Computer Science Department
University of Crete

Design and Evaluation of a Task-based Parallel H.264 Video Encoder for the Cell Processor

Master's Thesis

Michail Alvanos

June 2010

Heraklion, Greece

University of Crete
Computer Science Department

Design and Evaluation of a Task-based Parallel H.264 Video Encoder for the Cell Processor

Thesis submitted by
Michail Alvanos
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Michail Alvanos

Committee approvals: _____
Angelos Bilas
Associate Professor, Thesis Supervisor

_____
Dimitrios S. Nikolopoulos
Associate Professor

_____
Manolis G.H. Katevenis
Professor

Departmental approval: _____
Panos Trahanias
Professor, Director of Graduate Studies

Heraklion, June 2010

Abstract

Modern multi-core processors with explicitly managed local memories, such as the Cell Broadband Engine (Cell) constitute in many ways a significant departure from traditional high performance CPU designs. Such CPUs, on one hand bear the potential of higher performance in certain application domains and on the other hand require extensive application modifications.

We design and implement `x264`, a complete H.264 video encoder for the Cell processor, based on an open source H.264 library, *c264*. Our implementation achieves speedups of 4.5x on six synergistic processing elements (SPEs), compared to the serial version running on the power processing element (PPE). Our work considers all parts of the encoding process and reveals related limitations. `x264` constitutes an extensive redesign of the original *c264* code to employ fine-grain parallelization to cope with the small size of the local memory in the SPEs and achieve replication and privatization of shared data structures due to the non-coherent Cell architecture.

Our analysis allows us to identify the main limitations for further scaling H.264 video encoding on future multi-cores: (a) overheads for task management cause a heavy burden on the single master processor, (b) complex control flow in the code limits effective parallelism, and (c) small on–chip memories limit the overlap of communication and computation.

Supervisor professor: Angelos Bilas

Περίληψη

οντέρνοι πολυπύρηνοι επεξεργαστές με ρητά διαχειριζόμενες τοπικές μνήμες, όπως ο επεξεργαστής Cell Broadband Engine (Cell), αποτελούν από πολλές απόψεις ένα σημαντικό σημείο στην σχεδίαση επεξεργαστών για υψηλές επιδόσεις. Ο εν λόγω επεξεργαστής, από την μία πλευρά προσφέρει υψηλές επιδόσεις σε συγκεκριμένες εφαρμογές και από την άλλη πλευρά απαιτεί εκτεταμένες τροποποιήσεις στην εφαρμογή.

Σχεδιάσαμε και υλοποιήσαμε την εφαρμογή *c264* για τον επεξεργαστή Cell. Η εφαρμογή *c264* αποτελεί μια πλήρη υλοποίηση για συμπίεση βίντεο H.264, βασισμένη στη βιβλιοθήκη ανοικτού λογισμικού `x264`. Η υλοποίηση μας επιτυγχάνει επιτάχυνση 4.5x σε έξι synergistic processing elements (SPEs), σε σύγκριση με τη σειριακή εκτέλεση της εφαρμογής στην κεντρική επεξεργαστική μονάδα power processing element (PPE). Η υλοποίηση μας λαμβάνει υπόψιν όλα τα κομμάτια της συμπίεσης και αποκαλύπτει συναφείς περιορισμούς. Η εφαρμογή *c264* είναι αποτέλεσμα ανασχεδιασμού της αρχικής εφαρμογής `x264`, ώστε να επιτύχουμε παραλληλοποίηση με λεπτό καταμερισμό υπολογισμών μεταξύ εργασιών για να αντιμετωπίσουμε το μικρό μέγεθος της τοπικής μνήμης των SPEs και στην αλλαγή των κοινών δομών λόγω της μη συνεκτικής μνήμης του επεξεργαστή Cell.

Η ανάλυσή μας επιτρέπει να εντοπίσουμε τους κύριους περιορισμούς για την περαιτέρω κλιμάκωση της παράλληλης συμπίεσης βίντεο H.264 για μελλοντικούς επεξεργαστές πολλών πυρήνων: (Α) η επιβάρυνση για τη διαχείριση των εργασιών μπορεί να προκαλέσει μεγάλη μείωση επιδόσεων κεντρικού επεξεργαστή, (Β) σύνθετη ροή ελέγχου στον κώδικα περιορίζει τον βαθμό του διαθέσιμου παραλληλισμού, και (Γ) μικρές on–chip μνήμες περιορίζουν την επικάλυψη της επικοινωνίας με τον υπολογισμό.

Επόπτης καθηγητής: Άγγελος Μπίλας

Acknowledgments

I would like to thank all the people who made this work possible. First, i would like to thank my supervisors Dr. Angelos Bilas and Dimitris Nikolopoulos for their assistance and guidance during this work. I would like to extend my gratitude to all the people of the laboratory that help me during the various stages of my thesis. George Tzenakis for the endless hours of debugging, Konstantinos Koukos for performance debugging and Polyvios Pratikakis for useful discussions during the course of this work. I would also like to thank the staff members Manolis Marazakis, Stavros Passas and Michalis Ligerakis, for providing their technical expertise on many issues.

I would like to thank the people that surrounded me in the CARV Laboratory all these years: Yiannhs Klonatos, Zoe Sebepou, Yiannis Manousakis Thanos Makatos, Markos Fountoulakis, Yiannis Kessapidis, Dimitris Tsaliagos, George Nikiforos, and Dimitris Apostolou. I would also like to thank all of my friends for the encouragement given in the hard times: Matthaios Kavalakis, Maria Zaharaki, and Michaella Likopanti. Last but not least, I would like to thank my family for their support in many aspects.

I would like to thank the Barcelona Supercomputing Center (BSC) and the Polytechnic University of Catalonia (UPC) for making available the QS20/QS21 boards for performance debugging experiments. Finally, i would also like to mention that this work was supported by the European Commission through the SARC IP (Contract No. SARC-27648) and HiPEAC NoE (Contract No. IST-004408 and IST-217068) projects.

