UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Parallel Processing of Multimedia Applications on Multi-core Architectures

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Dung Tien Vu

August 2014

Dissertation Committee:

Dr. Laxmi N. Bhuyan, Chairperson
Dr. Chinya V. Ravishankar
Dr. Rajiv Gupta
Dr. Walid A. Najjar

Acknowledgments

I want to thank Professor Chinya V. Ravishankar, Professor Rajiv Gupta and Professor Walid Najjar for being my committee members, and their comments contributing to my dissertation. Especially, distinguished Professor Laxmi N. Bhuyan, my adviser and the committee chairperson, whom I am proud of and so grateful for mentoring my research through the years and the completion of this dissertation. I also would like to thank Professor Vana Kalogeraki for her advising in the first years. During that period, I have published two papers.

I want to thank Computer Science Professors at California State University, San Bernardino, where I earned a MS degree, and support of my colleagues; Mr. Khalil Daneshvar, Ms. Hong Cullen in this university, where I work while studying at UCR. My grateful thanks to Professor Josephine Mendoza and Professor Arturo Concepcion who invited me to join their research projects at California State University, San Bernardino. I sincerely appreciate their unquestioned confidence about my research abilities as well as my unshakable determination in pursuing this Ph.D. endeavour. Professor Mendoza, my MS adviser and mentor of Calstate Doctoral Incentive Program, helps proof-read my papers and this dissertation.

I want to thank my brothers Cuong Vu, Quyet Vu, San Vu, Khue Vu, Van Vu, and Hien Vu, whom I am always proud of, not only their achievements, but also their caring for each others. Dr. Khue Vu, his wife, and his kids who have been with me since a night of April 1989, when we worryingly but bravely got abroad a tiny fishing boat, with just a self-taught navigation knowledge, to sail for the first time through the vast East Sea full of imminent dangers and inevitable misfortunes

I want to thank my wife, Thanh Hai, my staunchest supporter, who shares my worries and difficulties, and takes good care of our children, so that I can concentrate with my study.

I am grateful to my late beloved parents, also my teachers, who may know my academic achievement today. Their teaching of kindness, benevolence, generosity, self-esteem, striving, and examples of their own lives are the greatest things I have learned from them.

To my beloved parents, my brothers, my family, and my Professors.

ABSTRACT OF THE DISSERTATION

Efficient Parallel Processing of Multimedia Applications on Multi-core Architectures

by

Dung Tien Vu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2014
Dr. Laxmi N. Bhuyan, Chairperson

The well-known *wave-front* parallelization is proposed for parallel H.264/AVC video processing. Under this approach, groups of independent macro-blocks (MBs) are simultaneously processed, one group after another. Barrier mechanism is employed to synchronize processing of the independent MBs. This approach, however, has a substantial synchronization overhead that significantly affects the throughput performance. A novel *dynamic scheduling* scheme with recursive tail submit provides a good throughput performance by exploiting macro-block level parallelism and alleviating the synchronization overhead and thread contention. Nevertheless, it fails to achieve an optimal performance due to the use of a global queue, and an unawareness of cache locality of the underlying multi-core architecture. We propose a *distributed dynamic scheduling* scheme that employs distributed LIFO queues, and schedules tasks in a cache locality-aware and load-balancing fashion.

In H.264 video encoding, hierarchical search is widely proposed for the most expensive motion estimation. As a graphics accelerator, GPGPU is able to off-load compute intensive functions. GPGPU is, therefore, suitable, especially with full search-based approaches as the process can be efficiently parallelized. However, its fixed pyramid structure lacks a mechanism to select the best multiple-candidate scheme considering diverse video encoding characteristics. We propose profiled-based *fixed multiple-candidate motion vector selection* scheme, and an efficient *dynamic multiple-candidate motion vector selection* scheme to dynamically select the best multiple-candidate motion vector schemes at runtime.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

H.264/AVC [24], the current state-of-the-art video standard is designed with outstanding features that enhances not only video quality but also offers a high compression ratio. H.264 compression ratio is reported to be 2X higher compared with DivX format [18]. The standard has been widely adopted in a broad range of video services such as television broadcasting, video storage and transmission applications. However, processing of H.264/AVC video is very compute-intensive. Multi-core processing either using multiple conventional MIMD (*multiple instruction, multiple data*)-based CPU or integrated with SIMD (*single instruction, multiple data*)-based General Purpose Graphic Processing Units (GPGPU) are the choices due to high throughput and low latency. They are also low cost and widely available in the market. *Non-uniform memory access* (NUMA) multi-core architecture [36], where shared memory is distributed among the nodes of processors, is becoming popular for its performance and scalability. However, parallel video processing on this architecture is challenging due to communication with remote memories, where data may be stored. High resolution videos (FHD, quad HD) even impose a greater challenge to parallel processing because adding more cores would waste computing resources and degrade QoS. In this paper we attempt to tackle those challenges with a focus on efficient processing (encoding and decoding) through proper scheduling of tasks on the NUMA multi-core architecture, and GPGPU as the graphics accelerators.

In H.264/AVC video processing, there are two types of decompositions, namely, task-level and data-level decomposition. In task-level decomposition, processing functions are assigned to different cores in a pipeline fashion. This decomposition type requires significant communication exchanges between tasks. Its main drawbacks are load imbalance and poor scalability [10, 31]. In data-level decomposition, a video is typically partitioned into data units, each assigned to a different core. The parallelism can be handled at different level of granularity: group-of-pictures (GOP), frames, slices, and macro-blocks (MB) of 16x16 display pixels. In general, more parallelism is available at fine granularity. In this research, we focus on MB parallelism, the finest granularity level, to improve the processing throughput.

In MB parallelism, the popular *wave front* parallelization has been proposed to simultaneously process groups of independent MBs subject to data dependency [47, 48, 16, 26, 25]. Processing of a previous MB group will resolve the dependencies of the next MB group. However, while running this algorithm on our multi-core server, we notice that the synchronization overhead limits the performance gains. The *wave front* approach employs *barrier* mechanism to synchronize encoding among independent MBs. Since MBs have varying processing time, threads can not complete the processing at the same time, but must wait for each other at the *barrier* before together starting processing the next MB group. In addition, larger resolution gives rise to more iterations and more MBs per iteration. Those result in a significant synchronization overhead. In [25], authors report the throughput of their encoder employing *wave front* parallelization achieves a maximum speedup of 4X compared with a single-thread one regardless of the number of cores employed.

Instead of using the barrier-based synchronization, we follow a dynamic approach to execute the MBs as soon as their dependencies are resolved. The synchronization is required only once at the end of each frame, regardless of its resolution. A technique called *tail-submit dynamic scheduling* has been proposed in [20], where current threads will recursively process any ready MBs, and dispatch other ready MBs to a shared global LIFO queue. This technique improves locality, reduces interprocessor communication, and hence, improve performance. Nevertheless, the *tail-submit dynamic scheduling* still fails to achieve an optimal performance due to 1) the use of a global queue, which incurs substantial contention overhead when the number of cores increases and; 2) unawareness of cache-locality with respect to the underlying multi-core architecture that results in unnecessary latency, communication cost and load imbalance. In this dissertation, we propose a *distributed dynamic scheduling* scheme that de-centralizes the global shared queue into multiple distributed LIFO local queues deployed at processor nodes. We take advantages of hierarchical core / cache topology of the underlying multi-core architecture while placing these queues. Each queue is accessed by a small number of local threads. Each thread performs the distributed dynamic scheduling to dispatch MBs to the remote queues to preserve load-balancing and cache-locality.

In video encoding, the compression is done by compressing the residuals or the differences between the target blocks and the most resembling blocks in previously encoded area either in the same frame or on other frames. This processing is called motion estimation. The most efficient one is between frames: inter-encoding, but is also most compute expensive, which accounts for over 80% of processing time in video encoding [6]. Among motion estimation approaches, block matching algorithms are the most popular with diverse search approaches: full search (FS) that considers all the possible matches within a search window, is the most accurate, but most computationally complex. Heuristic search is faster but less accurate and, therefore, offers less compression. Hierarchical search (HS) presented in this dissertation, is widely considered for its low computational complexity and high efficiency. In this approach, the original frame resolution is down-sampled into multi-resolution images that stack as a pyramid with lowest resolution on top. A full or a heuristic search is applied at the lowest resolution for a quick but crude motion estimate (ME). This result then is expanded for the full resolution by propagating the initial motion vectors towards the original resolution at the pyramid's bottom level for the final ME. This hierarchical search is also called pyramid search.

As a graphics accelerator, GPGPU embedded with hundreds of Single-Instruction Multiple-Data (SIMD) cores, is able to off-load intensive computing of H.264 encoding and decoding from CPU. GPGPU is suitable for the motion estimation, especially with full search-based approaches as the process can be efficiently parallelized [6]. *Multi-candidate motion vector* approaches are proposed to mitigate the erroneous estimation from the reduced resolutions. However, their fixed pyramid structure lacks a mechanism to select the best multiple candidate scheme considering diverse video encoding characteristics. We propose an efficient *dynamic multiple candidate motion vector selection* approach to dynamically select best multiple candidate motion vector schemes at runtime. To summarize, We make the following contributions on parallel processing on multi-core CPU and GPGPU:

- Identify the synchronization overhead of *wave front* parallelization that significantly affects the performance of video processing, and reduce CPU utilization.

- Analyze the MB parallelism, and determine the maximum processing speedup for a video format. These speedups can be used as performance metrics to evaluate an encoder or a decoder.

- Evaluate the access contention problem of the global shared task queue, and also study the impact of cache locality of the underlying multi-core architecture.

- Propose *distributed-LIFO-task-queue* design for multi-core architecture to reduce the access-contention problem at the global task queue.

- Propose an efficient *distributed dynamic scheduling scheme* with cache-locality and load-balancing awareness to dynamically dispatch MBs to distributed task queues.

- Implement our proposed scheduling scheme on an encoder and a decoder using OpenMP and POSIX threads on a shared memory multi-core SGI server, and carry out extensive experiments with video benchmarks.

- Propose a flexible multiple-level hierarchical model using motion-estimation hirerarchical search on GPGU with variable fine-tuning search window sizes that can increase speedups by reducing the computational complexity.

- Analyze and verify the correctness of computational complexity of the pyramid search considering the number of candidates, fine-tuning search window sizes, and hierarchical levels. The complexity analysis helps correctly determine speedups of any schemes in advance.

- Propose an important motion estimation efficiency metric that evaluates how efficient a multiple candidate scheme is, compared with FS. A multiple candidate scheme that may provide a substantial speedup would not be worth using if its efficiency is insufficient.

- Propose profiling-based multiple candidate scheme selection, in which all possible multiple candidate schemes are profiled. A best fixed multiple candidate scheme for a video can be selected that will achieve a highest possible speedup and satisfy a desire efficiency.

- Propose an efficient runtime-based multiple candidate scheme selection that dynamically selects flexible multiple candidate schemes for the next frame at runtime based on speedup and efficiency of current scheme.

- Quantify our approach and demonstrate its performance via extensive experiments on a real workload using a Nvida Telsa GPU.

The rest of this dissertation is organized as follows: Chapter 2 is on the background of video processing and its parallelism. Chapter 3 is about related research of current parallel video processing approach such as *wave front* parallelization, *tail-submit dynamic scheduling*, and their drawbacks. Chapter 4 is on design, implementation, and evaluation of our proposed *distributed dynamic scheduling* on multi-core. Chapter 5 is about evaluation of our approach on parallel decoding on multi-core. Chapter 6 is on parallel motion estimation on GPPGU employing our proposed *fixed and dynamic multiple candidate motion-vector selection*, and finally, we conclude and outline the future research in Chapter 7.

Chapter 2

Background

2.1 H.264/AVC Encoding - Decoding

In video encoding, the compression is executed by compressing residuals between MBs of the target frames and that of other previously encoded MBs. During the encoding process, a prediction of a current MB is created using previously-coded MBs, either from the same current frame (intra prediction), or from other previously encoded frames (inter prediction). The *intra-prediction* predicts the current MB from surrounding pixels which are previously encoded. The *inter-prediction* makes the prediction from the most resembling blocks of previously encoded frames. A residual formed by subtracting the prediction from the current MB is further compressed with three processes: 1) Integer transform, a Discrete Cosine Transform (DCT); 2) Quantization by dividing the coefficients using an integer quantization parameter QP to

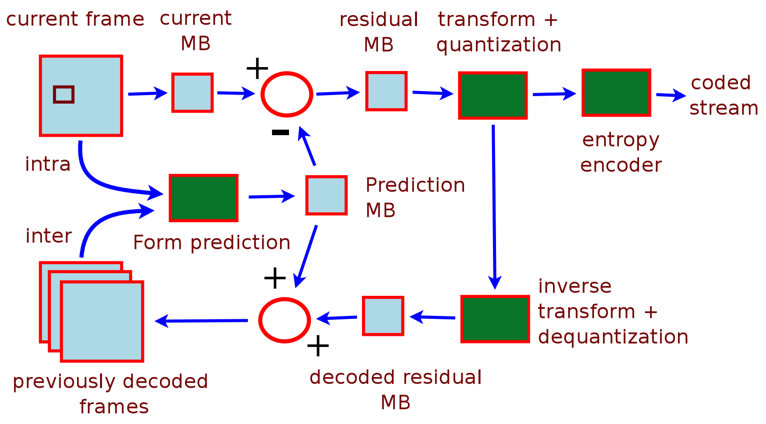


Figure 2.1: A Typical H.264 Encoding. A residual MB is formed by subtracting its original MB with a prediction created by intra or inter prediction.

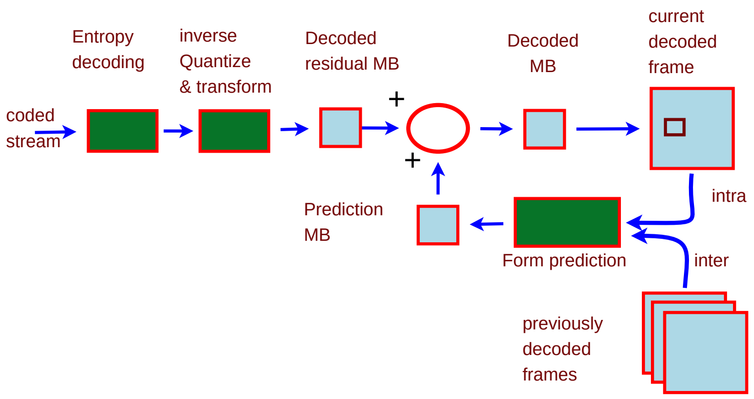


Figure 2.2: A Typical H.264 Decoding. Decoded MB is reconstructed by adding the decoded residual MB to the prediction MB.

reduce non-zero coefficients resulting in more compression; and 3) Context-Adaptive Variable Length Coding (CAVLC), or Context-Adaptive Binary Adaptive Coding (CABAC) to produce an efficient compact encoded bit stream, which can be stored or transmitted. Figure 2.1 illustrates the encoding process of a typical H.264 encoder. The decoding process is in reverse, in which the decoded residual of a macro-block is re-created by entropy-decoding, inverse quantization and inverse transform. The

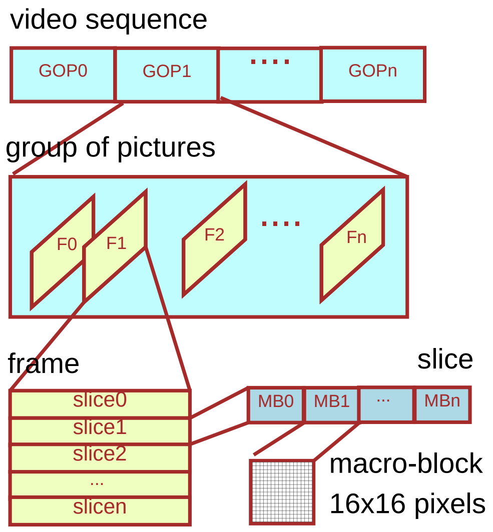


Figure 2.3: H.264/AVC Video Data Structure.

decoded macro-blocks and the whole frame is reconstructed by adding the decoded residual to the prediction MB formed either by intra or inter prediction. Figure 2.2 illustrates the decoding process of a typical H.264 decoder. With a hierarchical data structure as depicted in Figure 2.3, H.264/AVC has potential parallelisms at group-of-picture (GOP), frame, slice, and macro-block (MB) levels.

2.2 Parallelization Granularity

2.2.1 GOP-level Parallelism

GOP-level parallelism is coarsest-grained, in which the whole GOP consisting of many frames, is assigned to a processing unit (PU) or core. As GOPs are independent, they can be processed in parallel. The processing, however, requires a large number of frames being ready before the processing can start. The GOP-level parallelism needs a large memory to store frames, and incurs a long latency. It, therefore, well matches with cluster-based servers as proposed in [13], where machines have abundant computing resources, and video storing purposes as the latency requirements are not critical [25]. GOP-level parallelism is not well suited for multi-core architecture [10] and latency-constrained encoding scheme since it demands huge computing resources.

2.2.2 Frame-level Parallelism

When a video is encoded, frames are typically coded into three frame types; intra-coded (I), inter-coded (P), and bidirectional-coded (B) frames. Among the three types, B frame has the highest compression, followed by P and then I frame. H.264/AVC standard offers many options to choose reference pictures used in inter-prediction for particular usages. These options will affect compression efficiency, delay, storage, and parallelism.

1. *Base prediction* structure employs I.P.P.P GOP pattern, in which there are only I and P frames. P frame is reconstructed using one previous I or P frame only. This structure has low compression efficiency, no parallelism. However, it has minimum delay, and requires minimum memory storage to store reference pictures. This structure, therefore, is mainly used in video conferences.

2. *Classic prediction* structure employs I.B.B.P.B.B GOP pattern as shown in

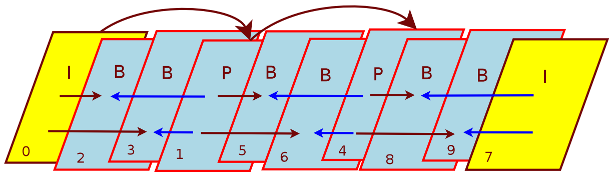


Figure 2.4: Classic Prediction Structure: B frames not used as reference frame are encoded using I(P) & P frames. The display order is from left to right, and the decoding order is in number .

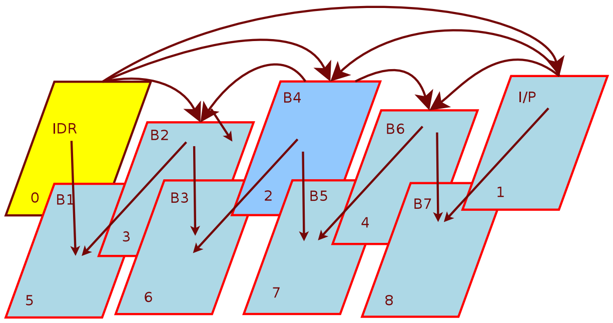


Figure 2.5: Hierarchical Prediction Structure. After Instantaneous Decoder Refresh (IDR) frame, B frames used as reference frames are decoded in order. Display order is from left to right, and decoding order is in number.

Figure 2.4. B frames are reconstructed using previously encoded I and P frames. This structure has better compression efficiency, but longer delay, and requires more memory storage. Its parallelism is limited as the number of B frames between reference frames I and P are small (maximum of 4).

3. *Hierarchical prediction* structure is where the B frame is also used a reference frame. This option offers the highest compression efficiency, but causes a longer delay and requires a large storage. Parallelism is limited since B frames must be processed in order.

Even though the frame-level parallelism is limited, its frame-level load balancing is scalable in multi-stream decoding where multiple streams are simultaneously decoded. Frame-based processing would be suitable with multi-core architecture since the frame-level processing requires synchronization among processing units due to inter-dependency among frames. Communication among processing units in multi-core is much less compared with cluster-based approaches. Frame-based processing requires less computing resources compared with GOP-level processing so it can be suitable for multi-stream processing where multiple streams are simultaneously processed. GOP-level processing, on the other hand, is suitable with cluster-based [13]. Since GOPs are independently processed (for close GOP), GOP-level processing does not need synchronization among machines, but it needs a large memory storage, which are more abundant in clustered-based machines.

