A cache-aware strategy for H.264 decoding on multi-processor architectures

Abstract. H.264 is one of the most commonly used formats for the recording, compression and distribution of video. Encoders and decoders for the H.264 standard are widely in demand, and efficient strategies for enhancing their performance have been areas of active research. With the proliferation of many core architectures in the embedded community, there has been a trend towards parallelizing implementations of encoders and decoders. In this paper, we present a run time heuristic which exploits macro-block level parallelism and efficient scheduling inside a H.264 decoder to reduce the number of cache misses and improve the processor utilization. Experiments on standard benchmarks show a significant speed-up over contemporary strategies proposed in literature.

1 Introduction

H.264/MPEG-4 Part 10 or AVC (Advanced Video Coding) is one of the most common video formats in recent times. H.264 provides much better compression ratios than most other video formats such as H.263 and MPEG-2. Encoders and decoders for the H.264 standard are widely in demand, and efficient strategies for enhancing their performance have been areas of active research.

Security applications typically involve widespread deployment of H.264. In the security context, videos are mostly intra-coded, i.e. all existent motion dependencies are within the same frame. Intra-coded videos have therefore been a subject of active research in both the academic and industrial setting.

With the proliferation of many core architectures in the embedded community, there has been a trend towards parallelizing implementations of encoders and decoders. In general, these proposals have focused on efficient exploitation of inherent parallelism (at the frame level, slice level or macro-block level) in the video structure with an aim of achieving better decode performance by load balancing and distribution of workload between available processors, while honouring video dependency constraints as applicable.

For intra-coded videos, strategies exploiting macro-block level parallelism have been found to be more successful in general. The problem in this setting essentially is to identify the macro-block dependency structure inside a H.264 slice / frame, process the macro-blocks in parallel (honouring dependencies as applicable) on the available processors in a multi-core setting, with an objective to minimize the end-to-end decode time. Both static and dynamic macro-block scheduling strategies have been proposed. Static scheduling strategies in general assume worst case dependency patterns among the constituent macro-blocks and often, equal processing times, irrespective of their types.

This paper has two important considerations. A static scheduling approach, which assumes uniform macro-block processing times, leads to poor processor utilization. In reality, macro-block processing times vary depending on the inputs and the dependency structure. Hence, it is possible to improve the effective processor utilization by adopting a dynamic scheduling approach that assigns macro-blocks to free processors as soon as they are ready, as opposed to a static solution that would normally schedule at pre-defined intervals. In addition to improving utilization, we also show that the effective speed-up obtained crucially depends on the cache interaction of the decode strategy in a multi-processor setting with a hierarchical (private L1, shared L2, DRAM) memory structure. Many of the existing decode strategies often do not consider the cache misses resulting from cache oblivious selection of the macro-blocks to be processed, which in turn leads to significant slowdown in decoder performance due to frequent accesses to the lower and slower memory levels.

Our work has two proposals for harnessing the effective power of parallel computation in a multi-core setting. On one hand, we propose a cache-aware [5] scheduling strategy to minimize the number of cache misses, by carefully selecting the macro-blocks to be considered next, keeping in view the chance of a macro-block it depends on, getting evicted from the cache due to capacity or conflict misses. On the other hand, we attempt to improve the number of macro-blocks available for processing at every time point, which in turn implies better processor utilization and hence, improvement in speedup.

We implemented our schedule heuristic and evaluated it on a number of standard benchmarks. Experiments have shown significant speed up as compared to methods that currently exist.

2 Background and Related work

A H.264 video [1] consists of a sequence of frames. A frame is an array of luma samples and two corresponding arrays of chroma samples. Each frame is further divided into spatial units called slices. A slice consists of blocks of 16 x 16 pixels, known as macro-blocks (MB). A macro-block contains type information describing the choice of methods used to code the macro-block and prediction information such as intra prediction mode information and coded residual data. Within a macro-block, luma samples may be coded as blocks of 4 x 4, 8 x 8 or 16 x 16 pixels. Chroma samples are commonly coded as blocks of 8 x 8 pixels.

