EFFICIENT PARALLELIZATION OF H.264 DECODING WITH MACRO BLOCK LEVEL SCHEDULING

Jike Chong, Nadathur Satish, Bryan Catanzaro, Kaushik Ravindran, Kurt Keutzer

EECS Department, University of California, Berkeley, USA

ABSTRACT

The H.264 decoder has a sequential, control intensive front end that makes it difficult to leverage the potential performance of emerging manycore processors. Preparsing is a functional parallelization technique to resolve this front end bottleneck. However, the resulting parallel macro block (MB) rendering tasks have highly input-dependent execution times and precedence constraints, which make them difficult to schedule efficiently on manycore processors. To address these issues, we propose a two step approach: (i) a custom preparsing technique to resolve control dependencies in the input stream and expose MB level data parallelism, (ii) an MB level scheduling technique to allocate and load balance MB rendering tasks. The run time MB level scheduling increases the efficiency of parallel execution in the rest of the H.264 decoder, providing 60% speedup over greedy dynamic scheduling and 9-15% speedup over static compile time scheduling for more than four processors. The preparsing technique coupled with run time MB level scheduling enables a potential 7x speedup for H.264 decoding.

1. INTRODUCTION AND MOTIVATION

According to Moore's law, the number of transistors on a single die has been doubling every 18-24 months. This trend is expected to continue until at least 2011 [1]. Principally due to power limitations, single processor performance is not expected to scale with Moore's Law. As a result, industry is looking to improve performance by doubling the number of processor cores integrated on a chip. With 4-8 core general purpose processors already available, chips with hundreds of general purpose cores are expected to appear soon [2].

Techniques for single thread performance optimization no longer adequately utilize the parallelism on these manycore platforms. Instruction-level parallelism provides diminishing returns, and branch/value prediction can consume a prohibitive amount of energy due to incorrect speculations [3].

Static scheduling at compile time can maximize application performance on manycore processors without incurring speculation energy and performance penalty at run time. Static scheduling explores multiple levels of concurrency in an application, and optimally maps application tasks onto a parallel platform according to some cost function [4]. Static scheduling algorithms typically assume an application description in the form of a directed acyclic precedence task graph. The tasks are atomic computations, and the edges represent precedence constraints between the tasks. The approaches range from fast heuristics [4] [5] to exact methods that provide a known bound on the optimal result [6].

For static scheduling to give efficient results, an accurate estimate of task execution time, and a deterministic set of precedence constraints must be available at compile time independent of application inputs. Application domains such as DSP and networking satisfy these requirements quite naturally. In contrast, in modern applications such as the H.264 video codec, control flow increasingly dominates and results in two highly input dependent execution characteristics:

1. Input dependent variable task execution times
2. Input dependent precedence constraints

Ha, et al. [7] proposed a systematic approach to profile a dynamic construct at compile time, where a dynamic construct can be an input dependent task with variable execution time. This approach assumes that only a small portion of the application is input dependent. It is not effective when input dependent behavior is dominant. For H.264 rendering, *all* tasks, as well as *all* precedence constraints between tasks, exhibit input dependent behavior. Such behavior makes static scheduling approaches, including dynamic construct profiling, impractical.

With application knowledge and input data information at run time, input dependent task execution times can be estimated and input dependent precedence constraints can be resolved. There is an opportunity to schedule tasks dynamically at run time to achieve more efficient execution of the application.

Dynamic scheduling has been studied extensively. For example, Cilk [8] is a multithreaded dynamic scheduling framework that is very effective for load balancing programs with tree style computations. However, the forking of child tasks is not suited for task graphs such as H.264 rendering, which contains complex reconvergent paths in the task graph.

In this paper, we propose a strategy for dynamically scheduling general task graphs with input dependent characteristics at run time. With the availability of a large number of cores on chip, we propose dedicating some cores for running

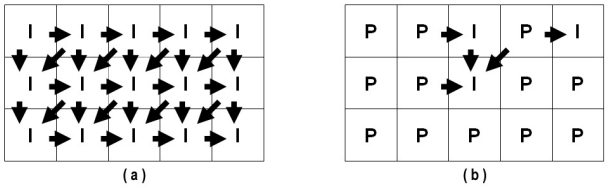

Fig. 1. Macroblock dependency graphs: (a) I frame with dense dependencies (b) P frame with sparse dependencies

more sophisticated scheduling algorithms with the aim to increase the efficiency of the entire system. We also evaluate the viability of three scheduling algorithms for this problem. We use H.264 decoding as a case study and explore the trade-offs of dedicating computation resources for scheduling to achieve more efficient use of a parallel platform.