2.2.3 Slice-level Parallelism

One of the simple ways to increase parallelism from the encoding side is to encode frames into multiple independent slices. A decoder can thus decode slices in parallel. and in any order as there are no dependencies across slices. This approach, however, has the following drawbacks:

1. It increases bit rate and bandwidth. In [10], authors report that with the same video quality, the bit rate increases up to 10%, 24%, and 34% for an increase

to 8, 32 and 64 slices, respectively.

2. It causes an imbalanced workload and synchronization overhead as slices may have different processing times

3. It has fixed parallelization opportunities as decoders have no control over the number of slices made at encoders. H.264/AVC standard allows a maximum of 16 slices in each frame that limits the slice-level parallelism. A combination of slice level and macro-block level parallelism is employed to increase parallelism and speedups.

2.2.4 Macro-block-level Parallelism

In macro-block parallelism, MBs as basic data units are scheduled to be simultaneously processed at multi-cores. MB-level parallelism is advantageous over GOP, frame and slice-level parallelisms as it is at the finest level of granularity that provides the highest parallelism. It is also scalable as the number of independent MBs increases along with frame resolutions .e.g. HD(1280x720) has 40 as the maximum number of independent MBs while FHD(1920x1080) has 60. Macro-block level parallelism can be exploited in two scopes:

1. Intra-frame MB-level parallelism, MBs are decoded inside a frame, frame after frame

2. Inter-frame MB-level parallelism, MBs across multiple frames are considered for simultaneous processing. MBs in proceeding frames can start processing before processing of preceding frames is completed.

The inter-frame approach is outside the scope of this dissertation. Interested readers can find it in [10, 47]. Our research focuses on macro-block level parallelism. With an efficient *adaptive dynamic scheduling* proposed in this research, thread synchronization overhead and queue access contention can be mitigated, good load-balancing and cache locality can be achieved.

2.3 Distributed Shared Memory Architecture

Distributed Shared Memory architecture, non-uniform memory access (NUMA), is a memory design for multiprocessors that provides separate memory node for each processor. Under NUMA the memory access time depends on the memory location relative to a processor. A processor can access its own local memory faster than non-local memory. Cache coherent cc-NUMA maintains consistent memory image when more than one cache stores the same memory location. As a result, cc-NUMA may perform poorly when multiple processors attempt to access the same memory address in rapid succession [36]. Figure 2.6 shows the hierarchical topology of our 8-node cc-NUMA 32 core SGI server. Each NUMA node has two dual-core Itanium 2 Montecito processors [39]. Figure 2.7, provided by *numactl –hardware* Linux utility,

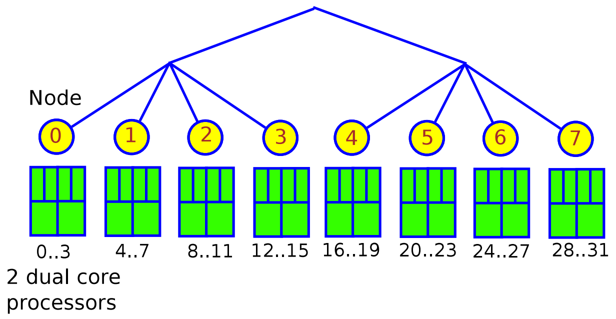


Figure 2.6: Fat-tree Hierarchical Topology of 32-core Atlix 4700 SGI Server. Each node has 4 cores, 7.5GB of memory accessible to all 32 cores.

Node	0	1	2	3	4	5	6	7
0	10	22	22	22	26	26	26	26
1	22	10	22	22	26	26	26	26
2	22	22	10	22	26	26	26	26
3	22	22	22	10	26	26	26	26
4	26	26	26	26	10	22	22	22
5	26	26	26	26	22	10	22	22
6	26	26	26	26	22	22	10	22
7	26	26	26	26	22	22	22	10

Figure 2.7: Distances Between Nodes of 32-core Atlix 4700 SGI Server.

shows distances among 8 nodes of the server that measures efficiency of accessing memory referring to increased latency and lower bandwidth of off-node memory.

2.4 Graphic Processing Unit (GPU)

Recently, GPU has been highly involved to further release the computation burden of CPU. General Purpose GPU (GPGPU) offers exciting capabilities for computation intensive applications with large degree of data parallelism. It is designed to be a single instruction multiple data (SIMD) processor, making it very efficient for some parts

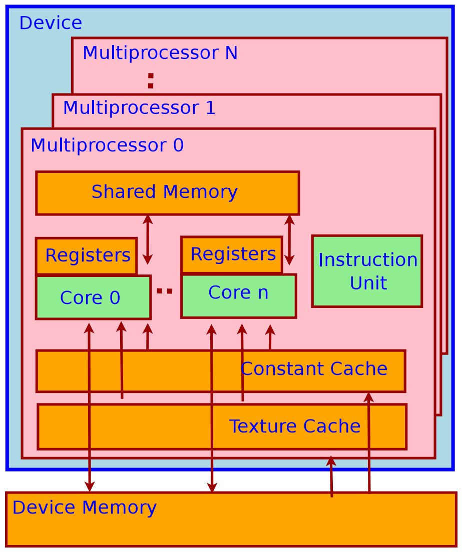


Figure 2.8: GPGPU Memory Hierarchy.

of the H.264 encoding process. Compared with traditional CPU design philosophy, GPGPU replaces the complex control logic with a large number of homogeneous cores to exploit the thread level parallelism. Thus the number of cores inside a GPGPU maybe in the hundreds or even thousands. Because of the architectural distinctions, GPGPU provides tremendous computation power compared with the CPU. The basic processing unit inside a GPGPU is a streaming processor. Depending on different architectures, a fixed number of streaming processors forms a group called streaming multiprocessor (SM) that runs the same instructions simultaneously. GPGPU supports several types of memory for designers to optimize their programs. As the fastest on-chip memory, registers are allocated per thread. The total number of registers for each SM is fixed. Shared memory is another efficient means for threads in the same block to share their data at runtime. Constant memory and texture cache are two read-only memory types, but the access time varies. Device memory is used for transferring data between a device and a host. The access efficiency is low compared with other memory types, but the size of global memory is relatively larger than other types. CUDA (*Compute Unified Device Architecture*) is developed by nVidia for high performance computation on GPGPU. It enables a CUDA-capable device to switch from graphical mode into high performance general purpose computation mode. After switching, the underlying hardware infrastructure remains the same, but the hardware usages are redefined for general purpose computation. Generally, CUDA-capable devices are connected with a host CPU, which is used for data transmission and kernel invocation for CUDA devices. CUDA explores a large amount of data parallelism inside an application and executes them simultaneously by invoking a huge number of threads at the same time.

Chapter 3

Related Research

3.1 Static Encoding Scheduling

3.1.1 Wave-front Parallelization

The data dependencies of an MB with its neighboring MBs as illustrated in Figure 3.1, include intra-dependencies within a frame: a) Motion vector prediction, intra-prediction; b) De-blocking filter dependencies; and inter-prediction dependencies are with other reference frames. The encoding of an MB can proceed only after all data dependencies of the MB are resolved by previously encoded MBs. When a frame is encoded, its MBs can be encoded in a sequential *raster scan* order that begins from the top-left MB, from left to right, row after row. During the encoding process, a varying number of independent MBs can be simultaneously encoded without violat-

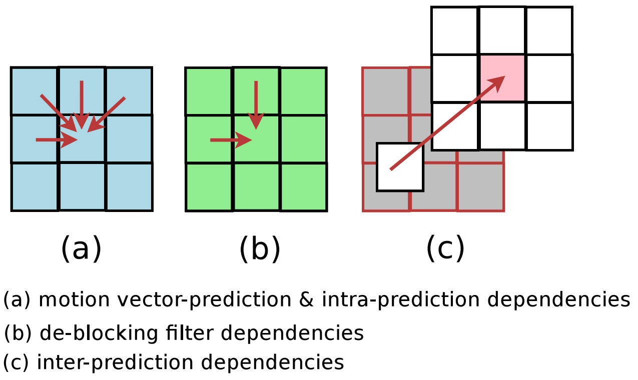

(a)
(b)
(c)
(a) motion vector-prediction & intra-prediction dependencies
(b) de-blocking filter dependencies
(c) inter-prediction dependencies

Figure 3.1: Intra & Inter MB Dependencies.

ing their dependencies. Figure 3.2 shows groups of independent MBs in diagonals, which are assigned the same time stamps TS. Parallel scheduling can be static or dynamic. One of the widely proposed static approaches is *wave-front* parallelization. Employing *wave-front* approach [47, 25], groups of the independent MBs are simultaneously encoded, one group after another, starting from the top-left corner, and moving towards the bottom-right corner in a wave-by-wave fashion. Barrier mechanism is employed to synchronize encoding of a group of the independent MBs. During the encoding, there is a varying number of independent MBs in diagonals that can be simultaneously encoded without violating their dependencies. Using *wave-front* parallelization [47, 25], encoding of a frame starts from the top-left MB, and spreads in a *wave-front* style, towards the bottom-right MB as the last encoded. Figure 3.2 illustrates groups of independent MBs in a diagonal assigned the same time stamp TS, e.g. TS5, TS6, …can be simultaneously encoded. The time stamp TS(x,y) assigned to each MB(row,column) can be computed using Equation (3.1).

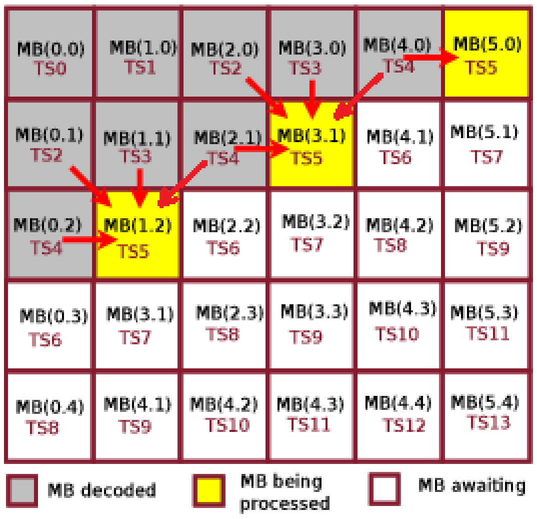


Figure 3.2: MB Parallelism. Group of 3 yellow MBs of the time stamp TS5, can be simultaneously encoded.

$$TS(x,y) = \begin{cases} x & \text{if } y = 0 \\ x + 2y & \text{if } y \geq 0 \end{cases} \tag{3.1}$$

Number of independent MBs $M(TS)$ that can be simultaneously processed at each time stamp, can be calculated using Equation (3.1.1), in which W_{frame}, H_{frame} are frame width and frame height in MBs, respectively.

$$M(TS) = \begin{cases} \lceil \frac{TS+1}{2} \rceil & \text{, if } TS < W_{frame} \\ \lceil \frac{TS+1}{2} \rceil - \lceil \frac{TS-W_{frame}+1}{2} \rceil & \text{, if } W_{frame} \leq TS < 2H_{frame} \\ \lceil \frac{TS+1}{2} \rceil - \lceil \frac{TS-W_{frame}+1}{2} \rceil & \text{, if } TS \geq 2H_{frame} \\ \quad -\lceil \frac{TS-2H_{frame}+1}{2} \rceil & \end{cases} \tag{3.2}$$

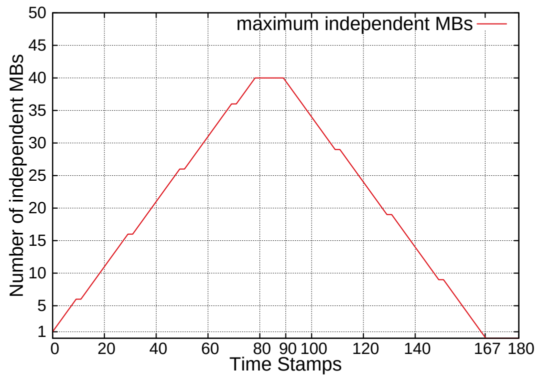


Figure 3.3: Concurrent MBs Over Time Stamp. HD(1280x720) has a maximum independent MBs of 40.

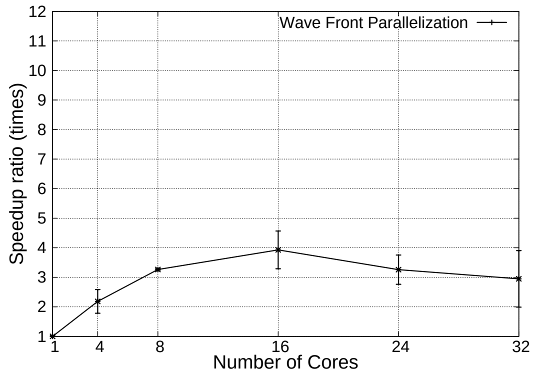


Figure 3.4: Wave Front

Figure 3.3 shows the number of independent MBs M over time stamps TS using Equation (3.1.1) for HD frame that reaches a maximum of 40. This maximum parallelism $maxM$ depends on frame size, and is calculated by equation (3.3).

$$maxM = \min\left(\lceil\frac{W_{frame}}{2}\rceil, H_{frame}\right) \tag{3.3}$$

For example, a HD(1280x720) frame is partitioned into 3600 (80x45) MBs. Using

Equation (3.1) the top left MB(x=0,y=0) is assigned time stamp $TS = 0$. The bottom right MB(79,44) is given a latest $TS = 167$. Assuming each MB had a fixed processing time, the encoding would sequentially take 3600 time units. However, it takes only 168 (167+1) time units when all independent MBs of the same time stamp are simultaneously encoded. These time units are also the maximum synchronization iterations $maxSync$ that occur and can be calculated using Equation (3.4)

$$maxSync = W_{frame} + 2(H_{frame} - 1) \tag{3.4}$$

The theoretical MB parallelism speedup S for the HD frame is 21.43 (3600/168) (assume there are enough cores to simultaneously encode all the independent MBs). The parallelism speedup S can be computed by Equation (3.5).

$$S = \frac{W_{frame} * H_{frame}}{W_{frame} + 2(H_{frame} - 1)} \tag{3.5}$$

Assuming encoding functions, which can be parallelized with MB-level parallelism, can account for 96.8% of the total encoding time [16], the maximum encoding speedups E can be achieved using Amdahl's law in Equation (3.6)

$$E = \frac{1}{(1 - P) + \frac{P}{S}} \tag{3.6}$$

In Equation (3.6), $P = 0.968$ is the total processing portion that can be parallelized, S is the speedup of the portion P. A maximum utilization U corresponding to the maximum speedup is computed by Equation (3.7)

$$U \approx \frac{2 * H_{frame}}{W_{frame} + 2(H_{frame} - 1)} \tag{3.7}$$

Table 3.1 shows the theoretical MB-parallelism speedups S, maximum encoding speedups E, and maximum CPU utilization for standard definition (SD), high definition (HD), full HD (FHD), Quad FHD (QFHD). These speedups E can be used as performance metrics to evaluate throughput performance of an encoder. With a corresponding parallelizable portions P for decoding, we can establish maximum decoding speedups D to evaluate the performance of a video decoder.

Table 3.1: Video Formats and Maximum Encoding Speedups

Video formats	Resolutions (pixels)	Macro-blocks	Max independent MBs	Theore. Parallel. Speed-up S	Max. Encode Speed-ups E	Max CPU Utilization U
SD	720x576	45x36	23	14.08	9.92	62.61
HD	1280x720	80x45	40	21.43	12.96	53.57
FHD	1920x1080	120x68	60	32.13	16.09	53.54
QFHD	3840x2160	240x135	120	63.78	21.20	53.15
UHD	7680x4320	480x320	240	127.31	25.25	57.25

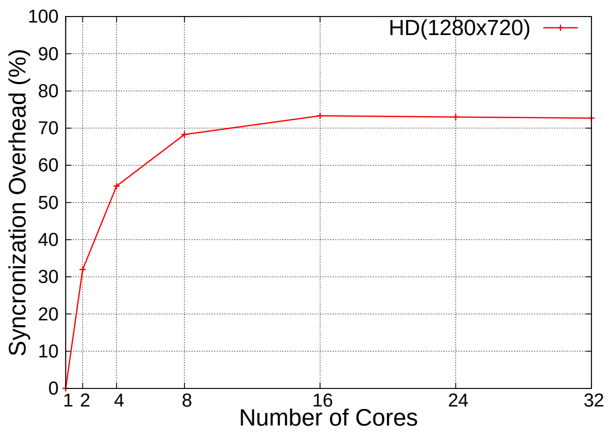


Figure 3.5: Barrier-based Synchronization Overhead. Percentages of maximum synchronization overhead against total processing time can be as worst as 70%.

3.1.2 Barrier-based Synchronization Overhead

Experimenting on the encoder [25] employing *wave-front* algorithm on a 32-core cc-NUMA SGI server, We identified a major drawback of a scalable synchronization overhead. The reason is that the *Wave-front* parallelization employs *barrier-based synchronization* mechanism to synchronize processing of groups of the independent MBs. The speedups would reach the maximum speedups shown in Table 3.1 if all the independent MBs, or at least between the barriers, had the same processing time. However, in reality, due to varying MB processing time, all encoding threads must wait for each other at the barrier before starting a next processing wave. This causes a substantial synchronization overhead that significantly affects the performance. Figure 3.5 shows percentages of maximum synchronization overhead over processing time of HD (1280x720) video that could reach 70%. Results in [25] and our experiments shows the throughput of *wave-front* static scheduling is saturated at 16 cores. In [16], authors report that barrier-based synchronization overhead accounts for 42% of the total encoding time. Larger video resolution has more synchronization iterations, e.g. 168 and 254 times using Equation (3.4) for each HD and FHD frame, respectively. In addition, load balancing among cores and the underlying cache/core topology of multi-core architecture are not considered.

3.1.3 Related Works

Macro-blocks parallelism has been extensively studied in the literature, and proposed for parallel video processing. In [31] Van der Tol et al. analyze advantages of processing decomposition at data-level over function-level. The function or task-level decomposition incurs a significant communication overhead, and load-imbalance among tasks while data decomposition inherently results poor data locality, and low scalability without code re-writing. As a basic processing block in video's hierarchical structure, MB-level parallelism is proposed for its highest parallelism.

Macro-blocks parallelism can be conducted by static or dynamic scheduling approaches. In static approaches, the popular *wave-front* parallelization is proposed to process independent MBs on diagonals in parallel. In [47, 48], Zhuo et al. propose *wave-front* parallelization to encode multiple video frames together as the independent MBs can be processed in parallel across multiple frames.

In [16], Seongmin et al propose an analytical model to estimate the H.264 encoding performance on embedded multi-core using OpenMP programming model. The results show data-level parallelism using *wave-front* parallelization is more efficient than task-level's by 41%. The authors also observe that load-balancing is a critical factor on encoding scalability.

In [26], Seitner et al. employ simulation to evaluate different parallel decoding approaches at MB level. Their static scheduling splits frames into groups of MBs and attempt to process them simultaneously.

In [25] A. Rodriguez et. al propose *p*264, an open platform to design a parallel encoder. *wave-front* parallelization achieves a maximum speedup of 4X. The authors report that combined slice and macro-block parallelism can increase the speedups up to 7X. Still based on *wave-front* parallelization approach, the speedup is limited due to the synchronization overhead. Furthermore, load balancing and the underlying cache/core topology are not considered.