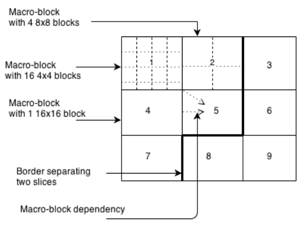
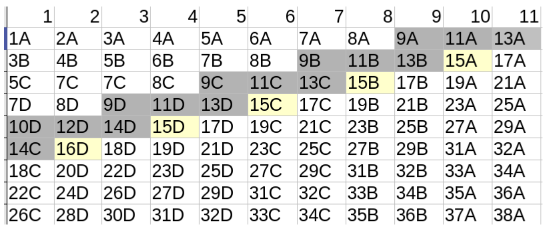


Fig. 1. A 3x3 H.264 frame



1	2	3	4	5	6	7	8	9	10	11
1A	2A	3A	4A	5A	6A	7A	8A	9A	11A	13A
3B	4B	5B	6B	7B	8B	9B	11B	13B	15A	17A
5C	7C	7C	8C	9C	11C	13C	15B	17B	19A	21A
7D	8D	9D	11D	13D	15C	17C	19B	21B	23A	25A
10D	12D	14D	15D	17D	19C	21C	23B	25B	27A	29A
14C	16D	18D	19D	21D	23C	25C	27B	29B	31A	32A
18C	20D	22D	23D	25D	27C	29C	31B	32B	33A	34A
22C	24D	26D	27D	29D	31C	32C	33B	34B	35A	36A
26C	28D	30D	31D	32D	33C	34C	35B	36B	37A	38A

Fig. 2. 2-d wavefront in action

Reconstruction is an important step in the decode of a H.264 video frame. Reconstruction of a decoded macro-block involves obtaining the data from neigh-

bouring macro-blocks based on which motion prediction had been made by the encoder. This cannot be done independently, but only after fetching data of neighbouring macro-blocks. In an intra-coded video frame, all dependencies are in the same frame of video. In addition, a MB includes a variable amount of residual information that cannot be inferred from previous MBs.
Attempts to parallelize the reconstruction step have been done at frame-level, slice-level and macroblock-level. At the frame level, different frames are decoded by different cores. However, this leads to too much pressure on the memory system. Since there are no dependencies among macro-blocks across slices, slice-level parallelism places much lower demands on the memory system. However, since the number and dimensions of the slices are variable, it leads to poor load-balancing. Thus, macro-block level parallelism is the most commonly used technique to implement parallel H.264 decoders. [3] presents an excellent survey of approaches to H.264 decoder parallelization proposed in literature.

The 2-d wavefront approach [3] for parallel H.264 decoding exploits macro-block level parallelism and computes a static schedule and a processor allocation strategy. This has been quite successful in practice and proved to be an efficient solution in a multi-processor setting. Figure 2 shows a snapshot of this method in action on a frame with 99 MBs, in a 4 processor setup. A, B, C, and D are the 4 processors. Each MB is labelled with the processor to which it is assigned. Also, the number labelling each MB denotes the cycle at which the MB will be processed. For example, the topmost leftmost MB (labelled as 1A) will be processed in the first cycle on processor A. The next MB to its right, being dependent on it (1A), cannot start till it finishes, and hence, is assigned to time unit 2 in the same processor. The entire frame is processed in 38 time units on 4 processors, in the schedule as mentioned as labels on the MBs in Figure 2. In order to improve scalability, this has been further extended to 3-dimensional approach (3-d wavefront), where two or more frames are decoded simultaneously depending on the number of idle cores in the multiprocessor system [2].

3 Motivation and Objectives

Our work has several important considerations that makes it different from those proposed in literature. Static approaches to parallelize decoding [3], in general, assume, a regular dependency structure for a MB and equal processing times, i.e. each MB is dependent on all its four neighbours [1] (top left, top, top right, and side left), depending on which of these are actually present according to its position (the top row MBs excepting the leftmost one, for example, only have left dependency edges). However, in reality, there is a lot of input-dependent variation, and in practice, the dependencies vary across MBs. In effect, a MB can actually turn out to depend on one / two / three / all /none of its neighbours, a fact that can lead to improvement in decode performance in a parallel setting. This motivates a dynamic run-time schedule strategy.