Michail Alvanos
Heraklion, June 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multi-core processors with many, relatively simple cores and explicitly managed memories are becoming an important design pattern for high performance computing architectures [16, 19]. These processors present three important problems to software developers: First, software needs to manage explicitly and efficiently the memory hierarchy, preferably with little or no involvement from programmers. Second, the existence of many, simpler cores, often requires fine-grain task parallelism with simpler units of work, as opposed to the coarse-grain parallelization used so far. This implies parallelization using small units of work and often extraction of more parallelism than using coarse-grain approaches. Third, exploiting fine-grain parallelism necessitates efficient runtime system support that minimizes overheads on the critical path and maximizes the utilization of the available on–chip memory bandwidth. All three problems remain challenging despite the effort that has been invested on them in the recent past.

We address the three problems in the context of a video encoder. Video encoders are essential components for real-time video processing on portable devices, such as cell phones, PDAs, and video cameras, as well as on personal computers and servers. With increasing application requirements on video resolution and frame rates, video encoding has become an extremely demanding application.

H.264 video encoding is a complex, multi-phase process, with high memory bandwidth requirements, challenging to parallelize efficiently both at the algorithmic and system level. H.264 video encoding has been parallelized in the past for shared memory multiprocessors, using coarse-grain parallelization strategies [36, 24]. These strategies are not appropriate for multi-core processors with small explicitly managed memories, due to their large memory requirements. On the other hand, fine grain parallelism is architecture independent because it is expressed in terms of the application and problem size. Furthermore, it achieves good load balancing by issuing a large number of tasks per core. Modern multi-core processors with explicitly managed memories have fast on–chip communication mechanisms, that allow efficient exploitation of fine-grain parallelism.

This work is the first to present a full system implementation of a H.264 video

encoder using task-based parallelism on the Cell processor. Our parallel implementation uses a master-worker execution model, which maps efficiently to the heterogeneous Cell processor architecture: Control code for generating tasks, tracking task dependencies, and scheduling runs on the control-efficient PowerPC core, whereas the computationally-intensive components are offloaded on the compute-efficient synergistic processing elements. The Cell is a heterogeneous multi-core processor with explicitly managed local memories. The limited size of these memories prevents the frame-level parallelization of `x264` [31], the encoder on which our implementation of H.264 relies. We address this using fine-grain, intra-frame parallelization, which allows efficient management of local memories by privatizing and replicating data structures, but makes it challenging to maximize available parallelism due to the complex control flow and data flow in the application.

Overall, our results show that although *c264* achieves speedups of about 4.5x on six SPEs for many realistic scenarios compared to the PPE-only serial execution, this requires a number of application and architecture-specific optimizations. Our implementation of *c264* achieves up to 65 and 13 frames per second (fps) for 720×576 and 1920×1080 input resolutions respectively, while running the full encoding process with six SPEs. Performance is sensitive to the input and algorithmic choices, and occasionally incurs significant idle times and communication overhead in the SPEs, as well as task wait and stall time in the PPE. Finally, we show that using dynamic scheduling can increase achievable speedup up to 3.5 compared to the PPE-only. In addition to scheduling and offloading serial code, different memory optimizations such as using large memory pages, can improve the speedup up to 4.5x compared to the PPE-only.

This thesis makes the following contributions:

1. We present *c264*, a complete, running version of the encoder for the Cell processor.

2. We describe the difficulties caused by explicitly managing memory in the Cell processor, as well as the solutions we tried.

3. We tested our implementation thoroughly, and we present a quantitative analysis of the main bottlenecks of the application and memory optimizations that affect *c264* performance.

4. We estimate the programming effort associated with building *c264*. Privatization of data structures requires significant programmer effort and discourage the port of complex applications on the Cell, despite its computational power.

The rest of this thesis is organized as follows. Chapter 2 presents the design and implementation of *c264*. Chapter 3 presents and discusses our experimental platform and results. Chapter 4 refers to related work. Finally, we summarize our work and draw conclusions in Chapter 5.

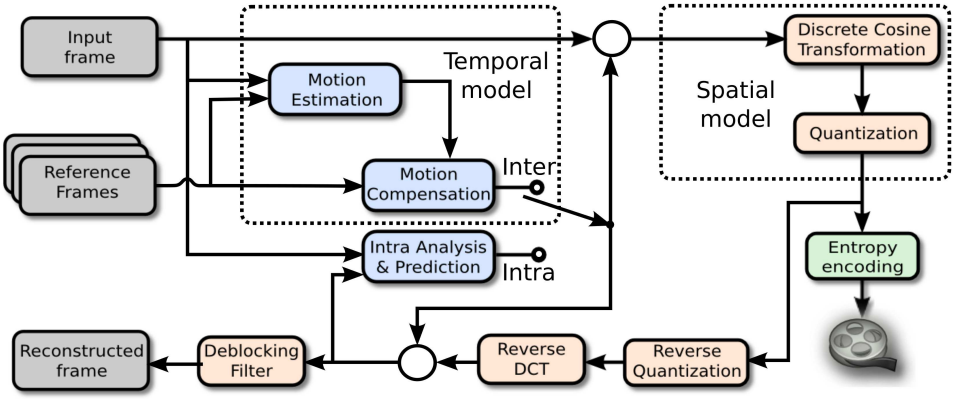

FIGURE 1.1: Block diagram of H.264 video encoding.

1.1 H.264 and x264 overview

H.264 [1] is a video compression standard, also known as MPEG-4 Advanced Video Coding (AVC). Video encoders take as input a raw, uncompressed video stream and process it picture by picture. For each picture (or frame), the encoder identifies differences from one or more previously processed frames, called reference frames. The resulting output is an encoded video stream including the reference frames and the differences required to reconstruct all dependent frames. H.264 imposes a structure on each frame, as follows. Each frame is divided into non-overlapping macroblocks (MBs). A typical size for macroblocks is 16×16 pixels. Within each frame, macroblocks are grouped into slices. The output, encoded stream contains information about frames, slices, and macroblocks.