3.2 Dynamic Decoding Scheduling

3.2.1 Global-Queue Dynamic Scheduling

In *dynamic scheduling*, the MB data dependencies are simply monitored and updated at each individual MB. As soon as the dependencies of an MB are resolved by the current processing, the ready MB is submitted to a global/single distribution LIFO task queue, which is accessed by all threads. This submission-distribution process illustrated in Figure 3.6, is dynamically handled at run-time. In this approach,

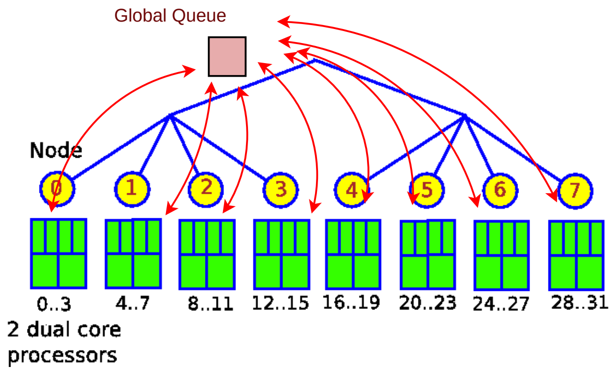


Figure 3.6: Global LIFO-queue Dynamic Scheduling on 8-node 32-core SGI Server. All cores dynamically submit and fetch MBs from a centralized global queue at run time.

the synchronization barrier is eliminated as all threads no longer wait for each other, but keep processing any available MBs in the global LIFO shared queue. Employing Last-in, First-out (LIFO) queue to select newest MBs instead of the oldest, is typically better for cache locality as the newest task has a better chance of getting cache hit from the last executed tasks. However, since only one MB at a time can enter or get fetched from the global queue, an access contention may occur and scale. Higher resolutions and larger number of cores worsen the contention problem. In the next sections, we will consider other approaches to mitigate this affect and improve the throughput.

3.2.2 Dynamic Scheduling using Recursive Tail Submit

One simple and efficient approach to mitigate the access contention bottleneck in the global shared task queue is reducing the number of MBs entering in the global queue. Instead of submitting all ready MBs to the global queue, current threads will themselves process these new MBs. This submission strategy resembles a *tail submit recursive* function [12], in which the processing recursively jumps back to its beginning to process the new MBs. The benefits are multi-fold:

1. Avoid further contention to the global queue as not all MBs have to be submitted to the queue.

2. Take advantage of cache locality as data needed for processing new MBs may be still available in caches of the current cores.

3. Reduce the submission-fetching overhead at the global queue.

When an MB is processed, at most two neighboring MBs become ready; a right MB and a left-down one, which are in the next time stamp as shown in Figure 3.2. There are three possibilities: 1) If only one MB is ready, it will be immediately processed using *tail submit*; 2) If both MBs are ready, there are two processing options: *i*) *Right-First order* processes the right MB and submits the left-down one to the global queue. *ii*) *Left-Down-First order* processes the left-down MB and submits the right one to the queue; 3) If none is ready, current thread will access its *home queue* to fetch any available MBs.

This scheme significantly reduces workload and enhances performance. Employing the *global-queue dynamic scheduling* approach, Azevedo et. al. in [10] propose dynamic 2D-Wave and 3D-Wave for parallelization of intra and inter-frame decoding, respectively, on cc-NUMA architecture. In their experiments with FHD video benchmark [2] on 64 dual core cc-NUMA SGI server [20], the authors report speedups of the 2D-Wave intra-frame decoding reaching a maximum speedup of 9.5X over the single thread.

The *tail-submit dynamic scheduling*, which is already much better than the *global-queue dynamic scheduling*, may not achieve a best performance because the global task queue still can cause the access contention, and the *tail-submit dynamic scheduling* still does not consider hierarchical cache/core topology of the underlying multi-core architecture. In the next section, We will further analyze the problem of the global distribution queue to explore appropriate solutions. A novel *tail submit dynamic scheduling* scheme [10] enables the current decoding threads to keep processing the newly ready MBs.

3.2.3 Global Distribution Task Queue - Two Drawbacks

Scalable Access Contention

In the global distribution task queue design, only one thread at a time can access the queue, either to submit or fetch MBs. The larger number of threads, the longer a thread has to wait to access the queue. The larger the number of MBs in the queue, the worse the contention is as each MB needs an exclusive access to enter and to get fetched. This problem resembles one of symmetric multiprocessing (SMP), where the access of multiple processors to the centralized main memory is serialized. To mea-

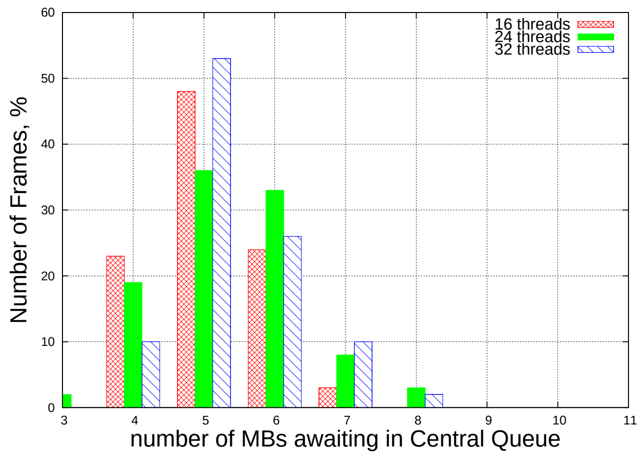


Figure 3.7: Tail Submit. Unprocessed MB in global queue does not decrease as decoder threads increase.

sure the access contention caused by the global queue, we run decoding experiments employing the dynamic scheduling on our cc-NUMA multi-core SGI server. Figure 3.7 shows frames percentages with corresponding highest number of MBs awaiting in the global task queue for 16, 24 and 32 employed cores. The figure shows that at certain time during the decoding process there exists significant percentages of frames, up to 80%, that have 5 or higher number of MBs still remaining unprocessed in the global queue. This number is supposed to decrease when the number of threads increases. Experiments in [20] shows the *global-queue dynamic scheduling* scheme on cc-NUMA multi-cores is no longer efficient when the number of threads exceeds 24. The observed opposite phenomena indicates that there exists some access contention problems that compromise the performance. This contention problem among threads can be mitigated when the global queue is divided into multiple distributed local queues, which are accessed by a smaller number of local threads. As the MBs submitted to the global queue are ready for processing, they can be divided and sent to the multiple distributed queues, where the MBs are submitted and fetched in parallel by groups of small number of local cores.

Unawareness of Cache Locality

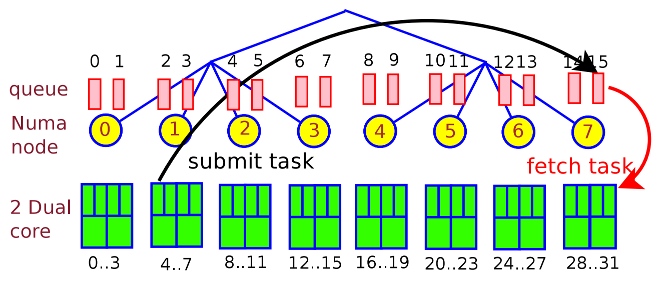


Figure 3.8: Undiscriminated Scheduling Scheme. Thread 4 submits MBs to distant queue 15.

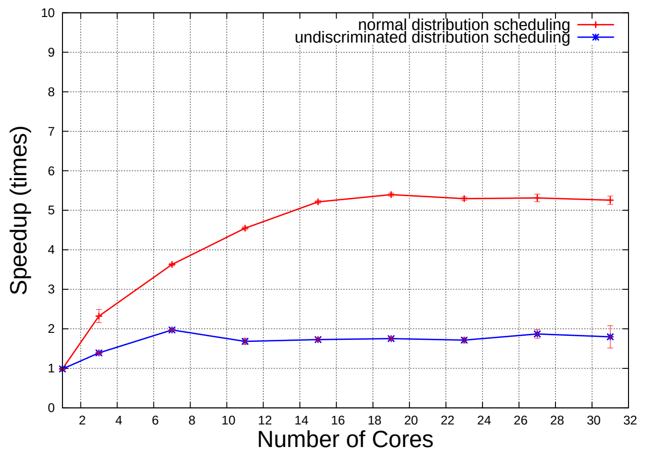


Figure 3.9: Normal Dynamic Scheme Outperforms Undiscriminated Dynamic Scheme.

As any thread can indiscriminately fetch the MBs from the queue, the *tail submit* may not be cache-locality aware. The MBs in the global queue are arbitrarily and indiscriminately fetched by any threads. They may be executed in different cores compared to where the previous MBs were executed. As a result, the cache locality is lost and time is incurred in transferring these data to the new cache.

To evaluate the impact of cache locality, We design a 2-thread-per-queue decoder, a multiple distributed task queue scheme, which is described in detail in Section 4. This scheme has 16 queues, each is accessed by two threads. Experiments are conducted with two schemes:

1. Normal dynamic scheme: all ready MBs are sent to a thread's home queue, unless other queues are empty. In this case MBs are sent to a closest empty neighbor queue;

2. Undiscriminated dynamic scheme: all ready MBs are sent to the farthest queue based on the distant map. Figure 3.8 illustrates an example, in which thread 4 submits MBs to queue 15, which is accessed by two distant cores 30 & 31.

Figure 3.9 shows the normal scheduling scheme outperforms the undiscriminated scheduling up to 2.5 times regardless of the number of cores employed. In video processing, as MBs also need data from their neighboring MBs, scheduling neighboring MBs to nearby cores will be significantly advantageous due to the cache locality. In the next section, We will propose the *distributed dynamic scheduling* that employs *multiple-distributed task queues* and schedules MBs with load-balancing and cache-locality awareness.

3.2.4 Related Work

In the dynamic approach, there are many schemes [3, 4, 10, 12, 20] proposed to dynamically schedule MBs at runtime. In [3], Amit Guy et al. propose a shared MB list of the ready MBs, and local-based scheduling, which attempts to improve data locality by letting each worker thread pop the MB from the MB list that is closest (in raster scan order) to the previous MB processed by the thread.

In [10, 4], Meenderinck et al. analyze parallelism of static 3D-wave and dynamic 3D-wave scheduling for inter-frame video decoder, in which MBs across consecutive frames are simultaneously decoded. The authors show that the static wave-front-based 3D-wave offers a significant parallelism when motion vector range is restricted to 16 pixels, which can not be guaranteed. Extensive simulation experiments employing dynamic 3D-wave, however, show a significant speedup.

Close to our work are from Maurico Mesa et al. [20], who propose dynamic scheduling employing a global-distribution-task queue and tail submit tactic on cc-NUMA architecture. Their work, however, does not address the access contention problem of the global task queue, and does not consider cc-NUMA with core/cache topology of the underlying hierarchical cc-NUMA architecture.

Other approaches employ work-stealing scheme to balance work load among cores such as Cilk [21] MIT's C language extensions for multi-core, and Thread Building Block (TBB) template library [15]. Hoogerbrugge et al. in [12] propose a dynamic scheduling with tail submit for Ultra HD on simulated architecture composed of 16 multi-threaded TriMedia cores, which is designed for media-processing. Each multi-threaded core employs a task queue, which is load-balanced by a distributed task stealing mechanism. When a core finds its queue empty, it will steal tasks from a randomly chosen queue.

3.3 Motion Estimation in Video Encoding

3.3.1 Hierarchical Pyramid Search

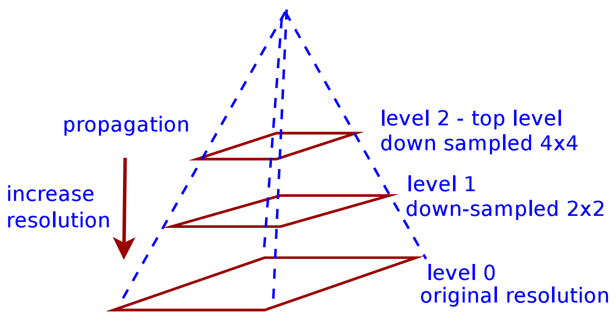


Figure 3.10: Three-level Pyramid. Level 0-original resolution. Level 2-top level has lowest resolution.

In video encoding, inter-prediction, the most efficient, creates the prediction from the most resembling blocks in previously encoded frames. The process of finding the best match is called *motion estimation*, which requires heavy computations accounting for 86% of the total encoding time [6]. *Block matching* algorithms are the most popular of *motion estimation* with diverse approaches: full search (FS), which considers all the possible matching within a search window to find a best matching block, is the most accurate, but also most computationally complex. Heuristic search is faster but less accurate and, therefore, offers less compression. Hierarchical search (HS) is widely considered for its low computational complexity and high efficiency. In this approach, the original frame resolution is down-sampled into multi-resolution images that stack as a pyramid with lowest resolution on top as illustrated in Figure 3.10. A full or a heuristic search is applied at the top image of lowest resolution for a quick but crude motion estimate (ME). This result then is expanded for the full resolution by propagating the initial motion vectors towards the original resolution at the pyramid's bottom level for the final ME. This hierarchical search is also called pyramid search. The multiple resolution images can be constructed by using a low pass filter. Gaussian lowpass filter reduces errors but with a cost of computational complexity. Mean intensity [22] can be used to build the images by a simple averaging Equation (3.8):

$$g_L(p,q) = \left\lceil \frac{1}{4} \sum_{u=0}^{1} \sum_{v=0}^{1} g_{L-1}(2p+u, 2q+v) \right\rceil, 1 \leq L \leq 2 \tag{3.8}$$

where $g_L(p,q)$ is grey level at pixel (p,q) of L_{th} level and $g_0(p,q)$ denotes pixel $g(p,q)$ of the original image. Sub-sampling is another popular resizing method in which a block of pixels is replaced by one pixel, normally the top left. This method is significantly faster but yields lower quality. Once the images are constructed, either a heuristic search or FS using block matching algorithm is conducted at the top level to obtain an initial or crude motion vector estimate. *Mean absolute difference (MAD)* or *Sum of absolute differences (SAD)* are popular matching criteria between blocks. The later is popularly proposed for H.264 video. The motion estimation cost includes cost of encoding the residual SAD value and cost for motion vector as in Equation (3.9)

$$Cost = SAD + \lambda * [C(MV_x - MV_{p_x}) + C(MV_y - MV_{p_y})] \tag{3.9}$$

Where $MV(MV_x, MV_y)$ is the median of three motion vectors of macro-blocks, which are on top, left and top right positions of the current macro-block. λ is an empirically obtained value and $C()$ is the cost of encoding a motion vector differential of that length. In this paper, We use motion estimation cost and motion estimate alternately. As the resolution is reduced, the FS is able to cover a large area and is no longer computationally expensive. To obtain the final motion estimate, the initial motion vectors must be propagated and refined towards the bottom level at each consecutive lower level. In the propagation process, the motion vectors of a higher level are multiplied by the corresponding scaling factor and refined by a search within a small area to find the best matching for this level. As the initial motion vectors are obtained at the reduced resolution, the final ME results propagated from these vectors may not be accurate since other motion vectors that even have a greater motion vector encoding cost may be better when they reach the bottom level. To solve this error, instead of propagating the single candidate motion vector, multi-candidate motion vectors are propagated and the best ones are selected. Besides low computation cost and high quality results, hierarchical search has a large number of parameters to specify: block size, number of levels, and scaling factors.

3.3.2 GPU-based Hierarchical Search

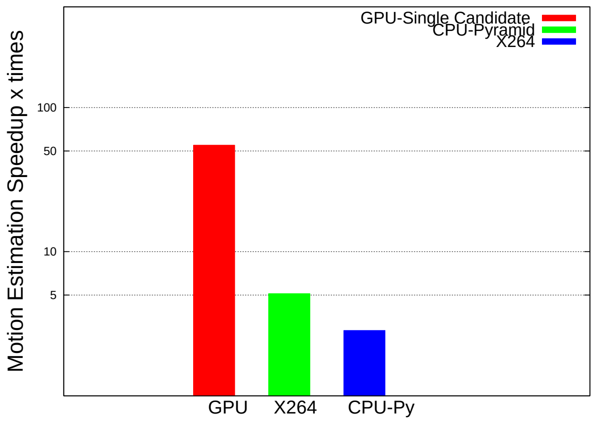


Figure 3.11: GPU-based Hierarchical search with Nvida's Tesla are 50x, 20x and 10x times faster than CPU-based full search, CPU-based pyramid search and X264 Hexagon fast search, respectively.

GPGPU embedded with hundreds of SIMD-based cores is most suitable for motion estimation, especially with the FS approach as the process can be efficiently parallelized. However, the common fundamental drawback of hierarchical search is the erroneous estimation at the top level of low resolution that may propagate to lower levels causing erroneous or inaccurate final ME. Some solutions have been proposed for CPU: in the initial motion estimation at the top level image, a couple of candidate motion vectors are selected for each macro-block. During the propagation process, all or a selected number of candidates are propagated towards the bottom level. At the end of the propagation, a lowest ME among the candidate motion vectors is selected as the final ME.

We run preliminary experiments for motion estimation to compare GPU-based searches with CPU-based searches. All GPU-based searches dramatically outperform CPU-based searches. Figure 3.11 shows speedups of GPU-based hierarchical searches (single-candidate) outperforms other CPU-based searches. The GPU-based hierarchical search, in our experimental platform are 50x, 20x and 10x times faster than CPU-based full search, CPU-based pyramid search and CPU-based Hexagon fast search, respectively. Although our server is multi-core, in our experiments, all CPU-based searches are single-threaded.

3.3.3 Major Drawbacks

The recent approaches, however, have failed to reach an optimal performance due to two major drawbacks:

1. They employ a fixed pyramid model and a fixed number of candidate motion vectors that may not achieve an optimum performance for all videos, even for

a video due to diverse video encoding characteristics.

2. They lack a mechanism to select the most efficient number of candidate motion vectors that can achieve a highest possible speedup, and provide a lowest possible ME.

In a recent paper [6], hierarchical search for single candidate motion vector has been proposed for GPU. To implement the search, a two-level pyramid is built where the original resolution is down-sampled 2x2 times to make the top-level image. One CUDA thread is assigned to compute ME for one position in the search window, and the entire thread block is dedicated to compute ME for one MB. This approach, however, is a single candidate vector approach that has the issue of inaccuracy as the initial motion estimate is obtained at reduced resolutions.

3.3.4 Related Work

Hierarchical search has been proposed in the literature [7, 17, 22, 28, 30] for CPU as a widely studied approach in motion estimation for its low, flexible computational complexity and high efficiency. However, most of these approaches do not fit GPU because either their block matching algorithms or optimization initiatives are heuristic where the number of iterations cannot be known and pre-determined. This kind of processing that is rich in control and branching, will not fit SIMD architecture, but causes threads to be serialized that significantly affects GPU's performance.

Authors in [22] proposed mean intensity as down-sampling method to build a *mean pyramid* of 3-level pyramid images. After the pyramid is created, a three-step search (TSS) [29] is applied at the top level image that has the lowest resolution using Mean Absolute Difference [44] as matching criterion to find the best matching block, and produce a coarse motion vector estimation. The motion vector is propagated down to level 1 of higher resolution using a corresponding scaling factor and refined with the heuristic search, and finally at level 0, where the final motion vector estimate is achieved. To mitigate the erroneous estimation at top-lowest resolution resulting in local minimums instead of global minimum, the authors proposed multi-candidate motion vectors and report that with two candidates, the *peak signal-to-noise ratio (PSNR)* is enhanced by half of the difference between TSS and full search (FS). With 9 candidates in TTS, one at the center of the coordinates, (0, 0), and eight positions surrounding the center co-ordinate, the PSNR is as good as FS's while the computational complexity in increased to half of FS's. Besides, the three-step search is heuristic, non-discrimination processing among 9 candidates would be disadvantageous for a large number of candidates.

In an effort to reduce the computational complexity, there are approaches to reduce the number of candidate vectors or bypass the refining process at the intermediate levels. Chan et al. in [7] propose an adaptive adjustment of the number of candidates at runtime based on MAD weights among candidates in Equation: $G_k = \frac{MAD_k - MAD_{min}}{MAD_k}$, where MAD_{min} is minimum MAD values among MAD_k of candidate k. If G_k are closed to 1 for all k, it means the search direction towards the current position with the minimum motion vector is probably correct. Thus, the search should concentrate on neighbors of this position. However, if one or more G_k values are close to zero, more candidates are needed as the accurate search direction is not clear. The authors report that simulations show their approach is better than TSS and multi-candidate approaches in both PSNR performance and computational complexity.

K. Lim et al. in [17] propose three candidate motion vectors to restrict excessive computational complexity. Two candidates are obtained from FS at top level while the third is obtained from motion vectors of neighboring blocks. The refining process of the motion vectors at lower levels that uses FS with reducing search range for lower levels and increasing pixel interval for higher levels, significantly reduces computational complexity. The authors report that the PNSR (Peak signal-to-Noise Ratio) performance is close to that of FS but computational complexity is only 1% compared with FS's.