Secondly, static methods often schedule MBs at uniform intervals on all cores, assuming all MBs have equal processing times. This is not true in H.264. This forces some of the cores to remain idle. For example, in Figure 2, if processor A finishes processing MB 15A early, it has to wait for other cores. We assume variable processing times, and also attempt to improve on processor utilization through our MB selection method.

Finally, most of the techniques for parallel decoding do not consider cache misses. Consider the working of the wavefront strategy as shown in Figure 2. The gray shaded MBs are the ones needed in the processor cache for processing the yellow ones (since worst case 4-neighbour dependency is assumed). For example, for processing the MB marked 15A in core A, the processor needs to load the MBs 9A, 11A, 13A and 13B in the local cache. Since 13B is being processed in core B, this means, we need to fetch its data present in core B of L1 through L2. An important consideration that can reduce cache misses is to avoid the same MB being reloaded into cache. This can be done by giving priority to MBs whose parents have a chance of being expelled due the cache replacement policy. In our schedule heuristic, we keep this under consideration.

4 Cache-Aware Heuristic

In this section, we present the details of our MB scheduling and processor allocation strategy in a multi-processor setting with a private L1 cache, a shared L2 cache and DRAM. We assume the following about the processor model.

– Every cache uses Least Recently Used (LRU) replacement policy.
– All caches are assumed to follow multi-level inclusion policy, i.e., if some data item is present in L1 cache, then it is also present in L2 cache and DRAM. Similar policy is also followed for L2 cache. However, the data present in DRAM may be obsolete.
– L1 cache is assumed to use write-through policy. L2 cache is, however, assumed to use write-back policy.
– Write-invalidate is used at L2 cache to ensure memory consistency. In other words, when some data is written to the cache, that data is immediately invalidated at all L1 caches, and the updated data is brought in from L2.
– Bus arbitration of L2 cache is done on First Come First Serve (FCFS) basis.

We further assume that all frames are fully accommodated in the main memory. Thus, there is no need for disk access at any point of time. We now define the concept of *cache flush time*, a key element of our schedule heuristic.

Definition 1 *The cache flush time τ of a node u is the number of cache misses for which the node will be present in the cache.*

Since we assume a LRU replacement policy, the cache flush times can be obtained from the LRU counters in a modern architecture. The MB scheduling problem is modelled as a scheduling problem on a task graph, where the MBs form the nodes and a directed edge between two MBs depicts the dependency relation (from a MB to its dependent MBs). We call such a task graph that is used to model this problem as a frame task graph. A node (i.e. a MB) of the frame task graph is labelled with three components.

- Macro-block type: Indicates which of 4x4 / 8x8 / 16x16 is present.
- Position of macro-block dependency in memory hierarchy: For each dependency of a macro-block, the position in the memory hierarchy (L1 cache, L2 cache or DRAM) where it is present.
- Cache flush time: For each dependency of a macro-block, the τ value.

4.1 Proposed Algorithm

Our heuristic uses simple priority-based scheduling. It involves having a ready queue A of macro-blocks that can be accessed by any processor. A processor that becomes idle accesses the ready queue, calculates the priority value of each macro-block present in the queue and then chooses the one for decoding having the highest priority. Priority of a macro-block depends on the position of the macro-blocks in memory, their cache flush times and the number of available macro-blocks in the ready queue. While assigning priority, we take into account the following factors:

- Minimum cache flush time of L2 among all nodes (CFT_{L2}): We examine each node in A whose one or more parents reside in the L2 cache. For each node, we compute the parent with minimum τ value (has the most chance of being evicted), and find that node in A whose parent has the highest chance of being evicted, i.e. the minimum τ value – we call this CFT_{L2}. We have a threshold value t_{L2}, which we compare with CFT_{L2}. If CFT_{L2} is greater than the threshold value, we then know that there are no nodes in the L2 cache that are at risk of being flushed. So, we are then free to choose which node will be selected depending on other constraints. On the other hand, if CFT_{L2} is less than or equal to the threshold value, then we need to ensure that those nodes which are about to be flushed are accessed quickly by the decoder. The scheduler, therefore, needs to schedule the children of nodes about to be flushed from L2 cache as quickly as possible.
- Minimum cache flush time of L1 among all nodes (CFT_{L1}): As with L2, we similarly use the threshold value t_{L1} to compare with CFT_{L2}.
- Memory access time of L1 and L2: We know that L1 has a much lower memory access time as compared to L2. When there is no danger of any required node to be flushed from L1 or L2, we assign higher priority to nodes whose parents are in L1. Higher the number of parents in L1, higher is a node's priority.
- Number of free nodes in available list: One of our objectives is to ensure that idle time of processor cores is reduced. So we aim to have enough nodes in our available list so that processor cores remain busy in processing nodes. So, when fewer nodes are present in the available list, we give more priority to nodes that have more outgoing edges.
- For source nodes i.e. nodes with no incoming edges, we give weightage to the number of neighbours that have been processed. This ensures adjacent nodes are processed at approximately similar times to maintain temporal locality.