Like most MPEG video encoders, H.264 consist of three main functional units: a temporal model, a spatial model, and an entropy encoder. Figure 1.1 shows the H.264 video encoder block diagram. The temporal model identifies similarities between macroblocks in a single or multiple neighboring frames using *motion estimation* (ME). Motion estimation determines *motion vectors* that describe how one macroblock is derived by transforming one or more reference macroblocks. Motion estimation algorithms vary both in the way they select motion vectors as well as the shape of the region they explore in each reference frame. Motion estimation identifies a region in the reference frames that minimizes a matching criterion and marks it as the "best match". The H.264 standard allows the use of up to 16 reference frames for motion estimation. The encoder, in addition to the temporal model, also employs intra-frame analysis and selects intra-frame encoding when its bitrate cost is lower than that of inter-frame analysis, to improve the overall coding efficiency. Intra-frame analysis tries to locate similarities between the current macroblock and its neighbors within the same frame. The video encoder subtracts the selected, best matching region in the reference frame(s) from the

current macroblock to produce a new macroblock (*motion compensation*), which in turn is encoded and transmitted together with a motion vector describing the position of the best matching region.

The spatial model transforms macroblock differences using the discrete cosine transformation (DCT) and generates a set of coefficients that are then quantized. In addition, the spatial model includes a H.264 decoder for the following reason: If an encoded frame needs to be used as reference for encoding other frames, it is better to use a version of the frame that is derived by decoding the encoded frame, rather than the original raw input frame. This leads to better quality video streams. Thus, the spatial model after encoding each frame, decodes and stores the reconstructed frame in memory to use it as reference for subsequent frames. A *deblocking filter* is applied to every decoded macroblock to reduce blocking distortion, created from quantization. This filter aims at improving visual quality and prediction performance by smoothing sharp edges between macroblocks.

Macroblocks encoded using *only* macroblocks of the same frame are called intra coded (I-type) macroblocks, while macroblocks that are encoded using macroblocks of other frames *also* are called either predicted-type (P-type) or bidirectionally-predictive (B-type) macroblocks. P-type pictures use temporal redundancies from a past I- or P-frame, whereas B-type macroblocks use both past and future reference frames, and consequently achieve the highest compression rate. Each P- and B-type frame can contain I-type macroblocks.

Finally, an entropy encoder combines the quantized coefficients and motion vectors in a single stream and encodes it using one of two approaches: Either context-based adaptive variable length coding (CAVLC) or context-based adaptive binary arithmetic coding (CABAC). Entropy encoding has multiple probability modes for different contexts. For each bit, the encoder selects which probability model to use and then applies arithmetic coding to compress the data. The video encoder then encapsulates the output stream in H.264 packets, called Network Abstraction Layer (NAL) units.

Overview of x264 The `x264` [31] is an open source library for encoding H.264 video streams. The `x264` uses raw input in YUV color space with 4:2:0 chroma subsampling. YUV color space encodes a color image or video taking human perception into account, and is typically used as intermediate representation between different stages of a video pipeline. The encoder supports constrained baseline and main profiles of H.264. Moreover, `x264` supports *matching blockings* of different size during motion estimation: 16×16, 16×8, 8×16, 8×8, 4×8, 8×4, and 4×4. The `x264` supports several optimized motion estimation algorithms, such as diamond (DIA), hexagon (HEX), uneven multi-hexagon (UMH), and exhaustive search [21]. The encoder performs mostly integer operations and has been optimized, using vector instructions, for various architectures, including PowerPC (altivec extensions) and x86 (streaming SIMD extensions, SSE).

1.2 The Cell processor

The Cell processor contains a general purpose PowerPC processing element (PPE) that implements the PowerPC ISA and eight special purpose synergistic processing elements (SPEs) implementing a vector ISA. Both the PPE and SPEs support SIMD extensions. Each SPE has a 128-bit datapath, 128 128-bit registers, and 256KB of software-managed local store. The SPE is a dual-issue, in-order, vector processor, designed to achieve high performance for straight-line vectorized code. The PPE and each SPE have a peak performance of 25.2 and 25.6 GFLOPS respectively for single-precision floating point operations, for a total of 230 GFLOPS.

The PPE has a conventional memory hierarchy with a two-level cache on–chip kept coherent with off–chip DRAM. SPEs transfer data from and to DRAM using DMAs. The DMA engine on each SPE is capable of contiguous transfers and scatter/gather operations and supports up to 16 outstanding transfers. The local stores of the SPEs are mapped to a global address space that includes off–chip memory; the PPE can access all local stores using regular load and store operations. The PPE and SPEs can also communicate with messages via small mailbox registers. Given the multiple available mechanisms for communication among SPEs and the PPE, software needs to assess the trade-offs between core-to-memory and core-to-core communication alternatives in order to use them effectively. All communication on the Cell processor flows through an on–chip element interconnect bus (EIB) that consists of four bi-directional rings. Finally, PPE has 1024-entry and each SPE has 256-entry translation lookaside buffer (TLB), however only PPE can resolve TLB misses.

1.3 Runtime system

In our design and implementation of *c264* we use Tagged Procedure Calls(TPC) [30], a task-based programming model and runtime that has been designed for the Cell processor. We use a runtime library to assist our porting of the application, for two reasons. First, we avoid the usage of IBM SDK [18] that usually requires higher programming effort than runtime environment or library. Second, we want high performance and efficient usage of available hardware characteristics.