Y. Shi et al. in [28] propose a threshold approach that bypasses the motion vector propagation and the refining process of remaining intermediate levels when a current level satisfies some threshold criterion. The approach is based on an observation that many motion estimates at lower level are relatively accurate to give a satisfactory motion compensation of the corresponding block at the bottom level of original resolution. The threshold criterion is obtained by squaring the *mean squared error* (MSE) from Equation $PNSR = 10log_{10}.\frac{255^2}{MSE}$ with a given PNSR. During the propagating process towards the bottom level, at any level, when an estimated motion vector for a block is obtained, a projected motion vector for the bottom level and accordingly MAD value of the block are computed. If the MAD value is below the threshold criterion, the propagation process for the block is terminated saving further propagation. The authors report that extensive experiments show their approach can reduce processing time up to 20% with the same compression quality as compared with other fast hierarchical searches. Along with [30] the authors also observe that two-level pyramid would give a better performance.

Close to our approach using GPU is [6], where motion estimation is obtained by GPU-based hierarchical search. Different from MAD as proposed in earlier papers, the authors propose cost of encoding the residuals and motion vectors as matching criterion. The cost approximately uses *Sum of Absolute Different (SAD)* considering motion vectors of neighboring blocks as described in Equation (3.9) A two-level pyramid is built with the original resolution is down-sampled 2x2 times to make the top-level image. One CUDA thread is assigned to compute motion estimate cost for one position in the search window and the entire thread block is dedicated to compute motion estimate for one MB. The results of the previous kernel is maintained in the GPU for the next call of the same kernel until the computation is complete and the final result of motion vectors are sent back to host CPU. Their approach, however, is a single candidate vector that fails to address the problem of inaccuracy as the initial motion estimate is obtained at reduced resolutions.

Chapter 4

The Design of Distributed Dynamic Scheduling

4.1 Parallel Processing Model

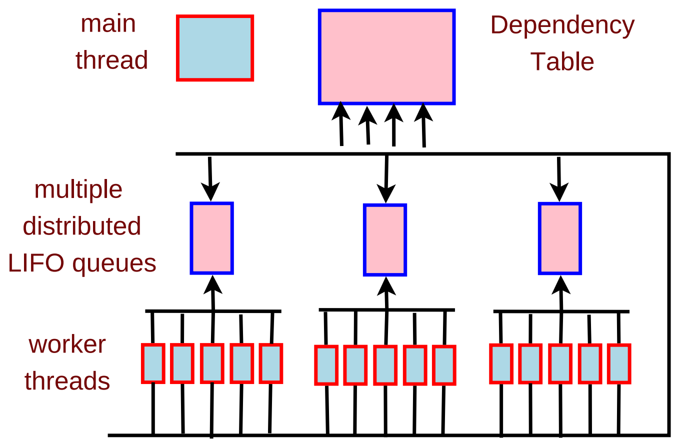


Figure 4.1: Parallel Processing Model on a Multi-core Machine. Worker threads update dependencies of MBs in parallel.

Figure 4.1 illustrates the processing model that consists of:

1. A master thread works as a control thread to create designated threads, and synchronize processing, frame by frame.

2. Distributed LIFO task queues deployed at cc-NUMA processor nodes to admit and distribute MB tasks, which are directly submitted by all processing threads, and distributed to local threads only.

3. Distributed dynamic scheduling employs multiple thread pools, one for each queue. Each pool consists of an equal number of local worker threads. Each worker thread is bound to a physical core. To minimize the memory access overhead, cores in the same thread pool will reside in the same processor node or the same hierarchical memory level.

4. Two-dimension *dependency table* is employed to simultaneously monitor and update dependencies of all MBs at run-time.

4.2 Multiple Distributed LIFO Task Queues

Our idea of distributed LIFO local task queues that de-centralize the global task queue into multiple local queues employed at processor nodes, resembles the advantageous design concept of *distributed shared memory* architecture, or non-uniform memory access (MUMA), where the centralized shared memory is distributed to mul-

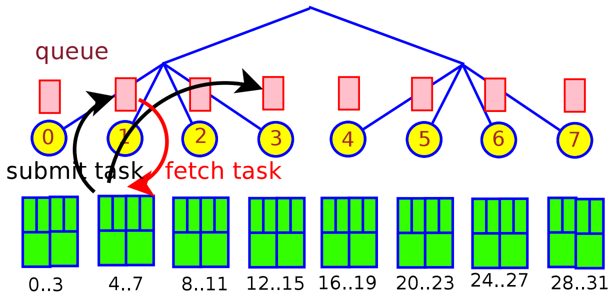


Figure 4.2: 8-node 32-core SGI Server with *4-thread queue* Topology. Threads 4-7 schedule MBs to any task queue, but fetch MBs from its home queue at node 1 only.

tiple processor nodes. This *distributed shared memory* architecture is known for better performance and scalability over shared memory architecture (SMA), or uniform memory access architecture (NUMA). *Distributed-queue topology* design is based on core/cache topology of underlying distributed memory architecture. We first appropriately choose a number queues depending on the number of processor cores, and assign threads to a queue called threads' *home queue*. Each thread is then bound to a designated core. In this dissertation, thread / core, and MB task / MB are used interchangeably. Next, we assign the smallest ID in each group to create the queue. A thread can schedule MBs to any task queue, but fetches MBs from its *home queue* only. Last, use a queue-queue distance map to reflect the average communication overhead between two queues. This distance map can be established through offline profiling or derived from distances between nodes shown Figure 2.7. Following are possible distributed-queue topologies that match our 32-core SGI server: *2-thread queue* topology: each queue resides on one processor and there are a total of sixteen queues in the system. This topology has a finest granularity with a minimal thread/access contention. However, the scheduling overhead on this topology is most expensive for up to 16 queues. Others are *4,8,16-thread queue* topologies. Since distances among queues in *2,8,16-thread queue* topologies do not match node topology, they are proportionally derived. The *global-queue dynamic scheduling* scheme is also considered having *32-thread queue* topology, where there is no scheduling overhead, but worst 32-thread access contention. Figure 4.2 shows the hierarchical topology of our 8-node 32-core SGI server, which is established with *4-thread queue* topology. There are a total of four task queues. Threads 4-7 schedule MBs to any queues, but fetch MBs from its *home queue* at node 1 only.

De-centralizing the global queue with distributed queues and having these queues, each accessed by a small number of local cores, achieves multi-fold advantages that significantly improve the throughput:

1. Mitigate the access contention problem with parallel access to multiple queues;

2. Fast data access with data locality and minimum memory access overhead.

Selecting the number of threads per queue that matches core/cache topology can narrow initial choices. In our SGI machine, selecting 2,4,8,16 threads per queue forms symmetrical topologies where our scheduling works best.

4.3 Distributed Dynamic Scheduling

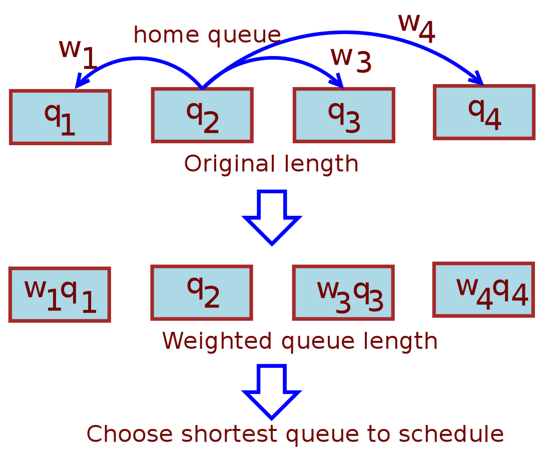


Figure 4.3: Minimum Weighted-Queue-Length-based Scheduling Algorithm.

In our scheduling design, we aimed at three objectives: 1) Minimize the MBs that must be scheduled; 2) Load balance to allocate each queue with the same amount of workload, and; 2) Cache locality awareness to schedule neighboring MBs to nearby queues so that cache-memory performance is optimal and the communication cost is minimal. As our target cc-NUMA platform has hierarchical core/cache topology, the inter-core communication cost is heterogeneous. Thus, a cache-memory locality aware scheduling can avoid unnecessary communication overhead and contribute to throughput improvement.

In this proposed scheme, scheduling is distributed to threads, each employs tail submit and dynamically schedules second ready-MBs to the distributed LIFO queues using *minimum weighted-queue length* scheduling algorithm. *Weighted-queue length* is defined as the original queue length multiplied by a *communication weight factor*, which is proportional to the distance between two queues. The scheduling algorithm is defined as follows: suppose the total number of queues is N, c is MB identifier and s is the queue identifier. w_i and q_i are communication weight factor and queue length of queue i, respectively. If an MB carries an identifier c, for instance, the mapping function $f(c)$ is then computed as in Equation 4.1. The result shows that this MB will be scheduled to queue s with corresponding weight factor w_s and queue length q_s.

$$f(c) = s \Leftrightarrow w_s * q_s = \min_{0 \le i \le N}(w_i * q_i) \tag{4.1}$$

Our scheduling algorithm works as illustrated in Figure 4.3. Suppose thre are four queues *1,2,3,4* with corresponding queue length q_1, q_2, q_3, q_4, and *communication weight factors* w_1, w_2, w_3, w_4. Each of these factors are normalized for a given hardware platform depending on the distance from the queue to the *home queue* so that $w_{homeQueue} = 1$. *Home queue* of a scheduling thread is the queue that the thread belongs to. In this example, the scheduling thread belongs to queue 2, therefore, *home queue* is queue 2 and $w_2 = 1$. Calculate the *weighted-queue length* for each queue and choose the queue with a least weighted length to schedule the MB. This scheduling algorithm achieves load-balancing because MBs are always scheduled to the least-loaded queue. In addition, as it employs weighted-queue length for workload comparison, which is aware of the underlying core/cache topology, the possibility of scheduling neighboring MBs to nearby queues is high.

In the next section, we implement our *distributed dynamic scheduling* on encoding employing OpenMP, and compare our scheme with existing schemes using extensive and diverse experiments on real H.264/AVC benchmarks.

Chapter 5

Parallel Encoding on Multi-core

5.1 The Implementation of Distributed Dynamic Scheduling

Each thread executes both encoding and scheduling tasks. In the encoding task, threads access their *home queues* when they are not empty, to fetch MBs, one MB at a time. In the scheduling task, threads access queues selected by the distributed dynamic algorithm to submit MBs, one MB at a time. This scenario resembles multiple producer-consumer problem. To implement the dual-task thread on OpenMP, a dual-lock mechanism as illustrated on Figure 5.1 is employed for each queue that consists of an *outer lock*, an *inner lock*, and *queue length* at run-time. The outer lock is available until its queue is empty as determined by its queue length, and available

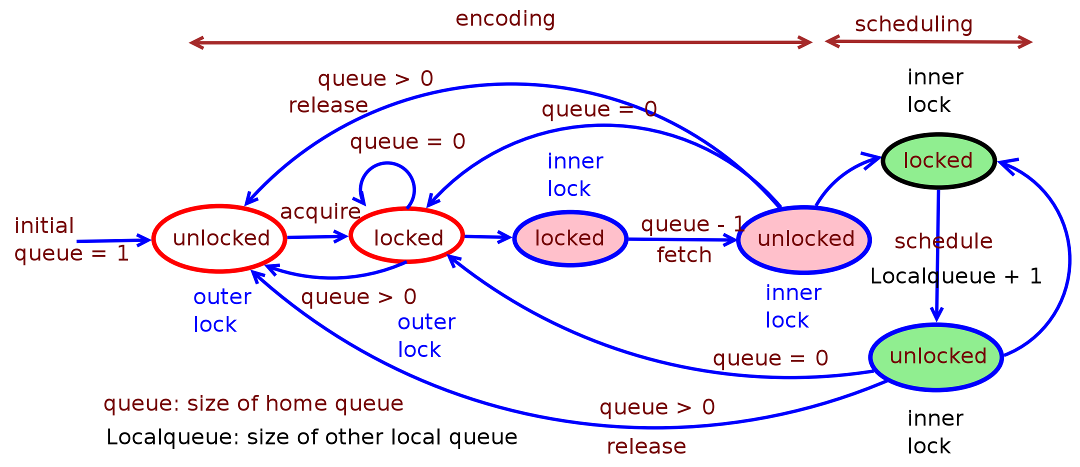


Figure 5.1: Dual-lock Mechanism for Encoding-Decoding Task: The outer lock is available until its queue is empty. The inner lock serializes access to its queue. During encoding, a thread accesses its home queue only, while during scheduling, it can also access other local queues. During the encoding initialization, queue 0 is assigned the top-left MB while all other queues are empty.

again when a new MB is added to the queue. To fetch an MB for encoding, a thread must acquire the *outer* and the *inter lock* while schedule an MB to a queue, the thread just needs to acquire the *inner lock*. The *outer lock* works partly like a semaphore while the *inner lock* works like a mutex, which allows only one thread at a time to access the queue to submit or fetch the MB tasks.

To monitor and update the data dependencies of all MBs, a two-dimension dependency table is employed. Each table element represents dependencies of an MB on others that are initially set to 2 as the left and the top-right MB are critical. If these two MBs have been encoded, all dependencies of the MB are resolved, and it is ready for encoding. The dependencies of MBs on the first row and the first column are set to 1 as MBs on the first row and the first column of a video frame do not have left

Algorithm 1 Encoding: each thread is bound to a core. Its access is mapped to a degsinated queue. *Repeat* command implements Tail submit by recursively performing encoding as long as one MB becomes ready after each encoding loop.

```
Require: $ThreadID, CoreID$
  $CoreID \leftarrow BindThreadtoCore(ThreadID)$
  $homeQueue \leftarrow MapCoreIDtohomeQueue(coreID)$
  while .NOT. End Of MB do
    $SetOuterLock(homeQueue); SetInnerLock(homeQueue)$
    $SetInnerLock(homeQueue); FetchaMB(homeQueue)$
    if .NOT.empty(homeQueue) then
      $UnsetOuterLock(homeQueue);$
    end if
    $UnsetInnerLock(homeQueue)$
    repeat
      $EncodeMacroblock(MB); UpdateDependency(MB)$
      if RightMBReady AND LeftDownMBReady then
        $MB \leftarrow RightMB; SubmitMBtoQueue(homeQueue, LeftDownMB)$
      else if RightMBReady then
        $MB \leftarrow RightMB$
      else if LeftDownMBReady then
        $MB \leftarrow LeftDownMB$
      end if
    until .NOT.RightMBReady AND .NOT.LeftDownMBReady
  end while
  return
```

and top-right MBs, respectively. As mutex of the dependency table is implemented at table element or MB, the finest-grained level, MB dependencies, therefore, can be independently and simultaneously updated and hence, MBs can be encoded in parallel resulting in a significant throughput increase.

Algorithm 1 shows the pseudo code of the encoding task. Each thread $ThreadID$ is bound to a physical core $CoreID$, and the thread's access is mapped to its home queue *homeQueue*. As long as its *homeQueue* is not empty, $ThreadID$ gains the

Algorithm 2 SubmitMBtoQueue(): employs distributed dynamic scheduling. To find a best candidate queue that has a *Minimum Weighted Queue Length*, the selection process loops through all queues begining at the nearest one, but closest empty queue takes priority.

```
Require: homeQueue, MB
  selectQueue ← homeQueue
  for all queue = nearest to farthest queue do
    queueLength ← GetQueueLength(queue)
    MinWeightedQueueLength ← 10000
    if queueLength == 0 then
      SetInnerLock(queue); insertMBtoQueue(queue)
      UnsetOuterLock(queue); UnsetInnerLock(queue);
      return
    end if
    WeightedQueueLength ← queueLength ∗ DistancetoHomeQueue
    if WeightedQueueLength .LT. MinWeightedQueueLength then
      MinWeightedQueueLength ← WeightedQueueLength
      selectQueue ← queue
    end if
  end for
  Set Inner Lock(queue); insert MB to queue(selectQueue)
  if GetQueueLength(selectQueue) == 1 then
    UnsetOuterLock(queue);
  end if
  UnsetInnerLock(queue)
  return
```

outer lock using *SetOuterLock*(), and the inner lock using *SetInnerLock*() to serialize fetching MBs from its *homeQueue* for execution. The tail submit recursively performs encoding *EncodeMacroblock*() with *repeat* statement as long as at least one neighboring MB becomes ready after every encoding. When MBs become ready, and assuming *Right-First order* is employed, the left-down MB is scheduled using *SubmitMBtoQueue*() and the right MB is immediately executed in the current thread. If no MBs are ready, the current thread will attempt to access its *homeQueue* to fetch any available MB. If an MB is the last encoded, the current thread will send End-of-MB tasks to all threads. Upon accessing the End-of-MB task, threads reach the barrier of the existing frame and start encoding a new frame. For simplicity, the end-of-MB handling is not mentioned in the pseudo code.

Algorithm 2 describes the pseudo code of distributed dynamic scheduling to schedule MBs to distributed queues that employ *minimum weighted queue length-based* algorithm with load balancing *cache-locality* awareness. During the scheduling, the selection process loops through all queues starting from the nearest one including the current *homeQueue* to find a best candidate queue *SelectQueue*, which has a minimum weighted-queue length $MinWeightedQueueLength$. The weighted-queue length $WeightedQueueLength$ of a queue is computed by multiplying the queue's *queueLength* with the distance *DistancetoHomeQueue* from the queue to *homeQueue* of the scheduling thread. Any empty queue (*queueLength* = *0*) first encountered will be selected, and the selection process is terminated. When a *queue* is empty, the queue's *outer lock* is yet to be released, the thread will release the *outer lock* after the new MB is submitted to the queue using *UnsetOuterLock()*. To submit a new MB into *selectQueue*, the current thread only needs to acquire the queue's inner lock using $SetInnerLock(queue)$ to prevent access from other threads. For the global-queue dynamic scheduling, the implementation is the same, except that there is a single queue only.

5.2 Performance Evaluation

5.2.1 Experimental Setup

We run experiments on 32-core SGI Atlix 4700 sever [27], configured as 8-node cc-NUMA architecture, as illustrated in Figure 4.2. Each node has two dual-core Itanium 2 Montecito processors (1.6GHz, 16KB L1, 256KB L2D, 12MB L3) , 7.5cm GB shared memory [39]. OS is Linux version 2.6.16.27 with gcc 4.6.2. We use HD (1280x720) VideoBench benchmark [2], which is a collection four H.264 video clips of diverse characteristics. Each video has 100 frames in YUV 4:2:0 format, 25 fps. To emulate the incoming streams, video frames are sequentially read from files. We implement our approach on p264 [25], a modified version of JM-reference software [1]. OpenMP [45] is employed to create and bind threads to cores. Encoding experiments are conducted employing 4,8,16,24 and 32 cores with three scheduling approaches:

1. *Wave-front* scheme, the base case.

2. *Tail-submit dynamic* scheme employs global-queue dynamic scheduling with tail-submit schemes in *Right-First* and *Left-Down-First* orders.

3. *Distributed dynamic* schemes employ distributed-queue topologies, tail-submit and load-balancing with core/cache awareness.

To employ a designated number of cores such as 8, 16 cores in the 32-core server, we run *nummactl* utility with *physcpubind* parameter. Results and errors, if applicable, are the average of the three runs.

5.2.2 Tail-Submit Dynamic Scheduling

Figure 5.2 shows encoding speedups E for *wave-front* parallelization, *tail-submit dynamic scheduling* in *Right-First* order, and *Left-Down First* order. Achieving the same performance as reported in [25], the *wave-front* reaches a 4X speedup, and is saturated at 16 cores. The reason is when the number of cores exceeds the optimum threshold, any slower performance of extra threads will drag down the performance. It is even worse when the barrier-based synchronization will force all other threads to wait. The dynamic scheduling, however, employs *tail submit* scheme, and balances load to other idle cores via the global shared task queue. It achieves a maximum speedup of 7X or a 1.75X faster than *wave-front's*. Monitoring the MB dependencies at the finest-grained level, the dynamic scheduling has any ready MBs processed with fastest pace, which results in having more MBs ready, and higher throughout accordingly. We also observe that there is no throughput difference between *Right-First* order and *Left-Down First* order as selecting any of the two orders would not affect much the expensive inter-prediction, which accounts for (86%-89%) of total encoding time [25].