Algorithm 1 CalculatePriority

```
1: $p_{data} \leftarrow CalculateDataPriority(A)$
2: $p_{available} \leftarrow CalculateAvailablePriority(A)$
3: for all node n ∈ A do
4:     $p[n] \leftarrow \frac{1}{p_{data}} + \frac{p_{\text{parallel}}}{x}$ // x is the number of available MB nodes in $A$
5: end for
```

Algorithm 2 CalculateDataPriority

```
1: $Q \leftarrow \underset{n \in A}{\cup} parent[n]$
2: $CFT_{L2} \leftarrow \min_{n \in Q \cap L2} \tau_{L2}[n]$; $CFT_{L1} \leftarrow \min_{n \in Q \cap L1} \tau_{L1}[n]$; $cachePriority \leftarrow none$
3: if $CFT_{L2} < t_{L2}$ then
4:     $cachePriority \leftarrow L2$
5: else if $CFT_{L1} < t_{L1}$ then
6:     $cachePriority \leftarrow L1$
7: end if
8: if $cachePriority! = none$ then
9:     for all node n ∈ A do
10:        $p_{data}[n] \leftarrow \min_{p \in parent[n]} \tau_{cachePriority}[p]$
11:    end for
12: else
13:    for all node n ∈ A  do
14:        $p_{data}[n] \leftarrow (l * |nodes[L2] \cap parent[n]| + |nodes[L1] \cap parent[n]|)/\,|parent[n]|$
15:    end for
16: end if
17: return  $p_{data}$
```

Algorithm 3 CalculateAvailablePriority

```
1: for all node n ∈ A do
2:     $\text{p}_{\text{outgoing}}[\text{n}] \leftarrow$ no of outgoing edges from current available node
3:     if $n$ is source then
4:         $\text{p}_{\text{source}}[\text{n}] \leftarrow$ no of spatial neighbours of this node processed
5:         $\text{p}_{\text{parallel}}[\text{n}] \leftarrow k * p_{source}[n] + (1 - k) * p_{outgoing}[n]$
6:     else
7:         $\text{p}_{\text{parallel}}[\text{n}] \leftarrow p_{outgoing}$
8:     end if
9: end for
10:
11: return  $p_{parallel}$
```

Algorithm 1 uses simple priority-based scheduling exploiting the factors discussed. There are two distinct priority components – p_{data} and $p_{parallel}$. p_{data} is calculated based on the position of a node's parents in the cache. A node whose parent is in danger of being evicted from the L2 cache (obtained by comparing the parent node's CFT with t_{L2}) is given priority. If no parent is in danger of being evicted from L2 cache, we then perform the same process for L1 cache.

The reason behind avoiding misses from L2 cache getting higher priority is that data that is expelled from L2 cache takes a much longer time to bring back compared to L1 cache. If we find that no parent node is in danger of being evicted from either L1 or from L2, we then calculate the average priority of each node, assuming that L1 has a higher priority than L2. This is done to ensure that memory access times are minimized. The other component, $p_{parallel}$, is used to assign priority depending on the number of outgoing edges. Higher the number of child nodes, greater is the value of $p_{parallel}$. k denotes a scaling factor. For source nodes, this also includes how many of its spatial neighbours have been processed. The total priority, p, is then calculated by assigning weightage to the two priority factors. The node having the highest priority is then processed by the core that executed the scheduler. If there is a tie among two processor who started the priority computation at the same time and ended up selecting the same MB, we resolve it arbitrarily in favour of any of the processors.