The TPC runtime system creates a task using a function descriptor and arguments descriptors as input and identifies the task using a unique task ID. The programmer uses library calls to identify certain procedure calls as concurrent tasks and specify properties of the data accessed by them, to facilitate their transfers to and from local memories. Each argument descriptor is formed in a triplet containing the base address, the argument size, and the argument type that can be either IN, OUT or INOUT. he programming model allows both contiguous and fixed-stride arguments that we both use extensively in our implementation. The sizes of task arguments define the granularity of parallelism within a region of straight–line code or a subset of the iteration space of a loop. The programmer implements synchronization by using either point–to–point or collective wait primitives. Task arguments and their sizes are determined before task issue. Tasks

have no return values and all arguments are passed by referenc

Currently, we issue tasks only from the PPE and only SPEs execute tasks. The PPE prepares task descriptors and places them in SPE task queues for execution. Each SPE has a private task queue, an array of task descriptors. Furthermore runtime maintains a completion queue for each SPE. The PPE polls each completion queue for task status notifications from the SPEs. When the PPE issues a task, that task executes asynchronously to an SPE, while the PPE continues with program execution. When issuing an asynchronous task, the runtime returns a handle which can be used later to check the status of the specific *instance* of the issued task, while the issuing task continues with program execution. The issuer may wait for completion of issued tasks using point-to-point or collective synchronization primitives. When a task completes, SPE updates the completion queue of PPE, and releases the entry of the corresponding task in the completion queue. The runtime system uses only on–chip operations when initiating and completing tasks, although argument data may require off–chip transfers. Finally, runtime uses task prefetching and outstanding writebacks techniques in order to hide DMA latencies.

Chapter 2

Design and Implementation

This section presents in detail the design of *c264*. We first examine approaches to parallelizing the H.264 encoding algorithm and then discuss our design choices.

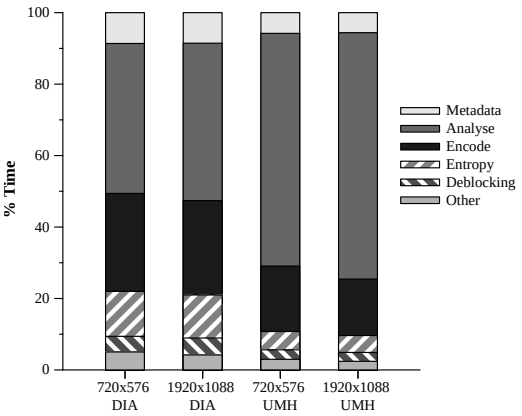

FIGURE 2.1: Normalized execution time breakdown of `x264` for different resolutions and motion estimation algorithms.

Figure 2.1 shows an execution profile for `x264` using manual code instrumentation, using one reference frame, 128×128 motion estimation search area and the UMH search algorithm. Profiling is done using the time base register of the Cell processor running at 79.8MHz, which provides adequate accuracy. Figure 2.1 shows that most of the execution time is spent in the analysis (temporal model) and encoding (spatial model) phases. We also observe that the input video resolution does not affect significantly the percentage of the execution time spent in analysis and encoding for the same motion estimation algorithm. When using macroblock-based parallelism, the potential parallel section of the code covers about 70% and 85% of the serial execution time on the PPE for DIA and UMH motion estimation algorithms respectively. Entropy encoding accounts for an additional 5–15% of the total execution time of the encoder. The deblocking filter accounts for 3–5% of total execution time and can be easily parallelize. The remaining execution time on the PPE is for task metadata handling, frame initialization, and memory copy operations between buffers.

2.1 Parallelization approaches

A coarse grain parallelization is to decompose the video sequence into groups of pictures (GOPS) [29], then every GOP is independently processed by a dedicated processor. However we opted not to use GOP parallelization because it has high memory consumption making it unsuitable for embedded processors with small on–chip memory. A typical group of pictures is around 100 frames. Encoding full high definition (1920x1088) video requires 6 MBytes for the input and reconstructed frame, plus 3 MBytes for each reference frame. Thus, the memory required for encoding a group of pictures is 600 MBytes for each task.

The `x264` uses coarse grain, frame-based parallelization, where different threads process different frames. When processing a frame, the master thread determines the frame type, calculates rate control, and then spawns a thread for this frame. Each worker thread proceeds to process all macroblocks through the phases, including entropy encoding. The worker thread synchronizes at the beginning of each line of macroblocks, when a frame requires referencing a not-yet complete part of another frame. In this case, the thread must wait for the completion of the appropriate macroblock line. Each thread updates a counter with the lines that the encoding process has completed. At the end of the each line of macroblocks, the thread also applies the deblocking filter. The master thread is responsible for final encapsulation in network abstraction layer (NAL) packets of the output streams of different threads. This coarse-grain approach is not appropriate for multi-core architectures with small, explicitly managed memories, due to memory requirements per thread (task). Encoding a full high definition frame requires 6 MBytes plus 3 MBytes for each reference frame. Thus, the memory required for encoding a frame is at least 9 MBytes, significantly higher than the per SPE local store on the Cell. Available parallelism is limited by the motion estimation window. High values in motion estimation range and advanced features, such as weighted prediction and adaptive B-frame decision, limits the available scalability. For example, Meenderinck et al. [11] using the default encoder parameters for the motion estimation range (16 pixels), they achieve 10.54X and 16.4X speedup when they use 16 threads and 32 threads respectively.

A more appropriate, fine grain approach for processors with small local stores is parallelization at the macroblock-level [6], where a single frame is being processed in parallel by multiple cores. Although this approach reduces memory requirements it increases communication significantly: the search region is required by all cores when processing a single macroblock, resulting in replication of data in multiple local stores over time. However, increased communication needs can be mitigated by the high on–chip and off–chip memory bandwidth available in modern multi-core processors.

A fourth parallelization alternative that lies between frame- and macroblock-based approaches is slice-based parallelization [28]. First, slice-based parallelization is less appropriate for the Cell processor due to the limited intra-frame parallelism and the increased memory requirements compared to a macroblock-based

approach. Second, having multiple independent slices increases the bitrate for a specific video quality, as a result of adding extra bits to the slice header and the reset of entropy contexts. Finally, slices can have arbitrary size from one macroblock up to a frame. Thus, in full high definition and using four slices, each task requires 1.5 MBytes of memory.

Parallelization at a granularity finer than the macroblock is feasible, for example by parallelizing motion estimation process for a single macroblock. However, this increases communication significantly; The search region is required by all cores when processing a single macroblock, resulting the need to transfer the same data to multiple local stores.