5.2.3 Distributed Dynamic Scheduling

Figure 5.3 shows speedups of three distributed-queue topologies (4,8 and 16-thread queue topologies employing *tail submit* scheme, *Right-First* order) that outperforms the *wave-front*. The *8-thread queue* achieves a highest speedup of 7.9X, which is 200% (7.9/4) and 120% (7.9/6.4 at 16 cores) of *wave-front's* and the tail-submit scheme, respectively; and over 62% (7.95/12.92) of the maximum encoding speedup E of HD format shown in Table 3.1. The reason for the significant throughput improvement is that the *distributed scheduling* scheme has an efficient scheduling algorithm, and robust mutex locking mechanism, which is implemented with minimum contention. In particular, the dependency table is employed with a mutex locking mechanism at the macro-block level, so that MBs can have their dependencies monitored and updated simultaneously resulting in a significant throughput increase.

Considering the performance of the three schemes at 8 cores, we can observe the impact of cache-locality and the access contention when the *4-thread queue* has higher speedup than *8-thread queue* and *16-thread queue*. It is because employing 8 cores, the *4-thread queue* deploys 2 queues, which have better cache-locality and access contention as two MBs can be simultaneously accessed, and neighboring MBs are shared among local cores while the scheduling overhead for two queues is low. The same observation for 16 cores, *4-thread queue* and *16-thread queue* has the same speedup while the *8-thread queue* has the highest speedup. The best queue topology would be a balance between gain and overhead, in which the underlying core/cache topology will have a significant impact.

Figure 5.4 shows the best scheme of each approach. The *8-thread queue* topology achieves highest speedups as expected because it is the best topology of the distributed-queues. Performance of *2-thread queue* is not plotted as it is expected to have lower speedups compared with the others.

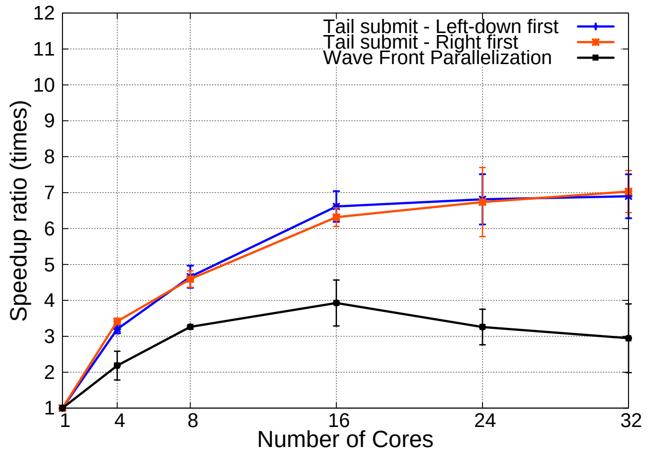


Figure 5.2: *Right-First* and *Left-Down-First* Orders Outperform *wave-front* Scheme.

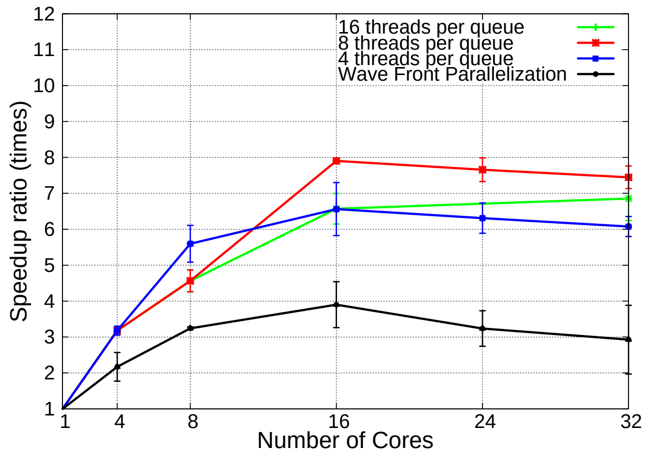


Figure 5.3: *8-thread queue* Topology Outperforms All Other Distributed-queue topologies.

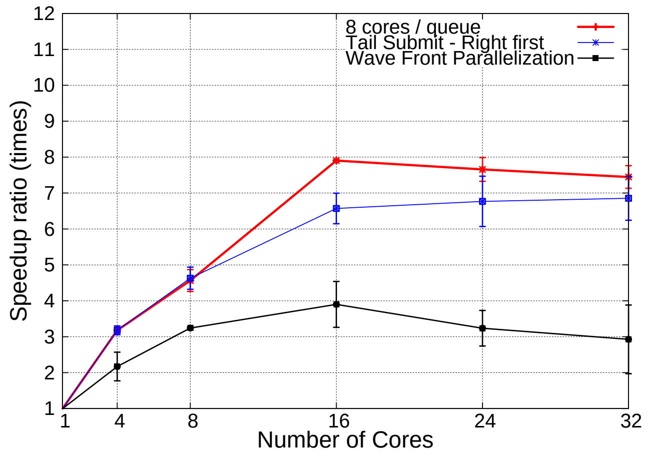


Figure 5.4: Best scheme: *8-thread queue* topology outperforms *tail-submit* schemes, other distributed-queue topologies and *wave-front* scheme.

5.2.4 Load-Balancing and Utilization Performance

Figures 5.5, 5.6, and the 5.7 show CPU utilization of *wave-front*, tail submit dynamic scheme and distributed dynamic scheme (using *8-thread queue topology*) when 32 cores are employed, the speedups have been saturated. We observe some interesting insights. With respect to load balancing, distributed dynamic scheme using distributed-queue topology is unexpectedly worst, *wave-front* and global queue using the global queue are better. This is because, in *wave-front* and the tail submit dynamic scheme, every core has an equal chance to get MBs. As a result, the distant cores, known to have high overhead, will grab MBs from the low overhead cores, whose load is not yet substantially imbalanced and overloaded, causing low throughput. Distributed dynamic scheme, on the other hand, only schedule MBs out of their home queues only when their load is badly imbalanced, otherwise, keep load on low overhead threads for the sake of high throughput.

In Figure 5.7, cores from 0-15 allocated in 1st and 2nd queues (by *8-thread queue*) are balanced and better utilized as the results of the distributed scheduling algorithm that schedules neighboring MBs to neighboring queues. Cores from 16-23 in the 3rd queue are almost idle, while cores 24-31 in the 4th queue are imbalanced as a result of tail submit when current thread keeps encoding ready MBs. This observation also explains and suggests employing 16 cores will have the best throughput for HD resolution. In terms of CPU utilization, *wave-front* approach has high utilization but low speedups. The reason is the *wave-front* parallelization employs barrier-synchronization, which is computationally expensive even in an efficient way [8]. In each HD frame alone, the synchronization across cores iterates 168 times. Our approach does not use barrier synchronization, but a light weighted scheduling algorithm.

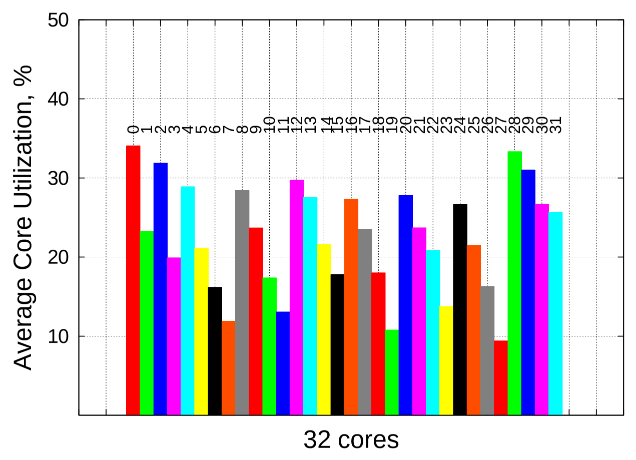


Figure 5.5: Wave-Front - Load is scattered all cores, inefficiently high CPU utilization.

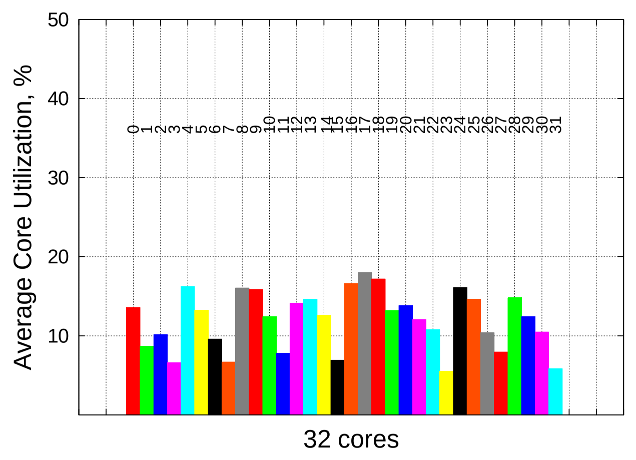


Figure 5.6: Tail Submit using Global Queue - Load is scattered to distant, high overhead cores.

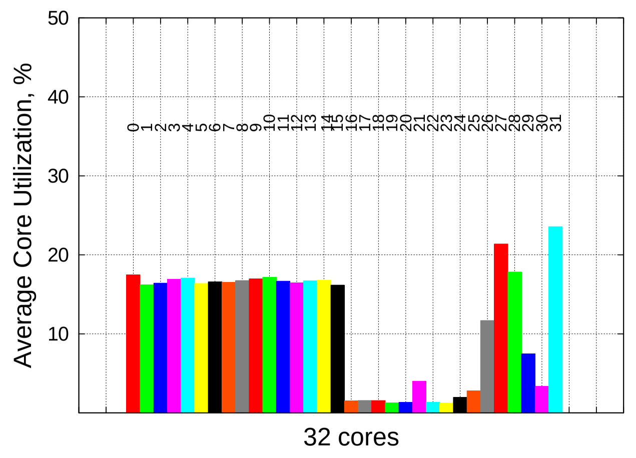


Figure 5.7: Distributed Dynamic Scheduling using Distributed Queues - Load is balanced within efficient cores.

5.3 Summary

In this chapter, we propose an efficient parallel encoding with *distributed dynamic scheduling*, which consists of *distributed LIFO task queues*, and load-balancing-and-cache-locality-aware scheduling that considers underlying multi-core architecture. Employing the distributed task queues, the access contention at the task queues is effectively mitigated; MBs are fetched and executed in parallel with fast data access thanks to data-locality and minimum memory access overhead. Employing the *minimum weighted-queue length* algorithm, the scheduling scheme preferably schedules neighboring MBs to nearby cores achieving both locality and load balance. We implement the *distributed scheduling* using OpenMP with dual-lock mechanism to facilitate both encoding and distributed scheduling tasks. We carry out extensive experiments on a 32-core shared memory SGI server with video benchmarks, and compare the *distributed scheduling* with wave-front parallelism and tail-submit dynamic scheduling. Our scheme outperforms *wave front* and the tail-submit by ratios of 200% and 120%, respectively; and achieves 65% to 70% of the maximum encoding speedup. This research is published at the ICME 2014, in Chendu, China [32].

Chapter 6

Parallel Decoding on Multi-core

6.1 Evaluation of Different Dynamic Schemes

6.1.1 Experimental Setup

We employ the same hardware and OS platform as in decoding experiments, (32-core SGI Atlix 4700 sever and Linux version 2.6.16.57 with gcc 4.1.0). We implement our approach ffmpeg-2Dwave [11], a modified version of a ffmpeg-H264, and run experiments with FHD-VideoBench benchmarks [2]. POSIX Threads [38] are used to create threads and bind threads to cores. POSIX Semaphores and Mutex are employed to implement submitting and fetching MB tasks from the distributed task queues. At run time one thread is reserved for control, leaving the rest for decoding task. We run experiments with four scheduling schemes:

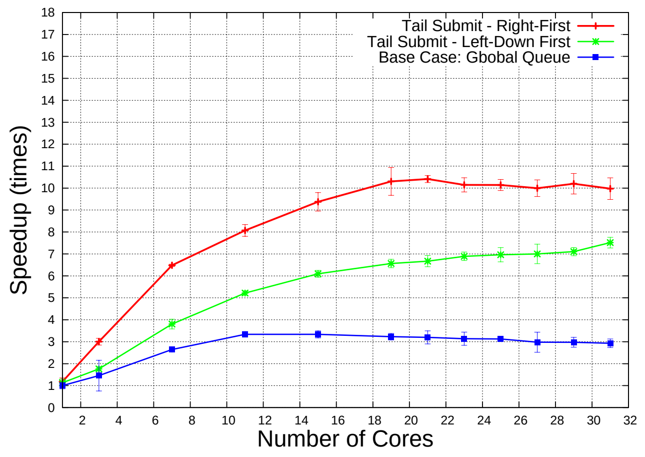


Figure 6.1: Tail-Submit. Righ-First Order outperforms Left-Down-First order and global-queue dynamic scheduling base case.

1. Global-queue dynamic scheduling, the base case: all MBs are sent to a global shared queue.

2. Tail-submit dynamic scheduling: global-queue, dynamic scheduling with tail-submit scheme in *Right-First* and *Left-Down-First* orders.

3. Simple distributed-queue dynamic scheduling: all MBs are scheduled to LIFO distributed queues.

4. Distributed dynamic scheduling: LIFO distributed-queue topologies, tail-submit scheme, and load-balancing with core/cache awareness.

Speedups of these approaches are compared against the single-core processing. Static *wave-front* parallelization is not presented because of its expected low speedups. Results and errors, if applicable, are averages of the five runs.

6.1.2 Tail-submit Dynamic Scheduling

In tail-submit dynamic scheduling scheme, ready MBs are submitted to the global queue by either the *Left-Down-First* order or the *Right-First* order. Selecting the right MBs to decode (*Right-First* order) achieves a substantial speedup over the (*Left-Down-First* order). Figure 6.1 shows speedup reaches 7.0X for *Left-Down-First* order, and 10.4X for *Right-First* order. The reason is the de-blocking filter, which removes artifacts between macro-blocks to improve the appearance of the re-constructed frames, uses left-MB and top-MB, whose data are more available in cache with *Right-First* order. In video decoding, the de-blocking filter accounts for 40% [35] of the decoding time, so that the *Right-First* order with better de-block filter performance, will achieve higher speedups than that of the *Left-Down-First* order. In encoding, however, both the *Right-First* and *Left-Down-First* orders achieve the same speedups as the de-blocking filter, which accounts less than 1% of the total encoding time [25].

6.1.3 Simple Distributed-Queue Scheduling

In this scheduling scheme, decoding threads submit all ready MBs to their home queues, unless other queues are empty. In this scenario, the closet empty queue, based on distance map, will get the priority. Figure 6.2 shows speedups for three distributed-queue topologies 2, 4, and 8-thread queues. The figure shows the speedups of all three topologies are higher than that of the global queue dynamic scheme. It indicates that the distributed-queue strategy is advantageous over the global-queue scheme by dividing the access contention of the global queue into distributed-queues. Unfortunately, this simple distributed-queue approach is still outperformed by the tail-submit scheme. The highest speedup of these kinds, as with 2-thread queue topology, is 5.4X, only about 50% of 10.4X, the highest speedup achieved by tail-submit scheme in our experiments. The reason is the tail submit scheme reduces access contention by effectively reducing the number of MBs submitted back to the global queue while in the simple distributed-queue approach, although access contention is divided into distributed-queues, 100% of MBs are still submitted back to the queues. Our proposed distributed dynamic scheduling is a combination of MB submission reduction and access contention division as presented in the next section will inherit all the advantages.

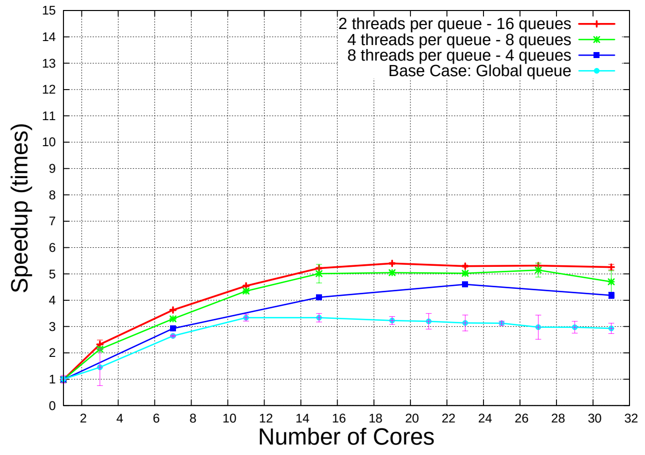


Figure 6.2: Simple Distributed Queues: All MBs are scheduled to multi-queues.

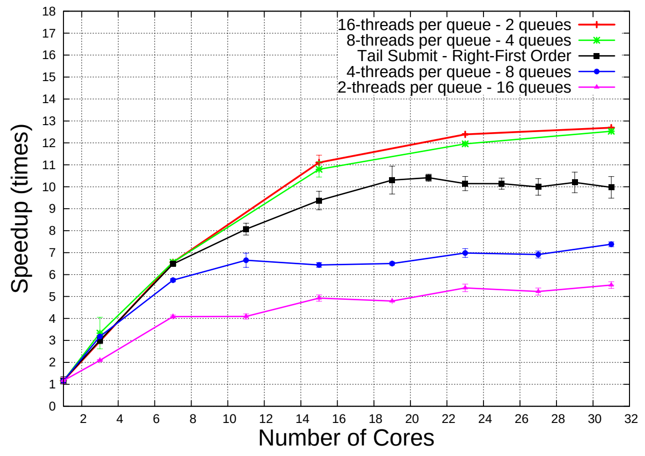


Figure 6.3: Distributed Dynamic Scheduling: distributed Queues, tail submit with load-balancing and cache-locality awareness.

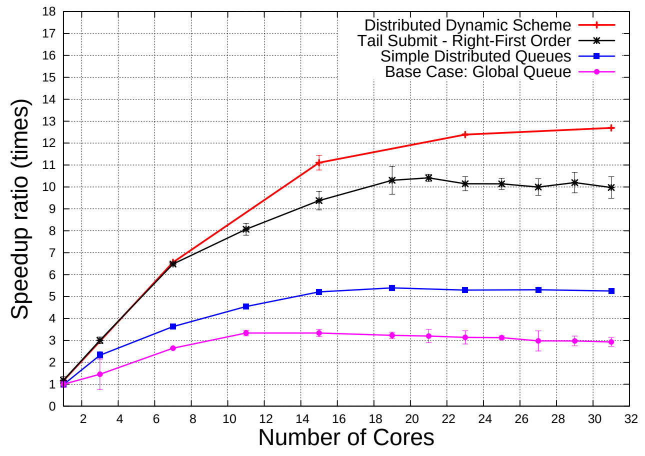


Figure 6.4: Best Scheme. Distributed dynamic scheme outperforms wave front and simple distributed-queue schemes.

6.1.4 Distributed Dynamic Scheduling

This distributed dynamic scheme combines distributed-queue topology, tail-submit scheme, *Right-First* order and schedule load balancing and cache locality awareness. Figure 6.3 shows speedups of different distributed-queue topologies compared with the base case. It shows *16-thread queue* topology achieves a highest speedup. We observe that in the simple distributed-queue approach (no tail submit), the smaller number of threads per queue, the better the performance will be. Figure 6.2 shows 2-thread queue, which results in 16, a largest number of queues, achieves a highest speedup (in simple distributed-queue category). However, in distributed-queue topology combined with tail-submit, the results show the opposite. *8-thread queue* and *16-thread queue* topologies achieve highest speedups. The reasons are:

1. In the simple distributed-queue approach, as 100% MBs are submitted, the thread access contention is a serious problem. Therefore, the more queues are employed, the better the performance is achieved;

2. In the distributed dynamic approach, the contention problem is mitigated, so that the best queue topology will be a balance between gains and overheads such as weighted-queue-length based selection process.

