed by slower hardware platforms to provide more cost-effective solutions or applied in higher resolutions. Furthermore, we provide general guidelines of using BP with examples for more applications, such as an extra mode in H.264 [18–20] and the next-generation video coding: High Efficiency Video Coding (HEVC or H.265) [21–23].

This paper is an extension of our poster in DCC 2010 [24]. The rest of this paper is organized as follows. The next section introduces some background of related technologies. In Section 3, we present and anaylze the batch-pipelining mechanism. Different optimizations are discussed in Section 4. Experimental results are shown in Section 5. Section 6 introduces the potential usage of batch-pipelining mechanism on similar applications. Section 7 concludes this writing and points out the directions of our future work.

2. Background

In this section, we briefly introduce the H.264 architecture and the thread anomalies in multi-threading.

2.1. H.264 Architecture

In H.264, a frame is encoded in one or more slices, and a slice consists of a header and a sequence of MBs. Each MB is of size 16×16 pixels and stores the coefficients which are entropy coded. The MBs are classified into intra MBs and inter (P or B) MBs. Intra MBs are predicted from previously coded data within the same slice (intra prediction). The MBs predicted only from the past frames are called inter P MBs and the MBs predicted from both the past and the future frames are called inter B MBs. There are three common types of slices, i.e. I, P, and B slices. I slice consists only of intra MBs, P slice consists of intra MBs and inter P MBs, and B slice consists of intra MBs and inter B MBs. As shown in Fig. 1, each MB follows the same decoding processes. The main decoding components are Entropy Decoding (ED), Inverse Quantization (IQ), Inverse Discrete Cosine Transform (IDCT), Intra Prediction (IP), Motion Compensation (MC), and Deblocking Filter (DF) [2,25].

Since the decoding components are constructed sequentially with lots of dependencies, it is difficult to apply multi-threading to explore parallelism on multicore systems, especially when several components are already optimized altogether. Therefore, before we present our batch pipelining mechanism, it is necessary to discuss the dependencies of decoding an MB. Suppose there is only one slice in a frame. Fig. 2 shows the dependencies of a current MB ($M_{i,j}$), where $M_{i,j}$ stands for the MB at row i and column j in a frame with two-dimensional representation. To decode an $M_{i,j}$, as shown by the solid lines with arrow, it might refer to much information from previous neighboring MBs. Since the bitstream is sequential in nature and coded in variable length, arguments and coefficients for decoding can not be retrieved until the previous MBs finish the bitstream parsing, as shown in dashed lines with arrow in Fig. 2. Therefore, MBs are usually decoded in a predefined scan order, e.g. raster scan.

Doing intra prediction in an intra MB might need the information of pixels from neighboring MBs, such as pixel value and intra modes, hence, it might need to wait until previous MBs finish decoding. Actually, for optimization, intra mode prediction is usually implemented with entropy coding. However, unlike an intra MB, an inter MB does not need to refer to pixels from neighboring MBs. Instead of intra prediction, an inter MB does motion compensation. Motion compensation will refer to relative pixels by motion vector decoded from previous frames. Note that the motion vector prediction is also usually implemented with entropy coding for optimization. In other words, motion compensation is independent among MBs in the same slice, and it consumes relative large percentage of time in processing an MB, e.g. up to 36.7% in our experimental bitstreams. Furthermore, IQ and IDCT of each MB are also independent and can be executed concurrently.

As described above, basically, the components MC, IDCT and IQ of inter MBs can be parallelized and decoded first if related reference frames are done and intra MBs have to be decoded after the neighboring MBs are processed. However, even though these components can be executed concurrently, it might incur large synchronization overhead if the granularity is too small. Moreover, considering inter slices (P or B), since usually the more intra MBs, the less compression, hence, the percentage of intra MBs in an inter slice is usually not much. Therefore, our idea is to parallelize these independent components systematically in inter slices "*in batch*" while leaving decoding of the I slices intact and intra MBs afterwards. Theoretically, these I slices and intra MBs can be decoded with other techniques, such as wavefront, for more parallelism. However, the problem of high synchronization overhead needs to be solved first for these techniques to be effective. This is not the focus of this paper and is left as our future work.