Macroblock-based parallelization requires addressing the following issues: Identifying dependencies, exploiting parallelism, offloading serial sections of code, task scheduling, overcoming memory limitations, and optimizing SPE code. Next, we discuss in more detail how we address each of these issues in our design.

2.2 Identifying dependencies

For a given macroblock, the following operations need data from neighboring macroblocks.

- Inter-frame prediction defines the predicted motion vector as the search center of motion estimation. Motion vectors of neighboring macroblocks determine the final motion vector.
- Intra-frame prediction that uses pixels from neighboring blocks of the current macroblock.
- The deblocking filter uses the pixel values of neighboring left and upper macroblocks.
- Entropy encoding selects one out of four look-up tables to use for encoding coefficients. The selection of table depends on the number of non-zero coefficients in previously coded upper and left neighboring macroblocks.

Macro-block level parallelization can theoretically increase the available degree of concurrency by exploiting parallelism across frames, using 3D-wavefront parallelization. Although, intra- and inter-frame macroblock-level parallelization are orthogonal approaches we must address some challenges to preserve data dependencies. First, there are control dependencies: allowing tasks across frames to execute concurrently would require postponing (moving) on the next frame the metadata handling of issued tasks from current frame. Second, runtime limitations, such as efficient multi-threading support, prevent the issue of tasks from different threads. Finally, issuing tasks from different frames to the same SPE would require using quantization matrices different frame types, which is prohibitive given the small local store memory size. The quantization matrices used in the encoder are typically around 84 KBytes. Thus, transferring entire matrices to local stores

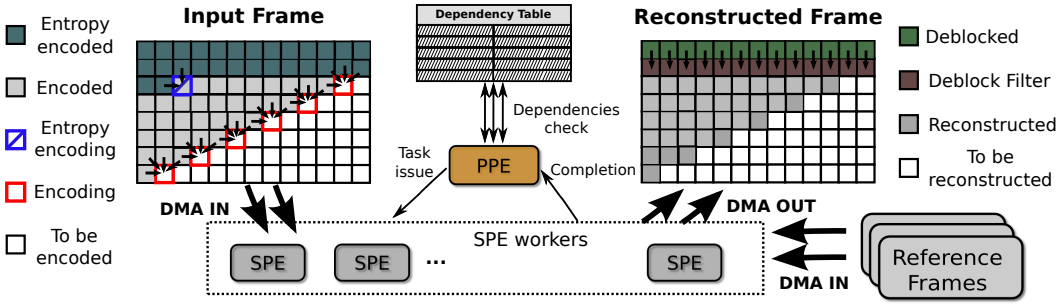

FIGURE 2.2: Design of *c264*, arrows inside frames express the dependencies among different tasks.

of each SPE is not possible. Instead, we transfer smaller portions of each matrix, each about 6 KBytes. For these reasons, we wait for all outstanding tasks to complete at the end of each frame before proceeding to the next frame and we leave for future work the inter-frame parallelism.

Figure 2.2 depicts the set of dependencies between macroblocks within a frame. Arrows show dependencies of outstanding macroblocks to previously encoded macroblocks on an antidiagonal manner.

2.3 Exploting parallelism

To take advantage of macroblock-level parallelism we create tasks that are related to macroblock processing for the three models (phases) of encoding: analyze and encode, entropy encoding, and deblocking. The rest of the application code is mostly serial code that cannot be parallelized. Some parts of this code can run in parallel with other tasks, whereas other parts are dependent on all tasks and need to run between groups of concurrent tasks. We offload the first type of code to an SPE as a single task that runs concurrently to other tasks, whereas the PPE executes the second type of code.

The simplest way to manage task dependencies is to issue all macroblocks in an antidiagonal-based manner and wait before issuing the next antidiagonal. This technique is also know as 2D-wavefront parallelism [15]. This leads to the maximum number of outstanding tasks, which is equal to the number of tasks on the antidiagonal. In this approach the number of independent macroblocks in each frame depends on frame resolution. Although motion estimation is independent, `x264` reads the results from neighboring macroblocks and uses them as "tips" for searching nearby to minimize the overall search time.

We create an other task for entropy encoding, the final step of the encoder before the encoder encapsulates the output stream in NAL packets. A main issue when creating tasks is to preserve control and data dependencies in macroblock processing within and across frames. The entropy encoding task is issued after the analyze and encode tasks, are completed, due to data dependencies. Entropy

encoding is *offloaded* as a single task to an SPE but can run concurrently to other tasks of the same frame. Entropy encoding is control-intensive and runs slightly faster on the PPE than the SPE, since the PPE is more control-efficient due to the use of dynamic branch prediction. Thus, the performance gain from offloading entropy encoding on SPEs arises from running the entropy encoding in parallel with other analyze/encode tasks, when dependencies are satisfied. However, for this reason, towards the end of each frame, when there are no additional available parallel tasks to execute, we execute the entropy encoding on the PPE. Moreover, when we issue a task for entropy encoding, the PPE can not issue a new task of the same type due to dependencies.

The third task for each macroblock performs deblocking. There are two variants of this task, one for the luminance component and one for the two chrominance components. Each task applies the deblocking filter in the macroblock line of the reconstructed frame. Deblocking tasks are issued when two dependencies are satisfied: first the completion of deblocking tasks on the previous line and second the analyze/encode tasks of all macroblocks in the current line.

Finally, we offload concurrent and independent `memcopy` operations to SPEs to allow for multiple outstanding transfers between application buffers and structures in off–chip memory. For example, we copy the input raw frame, from the read buffer to memory aligned buffers of the encoder.

2.4 Task scheduling

The first approach to enforcing dependencies among the analyze/encode tasks is to statically issue all tasks in an antidiagonal and then wait for all tasks to complete with a collective synchronization primitive. This *static scheduling* approach makes the assumption that each task has the same execution time. Unfortunately, heuristics in the motion estimation algorithms and non-predictable encoding time may introduce load imbalance and significant idle time.