Figure 6.4 summarizes speedups of the best cases for each approach with FHD benchmark 1) the base case is 3.3X; 2) the simple distributed-queue scheme is 5.4X; 3) the tail-submit is 10.4X and; 4) the distributed dynamic approach is 12.6X, an increase of 20% compared with the tail-submit. The figure also shows speedup of the tail submit saturates at 22 cores while it does not with the distributed approach. It indicates that our distributed approach scales better than the tail submit. Our approach would have more improvement impact for higher resolutions (QHD or UHD) as processing demand is more intensive and the number of simultaneously decodable MBs is much higher. Figure 6.6 shows percentages of MBs passing through the queues. The MB percentages in our approach is slightly increasing higher than that of the tail submit. The reason is the distributed approach reduces the access contention so that threads can access faster the awaiting MBs in queues. As a result, there are more cases, in which both right MB and left-down MB are ready so that there is one, the left-down MB can be sent to the queues. As the number of cores increases, the theoretical MB-parallelism speedup S increases to the maximum shown in Table 3.1, while the core utilization decreases as the number of timestamps when all cores are fully utilized, decreases. The single-core decoder has a highest utilization of 100.

Figure 6.5 shows frame percentages that have MBs awaiting in queues of our approach is significantly reduced. At 32 threads, less than 5% of frames that has 5 as the highest number of the waiting MBs while in the global-queue tail submit shown in Figure 3.7, it is over 80%. This indicates our distributed dynmaic scheduling solution significantly reduces the access contention.

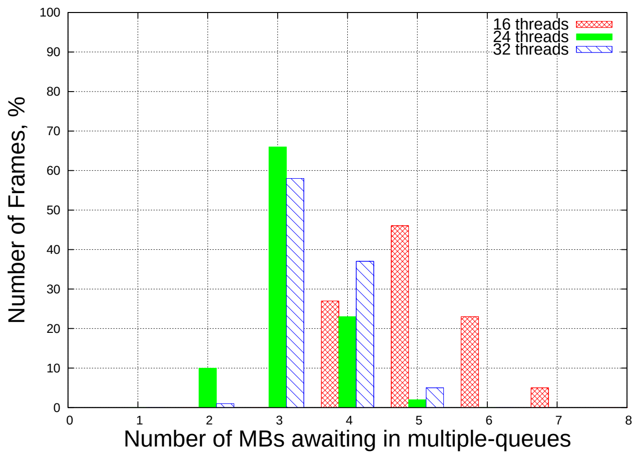


Figure 6.5: The Distributed Dynamic Scheduling: frame percentages vs. max. MBs in queue.

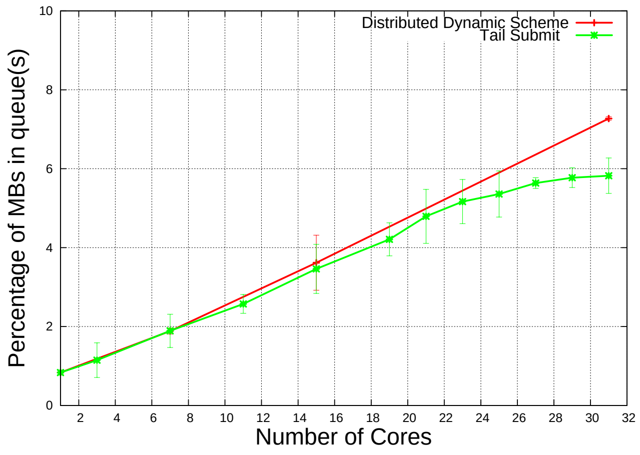


Figure 6.6: MB Percentages Sent to Queue(s) by Tail Submit vs. Distributed Dynamic Scheduling.

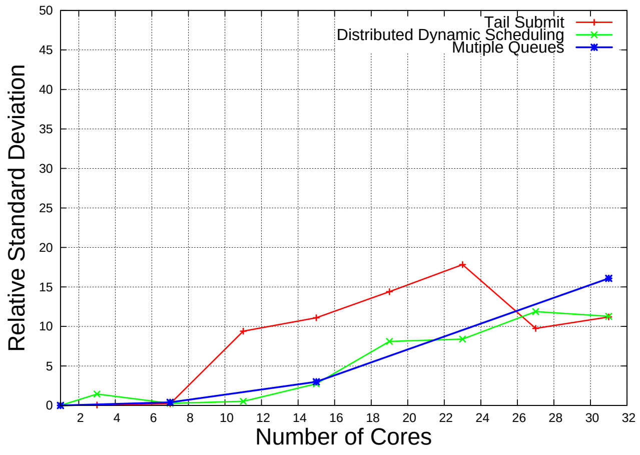


Figure 6.7: Relative Standard Deviation (RSD) of Thread Decoding Time.

6.1.5 Load-Balancing Performance

Figure 6.7 shows relative standard deviation (RSD) of decoding time among threads. As the number of threads increases, the RSDs increase. The global-queue dynamic scheduling, on one hand, has better RSD as each thread has equal priority in accessing the central queue. On the other hand, tail submit favors the actively decoding threads, which have needed data in their caches, to maximize the throughput. As a result, tail submit has the highest RSDs. Our distributed dynamic scheduling approach also employs tail submit, but its load-balancing and cache-locality aware scheduling efficiently alleviates the imbalance problem of the tail submit scheme. RSD of the distributed approach is slightly higher than that of the base case but does not exceed 20% at 32 threads. One interesting observation for tail submit scheme is that its RSD decreases as the number of threads exceeds 22. This drop can be explained as follows. As the number of threads goes beyond 22, tail submit causes performance degradation due to high thread contention. Thus, the RSD reduction results from the less workload on each thread, as opposed to more balanced workload given the same amount of workload compared to our scheme.

6.2 Summary

In this chapter, we propose the distributed dynamic scheduling scheme in video decoding on multi-core architecture. To evaluate our distributed scheduling scheme, We implement our approach on a 32 core shared-memory SGI server employing POSIX threads multi-core programming. We carry out extensive experiments using real video benchmarks comparing with diverse schemes such as global dynamic scheduling, tail-submit dynamic scheduling, and simple distributed-queue topologies. We observe that our distributed scheme outperforms tail-submit, one of the best scheduling algorithm, 20%, and achieves lower latency with more balanced workload and less communication cost. This research has been published in ICME 2012, in Melbourne, Australia [33]

Chapter 7

Parallel Motion Estimation on GPGPU

7.1 The Design of Dynamic Multi-candidate Motion-Vector Selection Scheme

7.1.1 Flexible Multiple-level Hierarchical Model

We build the multiple-resolution mean pyramid with 2x2 down-sampling factor. The macro-block, starting with 16x16 pixels, is down-sized 2x2 times at each hierarchical level. As a result, the resolution or total number of pixels are reduced, but the total number of MBs of images at any level is not changed. Our strategy is to design a *flexible pyramid model* that can increase the speedup by reducing the computational complexity, but keeping the motion estimation cost low. Search window is the important parameter that significantly affects the complexity. As the motion vectors are refined with FS as they are propagated towards the lower levels, the size of the fine-tuning search window, in general, can be reduced without significantly affecting the search quality as motion estimates are converged towards lower levels [28]. In particular, to cover the same coverage as in the corresponding FS, the search window at top level can be reduced proportionally to its down-sampling factor. At the bottom level, the search window does not need to be as large as the size of that in the FS. In fact, if the fine-tuning search window at lower levels is not reduced, the hierarchical search would not bring any benefit since it will be more computationally expensive than the FS. We introduce *variable reducing factor* f_r that adjusts the fine-tuning search window size at runtime to increase speedup where applicable. We will analyze and evaluate the effects of search window size in the complexity analysis and in the performance evaluation section. In general, the more hierarchical levels a pyramid model has, the more computational complexity is reduced, but with a trade-off of motion estimation efficiency due to more levels of down-sampling. Based on the computational complexity analysis presented in the next section, We select two-level pyramid model with variable fine-tuning search windows.

7.1.2 Computational Complexity Analysis of GPGPU-based Hierarchical Search

In FS, to find a best matching block in a search window, the SAD-based encoding cost obtained from Equation 3.9, must be exhaustively checked for all possible positions in the search window. The computational complexity to obtain the motion estimate (ME) for each macro-block (MB) in the original FS $\tau_{FS} = O(Sw.Sh.Bs)$, where Sw and Sh are width and height of search window in pixels, respectively, and Bs is size of each MB, which is 16x16 pixels. The complexity for the entire frame is $T_{FS} = O(Bw.Bh.Sw.Sh.Bs)$ [19], where Bw and Bh are the width and height of a frame in MBs, respectively . In our proposed pyramid search, the complexity to obtain ME for each MB at level l is $\tau_l = O(Sw_l.Sh_l.Bs_l.G_l)$, where Sw_l, Sh_l are width, height of search window in MBs, Bs_l is size of MB in pixels. G_l is the complexity factor of the lowpass filter. For mean intensity filter in Equation (3.8), $G_l = 4$. At the bottom level, the complexity $\tau_0 = O(Sw_0.Sh_0.Bs_0)$, where G_0 does not exist.

For a two-level pyramid, at top level ($l = 1$), size of the search window and MB are down-sized 2x2 times, the complexity for each MB is accordingly reduced, $\tau_1 = O(\frac{1}{2}Sw.\frac{1}{2}Sh.\frac{1}{4}.Bs.\mathrm{G}_l)$. As G_l =4, the complexity is 25% τ_{FS} At this level, the search window is reduced 2x2 times, but covering the same area and the same number of MBs as the MB block size is accordingly reduced. At the bottom level (level 0), to reduce the complexity We can choose a smaller fine-tuning search window using reducing factor f_r, such as 1/2, 3/8 or 1/4 of the original size. For $f_r = 1/2$, the complexity to fine-tune ME using FS for each candidate motion vector at bottom level, $(l = 0)$ $\tau_0 = O(\frac{Sw}{2}.\frac{Sh}{2}.Bs)$. We notice that τ_1 and each τ_0 account the same 25% τ_{FS}. For a single-candidate motion vector scheme, the total complexity for each MB is $\tau_0 + \tau_1$, just 50% τ_{FS}. Adding each extra candidate motion vector will increase the complexity to another τ_0, or 25% τ_{FS}. Two-candidate approach will be 75% τ_{FS} while three-candidate approach would not be any good since it is already 100% τ_{FS}. For $f_r = 3/8$ and 1/4, the complexity to fine-tune ME for each candidate motion vector at bottom level $\tau_0 = O(\frac{3}{8}Sw.\frac{3}{8}Sh.Bs)$ and $O(\frac{1}{4}Sw.\frac{1}{4}Sh.Bs)$ accounting just 14.063% $(9/64)\tau_{FS}$ and 6.25% $(1/16)\tau_{FS}$, respectively. This allows increasing number of candidates without exceeding FS complexity. Table 7.1 shows speedups of multi-candidate schemes (single can., 2 can....) and their percentages of complexity with respect to the base case FS. In this table, approach with $f_r = 1/2$ has only two practical multi-candidate schemes; single and two-candidate that offer 200% and 133% speedup (meaning compared with the corresponding FS), respectively, while approaches with $f_r = 3/8$ and 1/4 factor can have up to four and eight multi-candidate schemes, respectively, whose speedups are over 100%.

The total computational complexity of a multi-candidate motion vector scheme would also include the complexity of binary reduction needed to find the minimum motion estimation cost of the multi-candidate vectors $\tau_{Bn} = O(log\ n)$, where n is number of candidate motion vectors. e.g $log\ 2, log\ 3, ..$ that is negligent compared

Table 7.1: Percentages of Complexity and Speedups of Multi-candidate Motion Vector Schemes with respect to FS

Schemes	one cand.	2 cand.	3 cand.	4 cand.	5 cand.	6 cand.	7 cand.	8 cand.
$f_r = 1/2$	50%	75%	100%	N/A	N/A	N/A	N/A	N/A
Speedups	A1:200%	A2:133%	A3:100%	N/A	N/A	N/A	N/A	N/A
$f_r = 3/8$	39%	53%	67%	81%	N/A	N/A	N/A	N/A
Speedups	B1:256%	B2:188%	B3:149%	B4:123%	N/A	N/A	N/A	N/A
$f_r = 1/4$	31%	38%	44%	50%	56%	63%	69%	75%
Speedups	C1:320%	C2:267%	C3:229%	C4:200%	C5:178%	C6:160%	C7:145%	C8:133%

with total complexity of candidate motion vector τ_0, τ_1, τ_2.

For three-level pyramid, with $f_r = 1/2$, the complexity of top level (level 2), and each candidate motion vector at level 1 and level 0 are $\tau_2 = O(\frac{1}{4}Sw.\frac{1}{4}Sh.\frac{1}{16}.Bs.\mathrm{G}_l)$ or $(1.5625\%\tau_{FS})$, $\tau_1 = O(\frac{1}{2}Sw.\frac{1}{2}Sh.\frac{1}{4}.Bs.\mathrm{G}_l)$ or $(25\$\tau_{FS}), \tau_0 = O(\frac{Sw}{2}.\frac{sSh}{2}.Bs)$ or $(25\%\tau_{FS})$. respectively. The total complexity for single, two and three candidate scheme will be (τ_2, τ_1, τ_0) or $(51.5625\%\tau_{FS})$, $(\tau_2, 2\tau_1, 2\tau_0)$ or $(101.5625\%\tau_{FS})$, and $(\tau_2, 3\tau_1, 3\tau_0)$ or $(151.5625\%\tau_{FS})$, respectively. In these schemes, only single-candidate scheme can provide a useful 193% speedup. However, this scheme has the same search window size as single candidate scheme of $f_r = 1/2$ factor, named as A-1 scheme in the next section, but lower speedup. Furthermore, the more levels a pyramid model has, the more the resolution is reduced causing less-accurate results at bottom level after the propagation. This explains the reasons authors in [28, 30] recommend two-level pyramid for a good performance with FS-based hierarchical search.

7.1.3 Multi-candidate Pyramid Searches with Variable Fine-Tuning Search Windows

Based on the complexity analysis, We can increase speedup performance and number of candidate motion vectors by adjusting fine-tuning search window size at lower levels, accepting a trade-off of ME increase. Our *flexible pyramid model* will have multi-candidate motion vector search schemes with the following reducing factor f_r approaches:

1. *Approach A, reducing factor* $f_r = 1/2$: there are only two possible schemes; single-candidate and two-candidate namely, A-1 and A-2 schemes that provide speedups of 200% and 133%, respectively. The three-candidate scheme A-3 is not applicable as there is no speedup gain.

2. *Approach B, reducing factor* $f_r = 3/8$: there are up to four schemes of single-candidate, two, three and four candidate motion vectors, namely B-1, B-2, B-3, B-4 schemes that provide speedups of 256%, 188%, 149%, and 123%, respectively.

3. *Approach C, reducing factor* $f_r = 1/4$: there are up to eight schemes of single-candidate, two, three $\ldots$, namely C-1, C-2, C-3, C-4, C-5, C-6, C-7 and C-8 schemes that provide speedups of 320%, 267%, 229%, 200%, 178%, 160%, 145%, 133%, respectively.

7.1.4 Motion Estimation Efficiency Metric

Motion estimation efficiency E is proposed in Equation (7.1) to determine how good a scheme is, compared with FS.

$$E = (1 - \frac{C - C_{FS}}{C_{FS}}).100\% \tag{7.1}$$

Where C and C_{FS} are motion estimation cost of a scheme and FS, respectively. If a scheme that has a high speedup e.g. B-1, C-1, and a low *motion estimation efficiency* E indicating the resulting ME of the scheme is high, may not be valuable or practical. The efficiency reaches 100% when a scheme obtains an ME close to that of FS, and zero percent when the ME is twice that of FS.

7.1.5 Nvida CUDA Implementation

Nvida CUDA is selected to develop parallel programming for Nvida GPU. One CUDA thread is assigned to compute motion estimate cost for one MB at one search position in the search window. A $Bw.Bh$ pixel-sized search window will result in $Bw.Bh$ different search positions for each MB. One Cuda thread block is dedicated to find the best matching block for one MB. Sequential addressing with non-divergent branching [9] is employed to avoid shared memory bank conflict and divergent warps during the thread binary reduction process. Figure 7.1 shows diagram of our GPU-

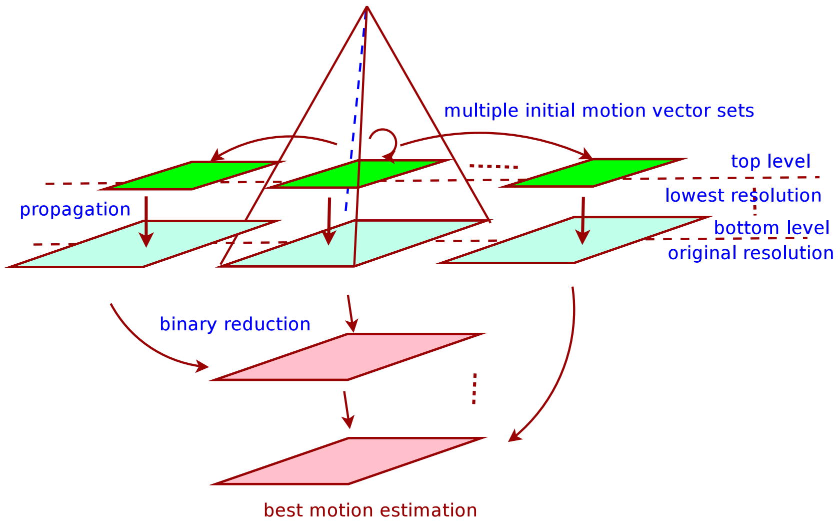


Figure 7.1: Diagram of GPU-based Multi-candidate Hierarchical Search: Multiple initial best motion vector sets are propagated and fined-tuned with FS to the bottom level. The final ME results are obtained by GPU binary reduction.

based hierarchical search approach. In the binary reduction process to obtain the initial MEs at the top level, multiple motion vector sets for all MBs corresponding to first, second . . . lowest ME cost are obtained. The number of initial motion vector sets depends on the multi-candidate scheme selected. These motion vector sets are sequentially propagated and fine-tuned towards the bottom level. At the bottom level, instead of selecting the set with the smallest total ME costs as the final ME, our approach obtains the final ME results by using GPU binary reduction, at MB level, to select the minimum ME cost for each MB. The final ME cost for the whole frame, as the result, is minimum. Algorithm 3 shows the algorithm in four main steps:

1. At top-level creating initial motion vectors, *runcuda3* kernel produces multiple

lowest result sets of motion vectors for all MBs.

2. *runcuda* kernel sequentially propagates and fine-tunes using FS all multiple initial motion vector sets to the bottom level.

3. *runCudaReduce* kernel using Cuda binary reduction obtains the best results among multiple final result sets.

4. *cudaMemcpy* kernel copies results back to host memory.

The above are the base steps to compute ME for a multi-candidate motion vector scheme. To select the best multi-candidate scheme for a video, We propose two strategies:

1. Profiling-based scheme selection. In this strategy, all faster-FS schemes with known speedups for a video are profiled. A best scheme, then is selected that will satisfy a desired speedup or efficiency.

2. Dynamic scheme selection at runtime. In this strategy, using the flexible pyramid model, a best scheme is selected at runtime to encode the next video frame based on ME results of current frames. These two strategies will be presented and evaluated in the evaluation section.

Algorithm 3 Enhanced Multi-candidate Cuda-Pyramid search

```
Require: x264 Context
Ensure: $motionVectorX, motionVectorY$
  Allocate Cuda memory for current frame - cudaMallocArray
  Allocate Cuda memory for reference frame - cudaMallocArray
  Copy current frame to Cuda memory - cudaMemcpy2DToArray
  Copy reference frame to Cuda memory - cudaMemcpy2DToArray
  Obtain multiple best initial motion vector result sets at top-level image–runCuda3
  Propagate the initial result sets to bottom level and obtain the final results for
  each initial result set -crunCuda
  Obtain the best results among multiple final result sets - runCudaReduce
  Copy results back to host memory - cudaMemcpy
```

7.2 Performance Evaluation

7.2.1 Experiment Setup

We evaluate our approaches on bulldozer-based server [40] configured as 8-node CC-NUMA [36] architecture running Ubuntu Linux 3.0.0-19, Gcc 4.6.1-9. This server is equipped with one Tesla C2050 NVIDA GPU [37]. The GPU has 448 cores, each runs at 1.15 Ghz, 2.5 GB total Memory with Fermi aarchitecture[23] that can support at most 1024 CUDA threads per each thread-block. We implement our approach on MIT's experimental open source for hierarchical research on GPU [6]. We use HD Benchmark library [2] as video benchmarks for the experiments. During the experiments, 100 raw video frames from a benchmark video are encoded with three resolutions; 4CIF, HD and FHD, using h264 standard with a standard 16-pixel search range. A full search with the same search range is used as a base case for evaluations as it provides the most accurate motion estimate. Experiment results are averages of five runs and relative standard deviations (STDEV) are reported when applicable.