2. H.264 PARALLELIZATION TECHNIQUES

H.264, or MPEG-Part 10 is an advanced video codec proposed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG). The main goal of H.264 is to provide good video quality with significantly lower bit rate, while maintaining a practical degree of complexity that is implementable with current technology.

2.1. H.264 macro blocks and frames

H.264 is a block based video compression standard. It separates a video segment into frames and divides each frame into macroblocks (MBs) of 16x16 pixels. There are two main types of MBs: Intra MBs (I MBs), and Predicted MBs (P MBs). The I MBs exploit spatial locality by referencing previously decoded MBs in the same frame of video, and the P MBs exploit temporal locality by referencing MBs from a previously decoded frame. Both types of MBs also include varying amount of residual information that cannot be inferred from previous MBs. I MBs have tight data dependencies within the same frame that are difficult to parallelize. In contrast, P MBs do not depend on other MBs within the same frame and are easily parallelized.

A video frame containing only I MBs is an intra frame (I frame), in which the dependencies between MBs form a regular mesh as illustrated in Figure 1(a). However, there is still significant input dependent variation to exploit in these frames, since the amount of work to decode an I MB depends on the residual information. Our work estimates the work necessary to decode an I MB heuristically using the compressed description of the MB residual information, which can be computed in the preparse stage. This information is then exploited in our scheduler to yield a more optimal decoding schedule for the I frame.

Frames containing both I MBs and P MBs are called predicted frames (P frames). These frames have highly variable, typically rather sparse MB level dependency graphs as illustrated in Figure 1(b). As with I MB, our work estimates the

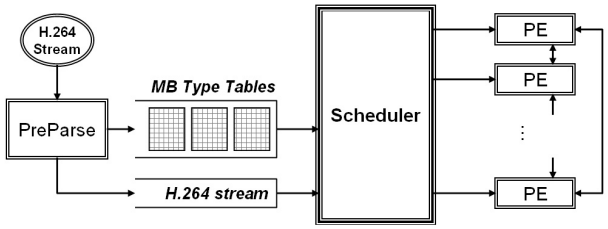

Fig. 2. Preparsing architecture of the H.264 decoder

decoding time for both I MBs and P MBs using the residual heuristic, and then creates a schedule for the frame which takes advantage of the unique inter MB level dependency information.

2.2. Parallelization challenges and techniques

For each video frame, H.264 goes through a three stages: parse, render, and filter. The parsing front end is control dominated and highly sequential. The rendering step involves motion compensation as well as intra prediction and has significant opportunities for parallelism, but these opportunities are highly input dependent. The filtering back end is obviously parallel and is not a parallelization bottleneck. We concentrate on the parallelization of the first two steps in this paper.

The parsing front end is the parallelization bottleneck that every H.264 decoder implementation faces. At the bit-level, variable-length encoded modes must be interpreted to select lookup tables for decoding the following bits. At the MB level, texture prediction modes and motion vectors (MVs) are inferred from neighboring MBs. As an implementation often starts from a piece of sequential software reference code with shared data structures, multiprocessor implementations of H.264 decode often degenerate to sequential execution with application specific accelerators for computation intensive kernels, an example of which is [9].

We have developed a preparsing architecture (shown in figure 2) to resolve the parsing bottleneck. The preparse stage uses data from the encoded stream and buffers a small amount of recently decoded prediction modes/MVs from neighboring MBs for its own use. By preparsing the stream, we determine the offset of each MB in the video stream, its type, and its texture prediction mode or MV. This step resolves all bit level and MB level dependencies in the encoded H.264 video stream. To schedule a frame, properties of each MB are passed to the scheduling stage, so that the scheduling stage can independently schedule each MB and access the MB information in the video stream in a *random access* pattern.

We do not pass all information extracted from the encoded stream in the preparse stage to the following stages. Doing so would unnecessarily load the communication channel between the preparse and schedule stages. Instead, we pass the original input stream from the preparse stage to the schedule stage. The final decompression occurs at the individual processing elements allocated to render the MB.

The preparsing architecture exposes task level parallelism

by functionally pipelining the preparse, schedule, and render stages in H.264 decoding, allowing computation in all three stages to overlap in time. It also exposes data level parallelism in the rendering stage. In the next section, we explore techniques to increase parallelism in the render stage.

3. SCHEDULING APPROACHES

The preparsing architecture exposes significant data level parallelism in the rendering of MBs. These rendering tasks, however, have highly input-dependent execution times and precedence constraints. The variability of task execution times makes efficient allocation and scheduling of these rendering tasks important.