An alternative approach to processing macroblocks, which we use in our implementation, is *dynamic scheduling*. We use per-macroblock state to maintain dependencies in the PPE. For this purpose, we issue tasks from next antidiagonals as soon as their individual dependencies are satisfied. We achieve this via a simple dependency table that represents all tasks within the frame.

2.5 Addressing memory limitations

The Cell processor requires attention to several aspects of memory management. Although data prefetching through multi-buffering is a common optimization for Cell code, the space available in the local stores for data prefetching in *c264* is limited. The SPE binary file of the application is about 156 KBytes, which does not leave space for storing both the working set of the active task and prefetched data. Also, the application has a large memory footprint due to the replication of data structures.

First, we reduce the memory latency of DMA transfers, using addresses aligned

at 128-bytes boundaries. Although DMA addresses in the SPE need to be 16-byte aligned, communication performance improves significantly when the data transfers are done from and to addresses aligned at cache line size boundaries. Moreover, we align strided arguments whenever possible to increase the efficiency of the DMA controller. In addition, we adjust the stride of non contiguous arguments to reduce the number of possible memory bank conflicts. For example, by using a slightly longer stride in the 1920×1080 video sequence we avoid 2-KByte strided access that introduce bank conflicts [23].

Second, *c264* shows high numbers of TLB misses, on SPE and PPE, due to wavefront parallelism. Wavefront parallelism requires irregular strided access patterns that harm TLB performance. To reduce the number of TLB misses we use large page sizes of 16 MBytes for the raw, uncompressed frame data via the `hugetlbfs` facility provided by the Linux kernel. We allocate the aforementioned large pages in the initialization phase of the encoder, thus removing the overhead of continuously allocating and freeing pages during the encoding phase.

Finally, replication of data structures required for parallelization (due to the multiple outstanding macroblocks in the analyze and encode process) increases significantly the memory footprint, causing TLB misses for high resolutions. To mitigate the impact of the increased memory requirements we use a custom, pre-allocated memory pool to recycle application data structures, reverting to the standard libc allocator only when the pool is empty.

2.6 SPE code vectorization and optimization

`x264` already provides vectorized versions of the kernels for the PPE using the Altivec extensions. In addition, we manually vectorize SPE code, eliminate branches and partially unroll loops in the encoder kernels. We manually enforce unrolling because the automatic loop unrolling done by the compiler produces code that does not fit in the local store. We also expand kernel variables to vectors, where possible, which works well with the wide SPE datapath. This reduces the overhead of loads and stores to local variables, because narrow loads require that data is rotated into the preferred vector element, and narrow stores require a read, a scalar insert, and a write operation.

2.7 Implementation of *c264*

Figure 2.3 shows the code size breakdown. The final SPE binary file is about 155 KBytes, including the DIA, UMH, hexagon (HEX), and exhaustive search motion estimation algorithms; the code segment is about 140 KBytes, the data segment is 1.2 KBytes, and the BSS segment is 12.8 KBytes. Macroblock analyze requires about 60 KBytes, macroblock encode 43 KBytes and entropy encoder 19 KBytes. Finally, runtime system requires about 12.7 KBytes, 5.1 KBytes for code and 7.6 KBytes for the BSS segment.

Limitations of *c264* . The main limitation of *c264* is that it does not allow the same flexibility in encoding parameters as `x264` due to the limited size of the local stores: *c264* supports only 16×16 block size for motion estimation and compensation and only inter-frame encoding in B-frames. The encoding of I-frames is done by the PPE to support smaller than 16×16 macroblocks, which is necessary for high quality. We modify the calculation of the limits for motion vector search to avoid producing motion vectors out of the reference region, which limits search window sizes to 128×128. We ported only the CAVLC encoder, although `x264` supports both the CABAC and the CAVLC algorithms. Finally, although the runtime system we use supports automatic prefetching of the arguments for enqueued tasks, this is effectively not used due to the limited amount of memory available in each SPE.

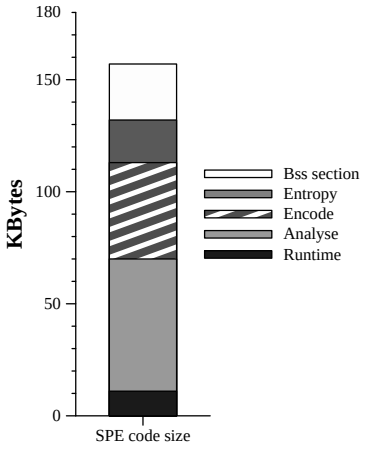

FIGURE 2.3: Code size breakdown for the SPE executable.

Chapter 3

Experimental Evaluation

This section presents experiments and results. We measure the impact of optimizations, speedup of our implementation, the available task parallelism, the impact of queue size, performance with other platforms, and the programming effort.

3.1 Experimental Platform and Methodology

We present results from experiments on a Playstation3 console with one 3.2 GHz Cell processor and 256-MBytes of main memory. In this platform, user programs have access to only six of the eight SPEs on the Cell processor. We use several video sequences from HD-VideoBench [7] that are listed in table 3.1.

In our evaluation we vary three H.264 parameters that affect application behavior significantly:

- *Resolution* affects the number of tasks. All input video sequences are available in two resolutions: 720×576 (small), and 1920×1080 (large). Each video stream consists of 100 frames at 25 fps.
- *Motion estimation algorithms* affect the computational workload of each task. We use two motion estimation algorithms one less and one more computationally intensive: diamond (DIA) and uneven multi-hexagon (UMH).
- *Motion estimation range* affects the size of data transfers for each task and the computational workload. We use two search ranges: 64×64 and 128×128 pixels.

For the rest of the encoding parameters, we keep the quantization constant at 26, we use four B-frames between I- and P-frames, we do not use B-frames as reference frames; all these are typical for the videos included in HD-VideoBench. We disable the adaptive selection of B-frames number, because the code responsible for the decision is computationally intensive and becomes the bottleneck. Finally, we use one reference frame for P-frame encoding and two for B-frame encoding due to local store size limitation.