7.2.2 Verification of Computational Complexity

Table 7.2: Processing Times of Top level, 1st., 2nd. and 3rd. candidate of A-3 scheme, the three-candidate motion vector scheme, match the complexity analysis

Resolution	Top	1st	2nd	3rd	FS
4CIF (ms)	2.41	2.27	2.19	2.27	9.16
HD (ms)	5.04	4.94	4.85	4.85	19.81
FHD (ms)	11.33	11.62	11.52	11.60	45.66
avg.%FS	25.52%	25.08%	24.53%	24.90%	100%
Theoretical	25%	25%	25%	25%	100%

To verify the correctness of our complexity analysis, We run experiments for A-3 scheme, a three-candidate scheme of approach A, and FS, the base case for three resolutions on Tesla GPU. According to the complexity analysis, the complexity of four periods; the initial motion estimation (τ_1) at the top level, the first (τ_0), the second (τ_0), and the third candidate (τ_0) at the bottom level of A-3 scheme, each is 25% of FS's. Table 7.2, shows processing time (ms) measured for these four periods, and FS for three video resolutions in columns *Top, 1st, 2nd, 3rd,* and FS, respectively. The average percentage of processing time of each period with respect to FS for the three video resolutions $avg.\%FS$ closely matches the theoretical values in our computation complexity analysis. This correctness is interesting and significant as it does not only prove the correctness of our complexity analysis, but also helps determine in advance the speedups of any selected schemes. In other words, when a scheme or multiple schemes are selected to encode a video, We will know exactly how fast the schemes are compared with the base case FS, or other schemes.

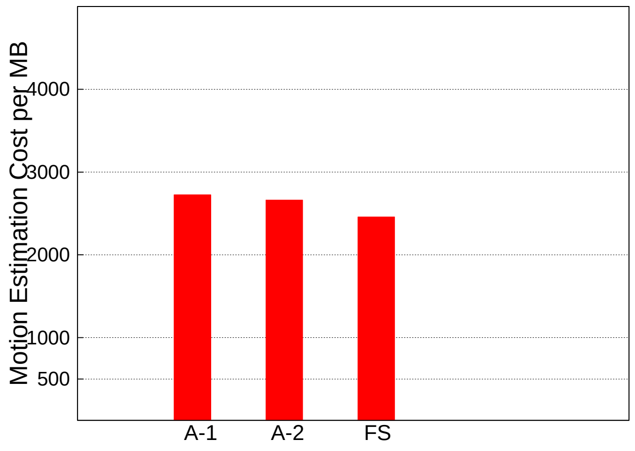
In the verification experiment, to perform FS on GPU, We use 8-pixel search range that covers a 289-pixel search window (2x8+1)x(2x8+1). In our CUDA implementation, as one thread block is dedicated to find the best matching block for one MB, our experimental Tesla GPU with Fermi architecture, capable at most 1024 threads per thread block, is not used to perform FS for 16-pixel search range since this search range will cover 1089-pixel search window that needs at least 1089-thread block size. Instead, the motion estimation cost of the base case FS, used for comparisons, is obtained by a CPU-based FS. To cover the same area as in the base case FS (16-pixel search range) our GPU-based hierarchical search only needs to search half of the 16-pixel search range since at top pyramid level, where resolution is reduced 2x2 times, a 8-pixel search range will cover the same area of 16-pixel search range at original resolution. With the verified correctness of the computational complexity analysis, the processing time of the FS on GPU for 16-pixel search range can be approximately derived. For example, knowing A-3 scheme, the three-candidate scheme in approach A, will have the same complexity as FS, We can derive processing time of FS on GPU from the processing time of A-3 scheme.

7.2.3 Motion Estimation Performance

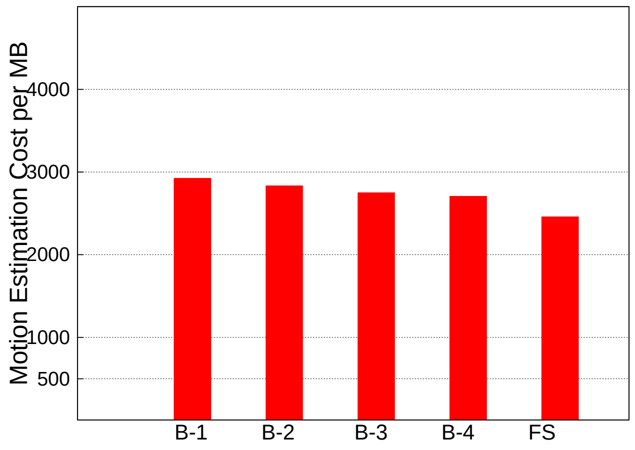
Figures 7.2a, 7.2b and 7.2c show ME per MB of three approaches in HD format. We observe the following:

1. MEs decrease along with schemes that have higher number of candidate motion vectors while FS has the lowest MEs. This is expected as more number of candidate vectors will help mitigate the problem of erroneous initial motion estimation at top level.

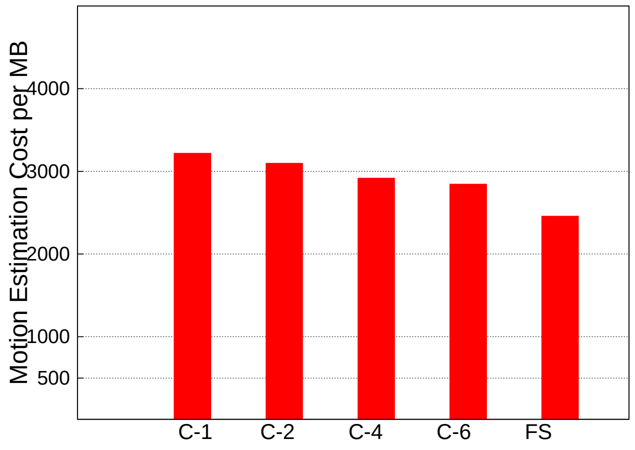
2. Schemes that have the same number of motion-vector candidates in approach A, will have loIr MEs than that of approach B and C, subsequently. For example, A-1, B-1 and C-1 schemes have one motion-vector candidate, but ME cost of A-1 is smaller than that of B-1, and C-1. It is explained that the earlier has larger fine-tuning search window that results in finding better matching blocks with lower MEs, however, with a trade-off of higher computational complexity.



(a) Approach A: single, two candidate motion vector scheme, and FS.



(b) Approach B: single, two, three, four candidate motion vector scheme, and FS.



(c) Approach C: single, two, four, six candidate motion vector scheme, and FS.

Figure 7.2: ME cost decreases as number of motion-vector candidates increases. ME cost of schemes with the same number of motion-vectors increase in approach A,B and C as search windows become smaller.

7.2.4 Speedups and Motion Estimation Efficiency Gains.

Figure 7.3, 7.4 and 7.5 show profiling results of average *motion estimation efficiency* and relative standard deviation on the benchmark video for all schemes of approach A, B and C. We observe the following:

1. Motion estimation efficiency of schemes with larger search windows (larger factor f_r) are always higher than that of the corresponding schemes, which has the same speedups and smaller search window (smaller f_r). The reason is larger f_rs results in larger fine-tuning search windows producing better efficiency.

2. For all approaches, the speedup-efficiency correlation is linear but not 1:1. In particular, it is 9.5:1. It means that the trade-off of speedup loss is not equally compensated by the efficiency gain. In each approach, the top performance is the single-candidate scheme, and any increase in the number of candidates will improve the efficiency but at multiple-timed loss of speedup.

Figure 7.6 shows processing-time breakdown of three periods in A-2 scheme, the two-candidate scheme of approach A. This scheme consists of FS for the initial motion estimation at top level, FS for the first candidate motion vector set, and that for the second candidate motion vector set at bottom level. It is interesting to observe that all three periods have nearly the same processing time of 25% of FS as discussed in the complexity analysis. These three periods constitute 75% of FS and result in 133% (1/0.75) speedup as shown in Table 7.1. The same observation is seen with complexity matches are for B-4 and C-6 schemes of approach B and C, respectively. This again verifies the correctness of our complexity analysis.

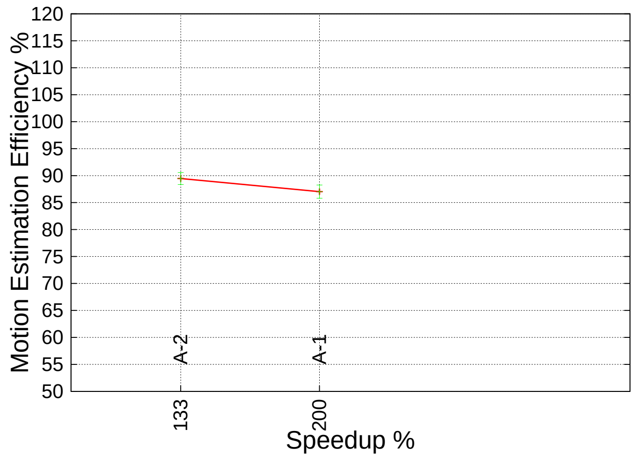


Figure 7.3: Approach A ($f_r = 1/2$): Motion Estimation Efficiencies.

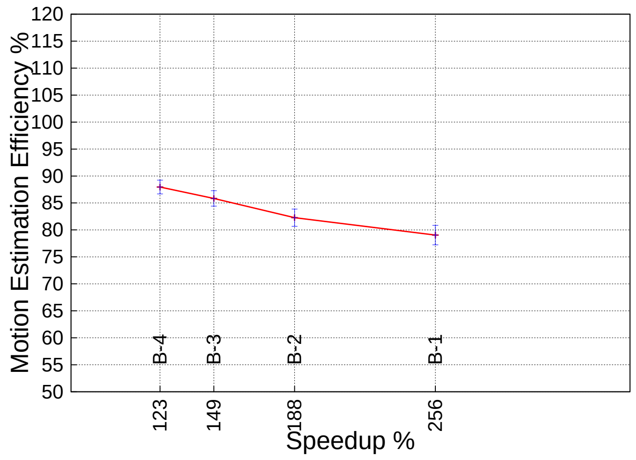


Figure 7.4: Approach B ($f_r = 3/8$): Motion Estimation Efficiencies.

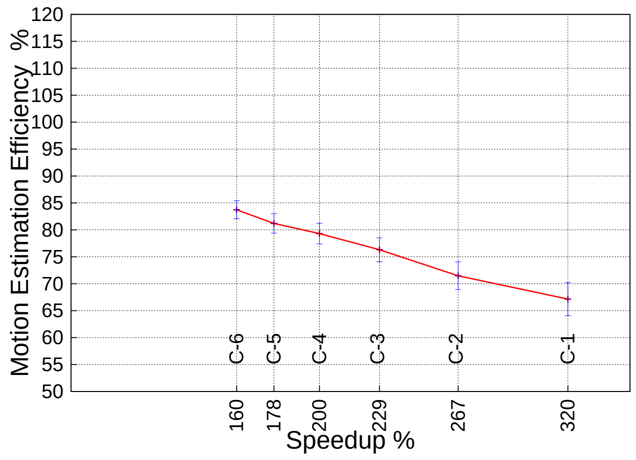


Figure 7.5: Approach C ($f_r = 1/4$): Motion Estimation Efficiencies.

7.2.5 Fixed Multi-candidate Scheme Selection

To build *profiling graph* shown in Figure 7.7, efficiencies of all schemes are merged and arranged in speedup order by removing schemes that have the same speedups but lower efficiencies. Using this *profiling graph* we can determine an efficiency, which is highest, for a video with a desired speedup, or a highest speedup scheme with a desired efficiency. We can also select a scheme that can balance a satisfied efficiency and a best possible speedup. For example, with a desired speedup of 200%, We can choose A-1 scheme that gives 87%, the highest efficiency. With a desired efficiency of 85% both A-1 and C-3 schemes can satisfy this. However, A-1 scheme (87.03% efficiency) can achieve 200% speedup, while B-3 scheme (85.84% efficiency) only achieves 149% speedup. Approach A can offer better motion estimations for a trade-off of reduced speedups, but it could only offer two schemes at most with the maximum speedup of only 200% (A-1 scheme). Approach B and C, on the other hand, have many more schemes. Among these schemes, C-1 scheme, the highest, can achieve 320% speedup.

Using profiling we can select a scheme for the entire video that can meet a desired efficiency and provide highest possible speedup. However, no fixed scheme would provide a best speedup and efficiency for all frames in a video due to the dynamic motion and diverse scene characteristics of a video. For example, in a video scene that has less motion, if a small fine-tuning search window can obtain the same efficiency as with a larger search window (A-schemes), then B or C-schemes can be employed to increase the speedups. If there is some scheme selection mechanism that can select a best multi-candidate scheme for current frames at runtime, the overall performance of motion estimation will be significantly improved. This observation motivates me to design a dynamic scheme selection, presented in the next section, to select best schemes in terms of highest possible speedup and satisfactory efficiency for current frames at runtime.

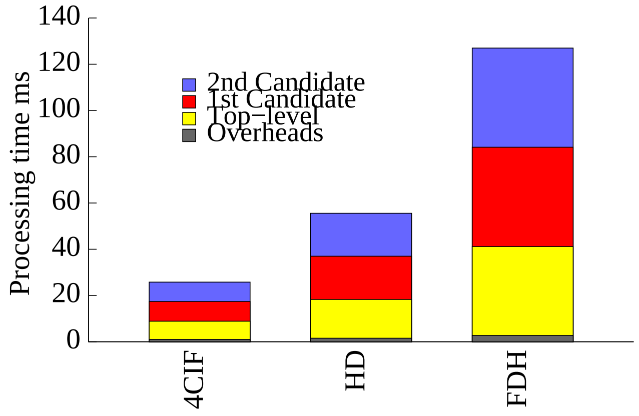


Figure 7.6: Equal processing times at top-level, 1st, and 2nd candidate, each accounts 25% of FS.

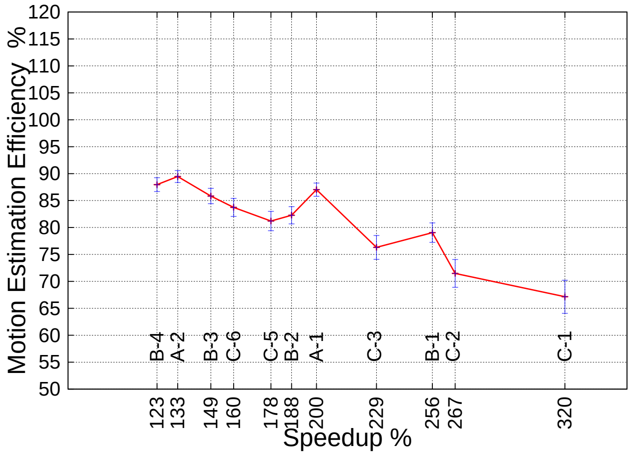


Figure 7.7: Profiling graph of schemes of three approaches in speedup order.

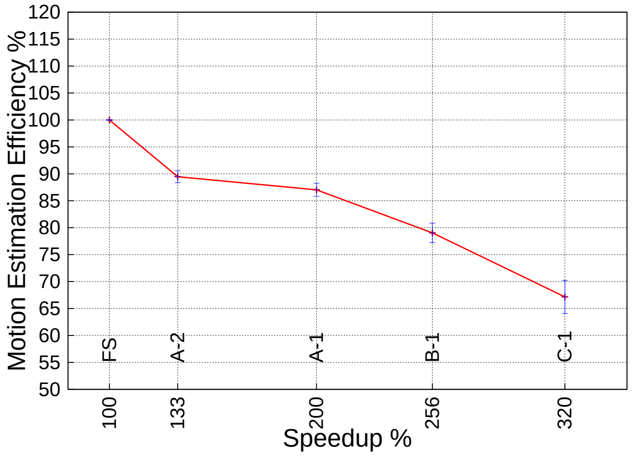


Figure 7.8: Linear scheme model graph is the upper bound of the best schemes (FS,A-2,A-1,B-1,C-1).

7.2.6 Dynamic Multi-candidate Scheme Selection

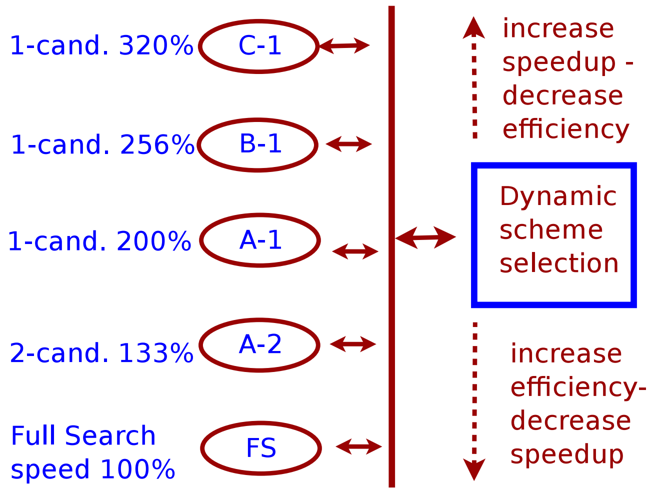


Figure 7.9: The scheme selection box slides up and down to select highest possible speedup schemes while satisfying the desired efficiency.

Algorithm 4 Dynamic Multi-candidate Scheme Selection

```
Require: $schemeTable, desireEfficiency, interval$
Ensure: $schemeSelect$
  $totalSpeedupIncrease, totalEfficiencyReduce \leftarrow 1$
  while not.EOF do
    if $interval$ number of frames has passed since last FS then
      $FSCost, MECost \leftarrow FS(currFrame);$
    else
      $MECost \leftarrow hierachcialSearch(currScheme, currFrame);$
    end if
    $currEfficiency \leftarrow CompEfficiency(MECost, FSCost)$
    $currSpeedup \leftarrow SpeedupLookup(currScheme)$
    $totalSpeedupIncrease + \leftarrow (currSpeedup - 100)$
    $diffEfficiency \leftarrow currEfficiency - desireEfficiency$
    $totalEfficiencyReduce + \leftarrow diffEfficiency$
    $nextSpeedup \leftarrow currSpeedup + diffEfficiency.\frac{totalSpeedupIncrease}{totalEfficiencyReduce}$
    $schemeSelect \leftarrow SchemeLookup(nextSpeedup)$
  end while
  return
```

From Figures 7.3, 7.4 and 7.5, we observe that in the three approaches, motion es-

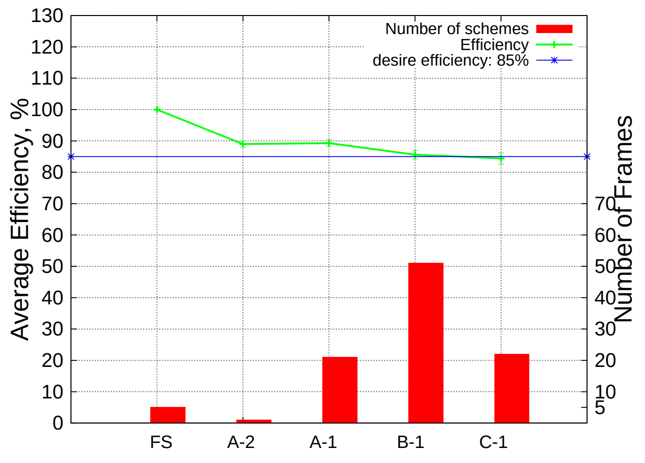


Figure 7.10: Efficiencies and number of frames selected by the dynamic selection.