We now explain the working of our algorithm on Figure 2. We assume L1 can accommodate 2 macroblocks, and L2 can accommodate 8 macroblocks. Let us assume the macroblock labelled 15A is dependent on 13B and 11A, while 15B is dependent on 11B and 13C. The first processor is idle and calls the scheduler. 11A is in L1 with τ value 1, 13B is in L2 with τ 5, 11B is in L2 with τ 4, whereas 13C is in L2 with τ value 8. The threshold values are 2 for L1 and 4 for L2. No MB in L2 is in the danger of being evicted (comparing τ values with the threshold), and hence, we look at nodes whose parents are in L1. Then, 15A is scheduled to be processed next using our algorithm, since its parent 11A will be flushed out of L1 cache before the threshold. If, however, 11A had a τ value of 2, whereas 13C had a τ value 3, then 15B would have been selected.

5 Experimental Results

We implemented an architecture simulator to evaluate our proposed scheduling strategy in a multi-processor setting. We assume a cache block size of 32 bytes, and fetching from DRAM in bursts of 256 bytes. We ran our method on a set of standard test videos. We selected a set of 12 video clips which are widely used across the digital video domain for benchmarking the decoding, encoding and other pre/post processing algorithms. All the clips are of HD (720p) resolution and contain 150 frames. Most of them have high details and large variations within a frame; hence the intra coded frames are expected to have very good distribution of intra prediction modes (and hence the dependencies among macroblocks). Our goal was to generate an intra-only stream suite which has a good variation of dependency relations among the macroblocks.

We used the Joint Model reference software (JM) [4] version 17.2 for encoding the contents in order to generate our test suite consisting of all only intra-coded videos. The streams were then parsed with our in-house H.264 decoder and the task-graphs were generated. For our experiments, we used the value 4 for both the cache thresholds and 0.5 for k. Simulations were done on a 2GHz machine.

We compared the speedup obtained by our method over the wavefront approach. We implemented the wavefront method using the algorithm discussed in [3] and our schedule heuristic as described in Algorithm 1. Results are shown in Table 1. Each row of the table represents the speedup obtained for a particular video clip when 4, 8, 16, 32 and 64 processors are present. For most of the videos, our algorithm offers improvements (speedup > 1) or comparable performances. For the ones, for which our method is slower than the wavefront method, the runtime overhead (graph extraction, dependency structure building, processing time calculation, scheduling overhead) turns out to slow down the decode process, in comparison to a simple-minded static strategy.

Video	4-proc.	8-proc.	16-proc.	32-proc.	64-proc.
bus	1.12	1.08	1.08	0.9	1.05
crowdrun	1.52	1.57	1.68	1.43	1.72
duckstakeoff	1.78	1.98	2.31	2.11	2.94
intotree	1.31	1.24	1.19	0.95	1.43
night	1.25	1.24	1.31	1.039	1.105
oldtowncross	1.26	1.16	1.1	0.26	0.93
parkjoy	1.79	1.76	1.87	1.64	2.1
parkrun	1.39	1.26	1.33	1.08	1.12
shields	1.07	1.19	1.27	0.95	0.93
shuttlestart	1.12	1.06	0.94	0.72	0.68
stockholm	1.26	1.25	1.29	0.99	0.9

Table 1. Speedup over wavefront

6 Conclusion

In this paper, we present a cache aware schedule heuristic for improving the speedup of a H.264 decode algorithm in a multi-processor setting. We believe that a cache-oblivious strategy can benefit tremendously using the proposed modifications. We are currently looking at extending similar ideas to the other compression standards.

References

1. Richardson, Iain E. *The H. 264 advanced video compression standard* Wiley, 2011.
2. Azevedo, A. et. al. "A highly scalable parallel implementation of H. 264." Transactions on High-Performance Embedded Architectures and Compilers IV. Springer Berlin Heidelberg, 2011. 111-134.
3. Juurlink, B. et. al., Scalable Parallel Programming Applied to H.264/AVC Decoding, Springer 2012 (ISBN: 978-1-4614-2229-7)
4. H.264/AVC JM software, http://iphome.hhi.de/suehring/tml/
5. Guan, N. et. al. Cache-aware scheduling and analysis for multicores, In Proceedings of EMSOFT 2009, pp. 245-254