Test Sequence	Resolution	No. frames	Comments
Blue_sky	720x576 1920x1088	100	Top of two trees against blue sky. High contrast, small color differences in the sky, many details and camera rotation.
Pedestrian	720x576 1920x1088	100	Shot of a pedestrian area. Low camera position, people pass by very close to the camera. Static camera.
Riverbed	720x576 1920x1088	100	Riverbed seen through the water. Very hard to code.
Rush_hour	720x576 1920x1088	100	Rush-hour in Munich city. Many cars moving slowly, high depth of focus. Fixed camera Shot.

TABLE 3.1: Input video sequences.

In addition, we vary the number of outstanding tasks per SPE. We always distribute tasks in a round-robin manner across SPEs, provided that there is a free slot available in the SPE task queue; otherwise the SPE is skipped.

In our analysis we present three metrics: speedups, concurrency characterization, and execution time breakdowns. We compute two types of speedups, using the execution time of the application: PPE-based only, where we compare *c264* execution time with the `x264` serial version running only on PPE and SPE-based, where we compare the execution time using *c264* with one SPE. To characterize the type of tasks and the resulting concurrency, we present a timeline with all executed tasks and their respective task execution times.

We show execution time breakdowns from both the PPE and SPEs. We break execution time on the PPE in four sections: (a) *PPE issue:* time spent in the runtime system for issuing tasks, (b) *Sync wait:* time waiting for specific task or tasks to complete, (c) *Queue stall:* time waiting for an empty queue slot in the task queues of SPEs, and (d) *Application:* time spent running application code. Similarly, SPE breakdowns consist of (a) *SPE Task:* compute time, (b) *SPE transfer:* runtime library and communication time, and (c) *SPE Idle:* idle time. Moreover in some PPE breakdowns we analyze further the application part to: (a) *Task management:* execution time of the main loop for dependency checks and issuing tasks in a wavefront manner. (b) Memory *allocator:* time to update offsets/pointers for the recycling of data structures. (c) *Metadata:* time for the management of motion vectors, number of coefficients, and calculating pointers to reference pictures. (d) *Entropy Parallel:* time spent executing tasks in PPE instead of SPE, if there are no empty slots in any SPE. Finally, (e) *entropy:* time spent at the end of each frame, when there are no additional available parallel tasks to execute and we execute the entropy encoding on the PPE.

In general, PPE application time is similar to the average SPE execution time, however, due to overlapping of the initiation of asynchronous tasks with task processing on SPEs, the match is not always exact. Also, as a reference point, we

Kernel	Speedup	Kernel	Speedup
sad_16x16	21.06	quant_4x4	2.15
sad_8x8	8.30	dequant_4x4	1.39
ssd_16x16	1.74	quant_4x4_dc	1.68
satd_4x4	1.67	dequant_4x4_dc	1.58
avg_16x16	22.59	zigzag_4x4	2.89

TABLE 3.2: Achieved speedup of different kernels.

include in our results application execution time on the PPE only. Note that we omit I/O time spent in reading the input from and writing the output to the disk. Finally, SPE code is transfered to each SPE once at application startup, as is typical for performance critical applications.

Next, we discuss four main issues: overall speedup and scalability, available task parallelism, and impact of task queue size and prefetching.

3.2 SPE code optimizations

To establish an appropriate baseline, we first discuss the impact of optimizations on the SPE application kernels. Table 3.2 shows the speedup achieved by each SPE optimization. The optimizations improve execution time between 1.39x and 22x, compared to the scalar version running on a single SPE. These results indicate the importance of kernel optimizations when running code on SPEs, before exploiting more parallelism. The SAD (sum of absolute differences) and AVG (average) kernels benefit most from these optimizations, since the SPE instruction set includes specific vector operations for both. Note that not all kernels, e.g. `dequant_4x4` (dequantization for 4×4 transform blocks), are fully vectorized, due to irregular data access pattern. However, starting with maximally optimized SPE code, leads to a more fair evaluation of *c264*. For this reason, all our results from this point on include these optimizations.

3.3 Overall speedup and scalability

Figure 3.1 shows results for different motion estimation algorithms and different search ranges, using the blue_sky stream and the large resolution. We note that using less computationally intensive algorithms, such as DIA, or a smaller search range (64 vs. 128) increases idle time on the SPEs by 52% and 33%, respectively on six SPEs. We also see that even with smaller search regions the computational requirements for UMH are considerably higher than DIA, leading to increased application execution time.

In Figure 3.2 we can see the normalized execution time breakdown for blue_sky using uneven multi-hexagon and 128×128 motion estimation range. For this run, we observe that sync wait time on the PPE is between 0.5% and 2% of PPE execution time. SPE idle time is between 11% and 28% of SPE execution time. SPE library time, that includes communication time is about 30%.