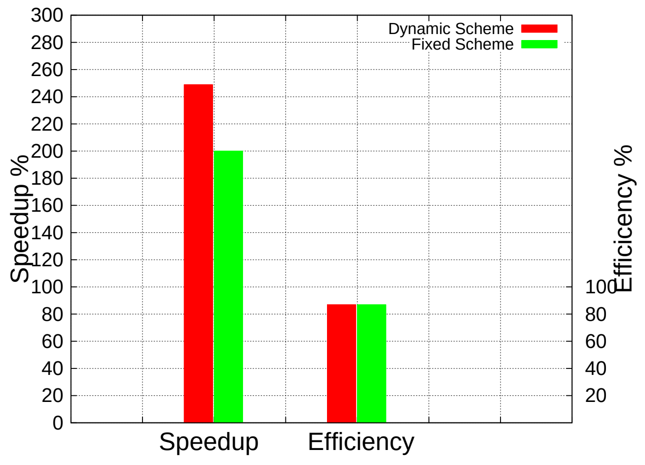


Figure 7.11: Fixed scheme selection can match the same efficiency of the dynamic selection but has lower speedup.

timation efficiency of a scheme has linear correlation with speedup within its approach only e.g. A or B, but not across approaches. When efficiencies of all these schemes are merged in speedup order as shown in the *profiling graph*, Figure 7.7, the local minimums, which are schemes with lowest efficiencies, are exposed that causes a non-linear correlation. Designing a dynamic scheme selection on this non-linear correlation may need a learning algorithm that is outside the scope of this research. To design a linear scheme selection across approaches we build a *linear scheme model* that excludes the local minimums, but includes schemes with maximum efficiencies. The resulting *linear scheme model* shown in Figure 7.8 is the upper bound of the *profiling graph* that consists of the following schemes: FS(100%), A-2(133%), A-1(200%), B-1(256%), C-1(320%). Figure 7.9 shows the diagram of a dynamic scheme selection process among the five candidate schemes from all three approaches. The selection algorithm selects scheme for the next frame based on the status of motion estimation efficiency of the current scheme at the current frame. If the acquired efficiency is greater than the desired efficiency, the selection algorithm will select a new scheme with a higher speedup for the next frame, trading-off its current high efficiency. Otherwise, a scheme with a lower speedup, expecting a higher efficiency, is selected to meet the *desired efficiency*. The speedup increment or decrement is proportional to how better and worse the current efficiency is, compared with the desired value. The diagram illustrates the dynamic selection process, in which the *dynamic scheme selection box* slides up and down from FS to C-1 scheme to get the highest possible speedup schemes while satisfying the desired efficiency. The selected scheme will be the best since it selects schemes among the best schemes at runtime. Algorithm 4 shows the pseudo code of the dynamic scheme selection. The FS is conducted every *interval* number of frames since the last FS, to get the FS cost $FSCost$ that is used as a base cost to approximately determine the current efficiency $currEfficiency$ of the current scheme. Based on the current efficiency compared with the desired efficiency, the algorithm will determine the $nextspeedup$ for the next frame based on how much increment or decrement compared with the current speedup. This amount is proportional to the ratio between the cumulative total speedup increment $totalSpeedupIncrease$ and the cumulative total efficiency reduction $totalEfficiencyReduce$. An accurate current efficiency can be achieved by conducting other FS on the current frame, but it will cause an impractical overhead for doubling motion estimation search. Increasing the $interval$ will reduce number of times the FS is used, indirectly increasing the speedup, but may result in more erroneous efficiency for frames far away from the frames that use FS. To reduce the effect of inaccurate current efficiency on the total speedup increment $totalSpeedupIncrease$ and the cumulative total efficiency reduction $totalEfficiencyReduce$, these values only accumulate speedup increment and efficiency reduction within $maxframeCtr$ number of frames after the FS. In the experiment, we select 25-frame $interval$ that is equivalent to one-second period, and $maxframeCtr = 5$. For simplicity, the $maxframeCtr$ parameter is not mentioned in the pseudo code. Given the proposed next speedup, a corresponding scheme select is looked up from the *linear scheme model*.

Figures 7.10 and 7.11 show performance of the dynamic scheme selection at runtime. Figure 7.10 shows number of frames and average efficiency of schemes selected with 85% desired efficiency for the experimental video. All selected schemes have average efficiency higher than the desired value, except C-1 scheme has average efficiency of 84.4%, which is very close to the desired efficiency (85%). To make sure all schemes have efficiencies higher than the desired one, we can select a higher desired efficiency to compensate for the FS, which has 100% efficiency, must be performed, at least, every interval number of frames. Figure 7.11 shows comparisons of speedup and efficiency gains between the two scheme selections. The dynamic scheme selection offers an overall speedup and efficiency of 249% and 87%, respectively, while with the fixed scheme selection, A-1 scheme, the best fixed scheme selected, can provide the same average efficiency, but its known speedup is only 200%. In our experiment, the dynamic scheme selection outperforms speedup of the fixed scheme selection 25%. In this research, we propose a design of the *flexible pyramid model, fixed and dynamic scheme* based on three *reducing factors*. More factors can be considered to have a smoother *profiling graph* and *linear scheme model*.

7.2.7 QoS Performance

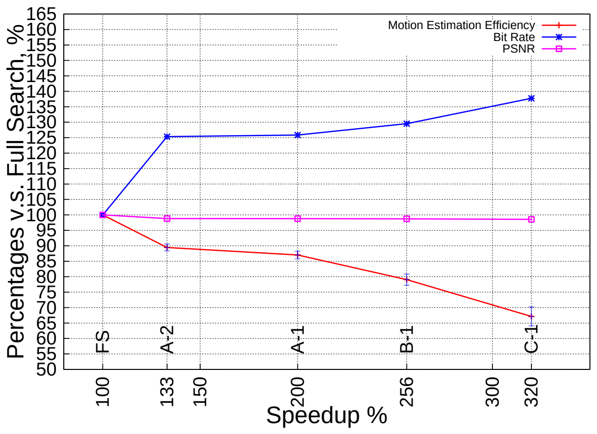


Figure 7.12: Percentages of bit rate and Peak-Signal-To-Noise Ratio (PSNR) of different schemes with respect to the full search base case. Bit rate is correlated with speedups, while PSNR is not much different among the schemes.

Figure 7.12 shows percentages of Peak Signal-To-Noise Ratio (PSNR) and bit rate of different schemes with respect to full search base case FS. Speedup gains for a trade-off of the bit-rate increase. In particular with our benchmark, scheme A-1, B-1 and C-1 (200%, 256% and 320% speedups) have bit rates increased 25%, 30% and 37%, respectively. Quality of video encoding measured by the PSNR, is less affected by the speedups. In our experiments, we observe that the PSNR is reduced around 1% compared to FS and no different among multi-candidate hierarchical search schemes. Close to our observation, authors in [22, 7, 17, 28] report that PSNR in their hierarchical searches are close to that of FS.

7.3 Summary

In this chapter, we propose a design of a flexible pyramid model, which can increase the speedup by reducing the computational complexity but keep the motion estimation cost low. Based on a correctness verification of the computational complexity of the multi-candidate motion-vector hierarchical search, we can accurately determine speedup of a selected motion-estimation hierarchical search scheme. We propose a motion estimation efficiency metric to evaluate a search scheme as low efficiency indicates a high motion estimate causing a a high bit-rate encoded stream. Based on the motion estimate efficiency, we propose profiling-based fixed, and a runtime-based dynamic multi-candidate scheme selections. The fixed profiling-based selection is able to select a best fixed multi-candidate motion vector scheme while the runtime-based selection is able to dynamically select best candidate motion vector schemes at runtime. Evaluating our approach using video benchmark, we observe that the runtime-based selection scheme outperforms the profiling-based fixed scheme selection, which is already good since it selects the best among schemes. This research has been published in IPCCC 2012, in Austin, Texas [34].

Chapter 8

Conclusion

8.1 Summary

In this dissertation, we address the problem of efficient parallel processing of multimedia on multi-core architectures with a focus on parallel encoding and decoding on shared memory multi-core architecture, and parallel motion estimation hierarchical search, one of the most expensive functions (86%) in video encoding, on GPGPU. We analyze parallelism granularity at GOP, frame, slice, and select macro-block level parallelism for highest parallelism. Parallel processing scheduling schemes are classified by static and dynamic approaches, where MBs are scheduled in static order or dynamically at runtime.

Studying static wave-front parallelization, we can determine maximum processing speedups, which can be used as performance metrics to evaluate an encoder or a decoder. We identify overhead of barrier-based synchronization that significantly affects the performance of video processing, and reduce CPU utilization. *Wave-front* parallelization achieves a maximum speedup of 4X compared with a single-threaded encoding [25]. A novel *dynamic scheduling* scheme with tail submit provides a good throughput performance by alleviating synchronization overhead, and mitigating access contention at the shared task queue. In this approach, current threads keep processing the next ready MBs that significantly reduces workload in the shared queue and thus alleviates the contention problem. Experiments with video benchmarks the scheme achieves a maximum speedup of 10.4X. Nevertheless, the *dynamic scheduling* still use a global queue, and scheduling is not load-balanced and aware of cache-locality of the underlying multi-core architecture.

We propose a *distributed dynamic scheduling* scheme that employs *distributed LIFO queues* to reduce the access contention at the global queue, and schedule MB tasks in a cache-locality and load-balancing fashion using *minimum weighted-queue length* algorithm so that neighboring macro-blocks are load-balanced to nearby cores. We implement our proposed scheduling on an encoder and a decoder using OpenMP and POSIX threads on a shared memory multi-core SGWe server, and carry out extensive experiments with video benchmarks. Experiment results shows employing the *distributed dynamic scheduling* that our encoder/decoder outperforms wave-front and tail-submit dynamic scheduling 100% and 20%-25%, respectively.

In motion-estimation hierarchical search on GPGU, we propose a flexible multiple-level hierarchical model with variable sizes of fine-tuning search windows that can increase speedups by reducing the computational complexity. We also propose an important motion estimation efficiency metric that evaluates how efficient a multi-candidate scheme is, compared with FS. A multi-candidate scheme that may provide a substantial speedup, but would not be worth if its efficiency is insufficient. We propose profiling-based multi-candidate scheme selection, in which all better-than-FS multi-candidate schemes are profiled. A best scheme for a video can be selected that will achieve a highest possible speedup and satisfy a desired efficiency. We also propose an robust runtime-based multi-candidate scheme selection that dynamically selects a flexible multi-candidate scheme for the next frame at runtime based on speedup and efficiency of current scheme. We observe that the runtime-based scheme selection outperforms the profiling-based fixed scheme selection that is already good since it selects the best among the schemes.

8.2 Future Research

Our future work will be evaluating the distributed dynamic scheduling with High Efficiency Video Encoding (HEVC), a recenlty released video compression standard. Our future work also focuses on GPU-based accelerator, where many parallelizable functions such as inverse transform, inverse quantization, motion compensation, intra prediction, de-blocking filter, are off-loaded to GPGPU.

Bibliography

[1] H264/avc jm reference software. `http://iphome.hhi.de/suehring/tml/`, 2014. [Online; accessed 04-1-2014].

[2] Mauricio Alvarez, Esther Salami, Alex Ramirez, and Mateo Valero. Hd-videobench. a benchmark for evaluating high definition digital video applications. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 120–125. IEEE, 2007.

[3] Guy Amit and Adi Pinhas. Thread-level parallelism: Gains and pitfalls. In *IASTED PDCS*. IEEE, 2005.

[4] Juurlink B. Meenderinck et. al. Azevedo, A. A highly scalable parallel implementation of h.264. *In Transactions on High-Performance Embedded Architectures and Compilers IV*, pages 111–134, 2011.

[5] Juan Carlos Arevalo Baeza, William Chen, Eric Christoffersen, Daniel Dinu, and Barry Friemel. Real-time high definition h. 264 video decode using the xbox 360 gpu. In *Optical Engineering+ Applications*. International Society for Optics and Photonics, 2007.

[6] Lawrence Chan, J Lee, Alex Rothberg, and Paul Weaver. Parallelizing h. 264 motion estimation algorithm using cuda. *Proc. of Independent Activities Period (IAP), MIT*, 2009.

[7] Y-L et al Chan. Adaptive multiple-candidate hierarchical search for block matching algorithm. *Electronics Letters*, pages 1637–1639, 1995.

[8] Gabriele Jost Chapman, Barbara and Ruud Van Der Pas. Using openmp:portable shared memory parallel progamming. *The MIT Press*, 2008.

[9] W. Chen and H. Hang. H.264/avc motion estimation implementation on compute unified device architecture (cuda). In *Multimedia and Expo, 2008 IEEE International Conference on. IEEE*. IEEE, 2008.

[10] Meenderinck Cor, Arnaldo Azevedo, Ben Juurlink, Mauricio Alvarez Mesa, and Alex Ramirez. Parallel scalability of video decoders. *Journal of Signal Processing Systems*, 57(2):173–194, 2009.

[11] Azevedo et. al. Ffmpeg-2dwave: A parallel h.264 decoder. `http://alvarez.site.ac.upc.edu/hdvideobench/ffmpeg-2dwave.html`, 2014. [Online; accessed 04-1-2014].

[12] Jiani Guo and Laxmi N. Bhuyan. A multithreaded multicore system for embedded media processing. *Transactions on HiPEAC*, page 2011, 154-173.

[13] Jiani Guo and Laxmi N Bhuyan. Load balancing in a cluster-based web server for multimedia applications. *Parallel and Distributed Systems, IEEE Transactions on*, 17(11):1321–1334, 2006.

[14] Wen-mei Hwu and David Kirk. Programming massively parallel processors. *Special Edition*, page 92, 2009.

[15] Intel. Thread building block. `http://threadingbuildingblocks.org`, 2014. [Online; accessed 04-1-2014].

[16] Seongmin Jo, Song Hyun Jo, and Yong Ho Song. Exploring parallelization techniques based on openmp in h. 264/avc encoder for embedded multi-core processor. *Journal of Systems Architecture*, 2012.

[17] J. Ra K. Lim. Improved hierarchical search block matching algorithm by using multiple motion vector candidates. *IEEE Journal of Electroonic Letters*, pages 1771–1772, October, 1997.

[18] Peter Lambert, Wesley De Neve, Peter De Neve, Ingrid Moerman, Piet Demeester, and Rik Van de Walle. Rate-distortion performance of h. 264/avc compared to state-of-the-art video codecs. *Circuits and Systems for Video Technology, IEEE Transactions on*, 16(1):134–140, 2006.

[19] Marie Cadennes Massanes, Francesc and Jovan G. Brankov. Cuda implementation of a block-matching algorithm for multiple gpu cards. *IIT, Med. Imaging Research Ctr.*, 2010.

[20] Mauricio Alvarez Mesa, Alex Ramírez, Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, and Mateo Valero. Scalability of macroblock-level parallelism for h. 264 decoding. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 236–243. IEEE, 2009.

[21] Mit. Cilk project. `http://supertech.csail.mit.edu/cilk`, 2014. [Online; accessed 04-1-2014].

[22] Joon-Seek Kim Rae-Hong Park Nam, Kwon Moon and Young Serk Shim. A fast hierarchical motion vector estimation algorithm using mean pyramid. *IEEE Trans. Circuits and Systems for Video Technology*, pages 344–351, 1993.

[23] Nvida. Fermi cuda architecture. `http://www.nvidia.com/object/fermi-architecture.html`. [Online; accessed 04-1-2014].

[24] Iain E. Richardson. The h.264/avc video coding standard, second edition. *John Wiley and Sons, Ltd*, page 12, 2010.

[25] António Rodrigues, Nuno Roma, and Leonel Sousa. p264: open platform for designing parallel h. 264/avc video encoders on multi-core systems. In *Proceedings of the 20th international workshop on Network and operating systems support for digital audio and video*, pages 81–86. ACM, 2010.

[26] Ralf M. Schreier Michael Bleyer Seitner, Florian H. and Margrit Gelautz. Evaluation of data-parallel splitting approaches for h.264 decoding. In *Proceedings of Mobile Computing and Multi-media, MoMM*, pages 401–404. IEEE, 2008.

[27] SGI. Sgi altix 4700. `http://www.sgi.com/products/remarketed/servers/altix4700.html`, 2014. [Online; accessed 04-1-2014].

[28] Yun-Qing Shi and X Xia. A thresholding multiresolution block matching algorithm. *Circuits and Systems for Video Technology, IEEE Transactions on*, 7(2):437–440, 1997.

[29] A.Hirano T. Koga, K. Linuma. Motion-compensated interframe coding for video conferencing. In *in Proc. Nat. Telecom. Con.*, pages G5.3.1–G5.3.1–5. IEEE, 1981.

[30] Michael G. Strintzis Tzovaras, Dimitrios and Haralambos Sahinoglou. Evaluation of multiresolution block matching techniques for motion and disparity estimation. *Signal Processing: Image Communications*, pages 56–57, 1994.

[31] Egbert G. Jaspers Van Der Tol, Erik B. and Rob H. Gelderblom. Mapping of h.264 decoding on a multiprocessor architecture. *Electronic Imaging 2003. International Society for Optics and Photonics*, 2003.

[32] Dung Vu, Jeremy Castillo, and Laxmi N. Bhuyan. An adaptive dynamic scheduling scheme for h.264/avc encoding on multicore architecture. In *ICME*. IEEE, 2014.

[33] Dung Vu, Jilong Kuang, and Laxmi N. Bhuyan. An adaptive dynamic scheduling scheme for h.264/avc decoding on multicore architecture. In *ICME*, pages 491–496. IEEE, 2012.

[34] Dung Vu, Yang Yang, and Laxmi N. Bhuyan. An efficient dynamic multiple-candidate motion vector approach for gpu-based hierarchical motion estimation. In *IPCCC*, pages 342–351. IEEE, 2012.

[35] Sung-Wen Wang, Shu-Sian Yang, Hong-Ming Chen, Chia-Lin Yang, and Ja-Ling Wu. A multi-core architecture based parallel framework for h. 264/avc deblocking filters. *Journal of Signal Processing Systems*, 57(2):195–211, 2009.

[36] Wiki. Non-uniform memory access. `http://en.wikipedia.org/wiki/Non_Uniform_Memory_Access`. [Online; accessed 04-1-2014].

[37] Wiki. Nvidia tesla. `http://en.wikipedia.org/wiki/Nvidia_Tesla`. [Online; accessed 04-1-2014].

[38] Wiki. Posix threads. `http://en.wikipedia.org/wiki/POSIX_Threads`. [Online; accessed 04-1-2014].

[39] Wiki. Moncecito processor. `http://en.wikipedia.org/wiki/Montecito_%28processor%29`, 1999. [Online; accessed 04-1-2014].

[40] Wiki. Bulldozer (microarchitecture). `http://en.wikipedia.org/wiki/Bulldozer_(microarchitecture)`, 2014. [Online; accessed 04-1-2014].

[41] Wiki. Context-adaptive binary arithmetic coding. `http://en.wikipedia.org/wiki/Context-adaptive_binary_arithmetic_coding`, 2014. [Online; accessed 04-1-2014].

[42] Wiki. Discrete cosine transform. `http://en.wikipedia.org/wiki/Discrete_cosine_transform`, 2014. [Online; accessed 04-1-2014].

[43] Wiki. Divx codec. `http://en.wikipedia.org/wiki/DivX`, 2014. [Online; accessed 04-1-2014].

[44] Wiki. Mean absolute difference. `http://en.wikipedia.org/wiki/Mean_absolute_difference`, 2014. [Online; accessed 04-1-2014].

[45] Wiki. Openmp threads. `http://en.wikipedia.org/wiki/OpenMP`, 2014. [Online; accessed 04-1-2014].

[46] Wiki. Quantization (image processing). `http://en.wikipedia.org/wiki/Quantization_(image_processin`, 2014. [Online; accessed 04-1-2014].

[47] Zhuo Zhao and Ping Liang. Data partition for wavefront parallelization of h. 264 video encoder. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on.* IEEE, 2006.

[48] Zhuo Zhao and Ping Liang. A highly efficient parallel algorithm for h. 264 video encoder. In *Acoustics, Speech and Signal Processingm ICSSP*. IEEE, 2006.