We explore three different approaches in this paper: greedy dynamic scheduling, static compile time scheduling, and run time MB level scheduling. We believe run time MB level scheduling can provide significant performance improvement for H.264 decode.

The greedy dynamic scheduling looks at only one MB at a time from the head of the H.264 encoded stream in scan order and greedily assigns it to the next available processor without considering the MB's type and execution time. This approach models a simple extension of a single processor sequential scan order decoding implementation to the multiprocessor platforms. The algorithm lacks the visibility of a larger scheduling window and can not effectively parallelize MB rendering in frames with dense MB level dependencies.

Static compile time scheduling lacks information about input dependent precedence constraints and execution times. It thus has to respect all possible precedence constraints between MBs and assume equal execution time for all MBs. One example of this type of implementation can be found in [10]. The result is that all frames, regardless of sparsity of dependencies, will have the same schedule. This is overly conservative for P-frames, where usually only 1-5% of the precedence constraints are required. Further, execution times of MBs can vary as much as a factor of 10 within the same frame, leading to load imbalance among the processors.

Our run time MB level scheduling approach leverages the information accumulated in preparse to estimate input dependent execution times and resolve precedence constraints. It uses MB types and encoded stream sizes to estimate MB execution times and resolve precedence constraints. With more accurate information, it is able to construct a customized schedule for each frame of input data. This way, the input-dependent properties of tasks in the frame are taken into to account to schedule and load-balance on the parallel platform.

4. EXPERIMENTAL SETUP

The experiment is based on a basic H.264 decoder implementation by Fiedler et al [11]. Our test video stream has 360 frames at a resolution of 320x144. The video contains one I frame for every nine P frames. We collected the decoding time of each MB on a PentiumM 1.5GHz machine with QueryPerformanceCounters at micro second resolution. These decoding times are used as real execution traces for scheduling on multiprocessors with one to nine processor. We assume that the computation time dominate the parallel execution time and memory contention will not significantly affect parallel execution time.

Using each of the three scheduling approaches, the stream is simulated on one to nine processors. For the greedy dynamic scheduling approach, the execution trace is scheduled with a simple simulator to compute the video segment's multiprocessor run time.

For static compile time scheduling, an optimal static schedule is generated by mapping a conservative precedence constraint graph assuming unit execution times for all tasks (onto one to nine processors). The same schedule is used for all frames in the manitu segment to compute multiprocessor run time.

For run time MB level scheduling, a task graph is generated for each frame based on a linearly-interpolated estimate of execution times and reduced precedence constraints based on MB types in the frame. The task graphs are individually scheduled using the DLS algorithm [4]. As illustrated in figure 2, the scheduling process uses a separate processing element at run time. The real execution time traces are used in simulating the schedules for individual frames to compute multiprocessor run time.

The Dynamic Level Scheduling (DLS) algorithm we used for run time MB level scheduling at run time has a complexity of $O(N^2P)$, where N is the number of tasks and P is the number of processors. We have optimized our DLS implementation such that the performance bottleneck of the decoder still lies in the rendering stage.

5. RESULTS

We analyzed the performance of the preparsing architecture shown in figure 2 with respect to the three pipeline stages: preparse, schedule, and render. The control dominated preparse stage limits the speed up for the preparsing architecture to 7x over single processor execution. Higher potential speedups can be achieved by using custom instructions for preparse subroutines to exploit instruction level parallelism.

The schedule stage using the DLS algorithm limits the speed up for the preparsing architecture to 6x over single processor execution. The scheduling heuristic can be modified to trade off the quality of scheduling results with the scheduling overhead, thus improving the potential speedup to beyond the 7x speedup of the preparse stage.

Figure 3 shows the potential speed up of the render stage for various number of processors. Run time MB level scheduling performs consistently 9-15% better than static compile time scheduling for configurations of more than four processors, and 60% or more faster than greedy dynamic scheduling for most cases.

The current run time MB level scheduling is based on a crude first order estimate of execution time. We present the

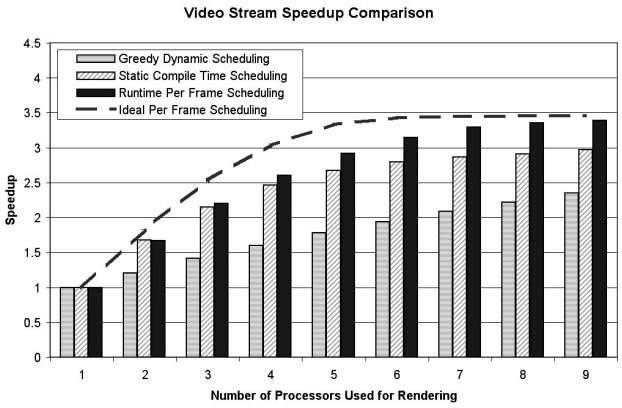

Fig. 3. H.264 decoding speedup comparison of the three scheduling approaches

resulting speedup if this estimate were perfect in figure 3 as the "ideal per frame scheduling", which is bounded by the input-dependent precedence constraints in each frame. We see some opportunity to refine our estimates to harvest more potential performance.

A typical P frame contains MB dependency chains that form execution critical paths dominating frame rendering time. As the frame size increases, the dominating MB dependency chains occur less frequently. For our low resolution test video stream, the numerous MB dependency chains reduced speedup improvements beyond five processors to 3.4x as indicated in figure 3. However, for a test movie segment at a higher 720p resolution, the speedup for the render stage extended beyond the bounds imposed by the preparse and schedule stages.

Given that the greedy dynamic scheduling approach is much more suited to handling P frames than I frames, one may consider a hybrid approach using static scheduling for I frames and greedy dynamic scheduling for P frames, eliminating the need for run time scheduling. Experiments show that such a hybrid approach performs significantly worse than the static scheduling approach and only marginally better than the greedy approach.

With run time MB level scheduling providing the boost in multiprocessor computation efficiency, figure 3 shows that the scheduling overhead can be easily amortized over multiple processors. Given an ideal estimate of task execution times, run time MB level scheduling can provide 2.5x speedup using only 3 rendering processors. In comparison, a 2.5x speedup requires 4 rendering processors using static compile time scheduling.

6. CONCLUSIONS AND FUTURE WORK

In this work, we showed that extracting fine grain dependences and input dependent task execution time at run time can enable efficient exploitation of available task level parallelism with run time scheduling. The speed up presented here could be implemented on top of frame level, instruction level and bit level parallelization exploited in [12] and [13]. We presented a MB level scheduling technique for H.264 which results in 60% speedup compared to greedy dynamic scheduling, and 9-15% faster schedules compared to static compile time scheduling on five or more processors.

The preparsing architecture has been modeled in SystemC. Work is under way to implement this architecture on an FPGA-based multiprocessor system and an ARM based multiprocessor system to verify the assumptions made in our work. We are also examining more efficient run time scheduling algorithms than the DLS variant used in this work in order to keep the scheduling stage manageably lightweight for larger size video frames. Extensions to the algorithm may also include MB clustering as proposed in [10], to enhance caching in platforms with hierarchical memory subsystems.

Although the speedups are sublinear with respect to the number of processor cores, it is important to keep in mind that single-threaded performance has leveled off, but the number of cores available for processing continues to increase. Consequently, we see the dynamic exploitation of data-dependent, fine grained parallelism as key to efficient utilization of future parallel platforms.

7. REFERENCES

[1] M. T. Bohr, "Intel's silicon r&d pipeline," 2006.

[2] K. Asanovic, K. Keutzer, D. A. Patterson, and et al, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[3] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 4th ed., 2007.

[4] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, 1993.

[5] T. Yang, *Scheduling and Code Generation for Parallel Architectures*. PhD thesis, New Brunswick, NJ 08904, May 93.

[6] N. Satish, K. Ravindran, and K. Keutzer, "A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors," in *DATE*, 2007.

[7] S. Ha and E. A. Lee, "Compile-time scheduling of dynamic constructs in dataflow program graphs," *IEEE Trans. Comput.*, vol. 46, no. 7, pp. 768–778, 1997.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[9] I. Y. Lee, I.-H. Park, D.-W. Lee, and K.-Y. Choi, "Implementation of the h.264/avc decoder using the nios ii processor," *Altera Design Contest Papers*, pp. 67–73, 2005.

[10] E. van der Tol, E. Jaspers, and R. Gelderblom, "Mapping of h.264 decoding on a multiprocessor architecture," *Image and Video Communications and Processing 2003*, pp. 707–718, May 2003.

[11] M. Fiedler, "Implementation of a basic h.264/avc decoder," *Seminar Paper*, June 1 2004.

[12] W. Li, E. Li, C. Dulong, Y.-K. Chen, T. Wang, and Y. Zhang, "Workload characterization of a parallel video mining application on a 16-way shared-memory multiprocessor system," *2006 IISWC*, pp. 7–16, October 2006.

[13] R. Ramanathan, "Extending the world's most popular processor architecture," *Intel Architecture White Paper*, 2007.