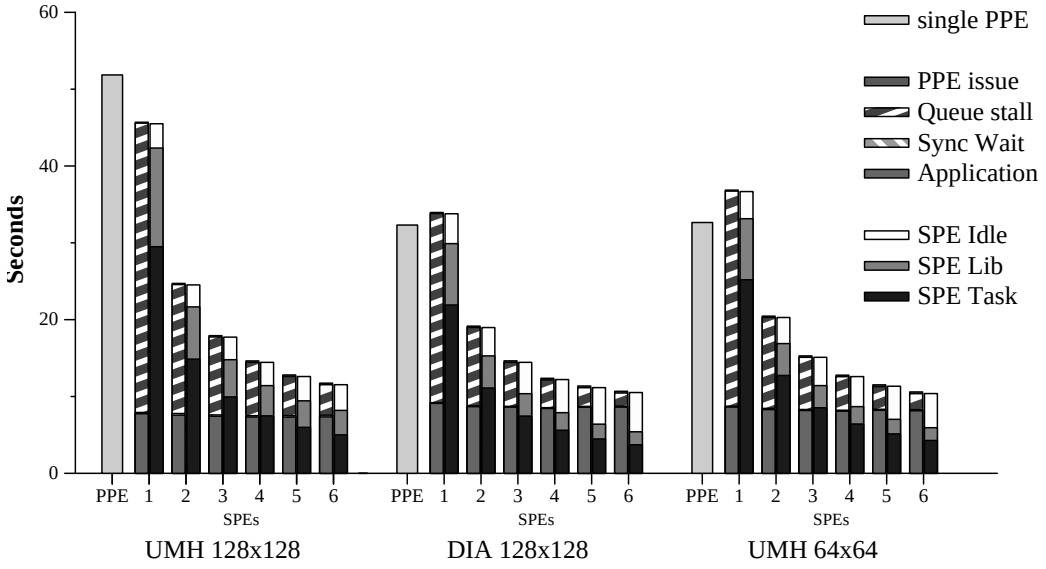

FIGURE 3.1: Impact of motion estimation algorithm and search range on blue sky input sequence with the large resolution.

Finally, application time on the PPE is between 16% and 63%, whereas task time on the SPEs is between 65% and 43% of the corresponding execution time. Less computational tasks, such as diamond motion estimation algorithm or smaller search window, increases idle time up to 50% of SPE total time when using six SPEs.

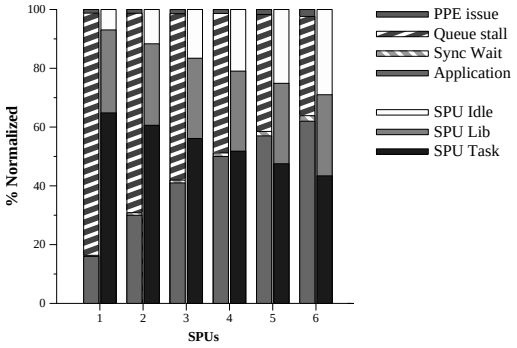

FIGURE 3.2: Normalized execution time breakdown for blue_sky using UMH and 128×128 range.

Figure 3.3(a) shows overall *c264* speedups as the execution time of the serial version running on the PPE only, divided by the execution time of the parallel version running on the PPE and a variable number of SPEs. The maximum speedup is between 3.9 and 7.1 on six SPEs and depends on the input stream *and* its resolution. We observe that the Riverbed video achieves super-linear speedup. The riverbed video sequence fails to perform well using the PPE only. In this input set, the temporal model fails to effectively reduce redundancy information, since there are limited similarities between neighbouring video frames. When inter-frame encoding cost is high, the encoder uses intra-frame only encoding. Intra-frame encoding is expensive due to irregular memory accesses in the current frame. Moreover, intra-frame encoding increases significantly the bitrate of the output video and

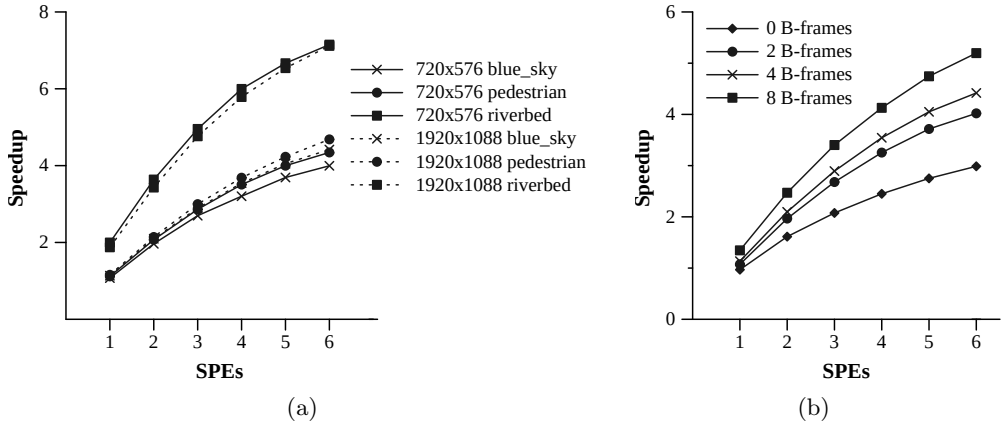

FIGURE 3.3: Speedup of *c264* calculated over PPE-only execution time (left) and for different number of B-frames for blue_sky with the large resolution (right).

creates extra computation overhead at the final phase of processing, the entropy encoder.

We measure scalability for high resolution video and UMH motion estimation. For out tests with four B-frames between I-and P-frames scalability is approximately the same, up to 4x for six SPEs. We note that scalability is independent of the input stream because the serial part of the application remains independent of the computation requirements of the task.

Figure 3.3(b) shows speedups using different number of B-frames between I- and P-frames, from 0 up to 8. Overall, the speedup increases when increasing the number of B-type frames. Analyse/encode tasks in a P-frame are less computationally expensive than the tasks of a B-frame. The two reference frames used in B-frames increase the execution time of motion estimation. In contrast, motion estimation in a P-frame is applied only to one frame, decreasing significantly the execution time of the serial version when only P-frames are used.

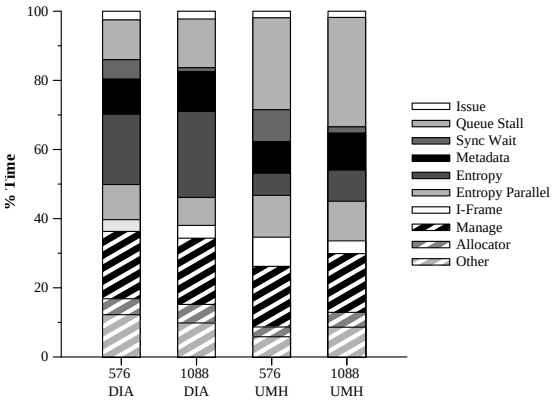

FIGURE 3.4: Breakdowns of *c264* PPE time using six SPEs and different motion estimation algorithms and resolutions.

Figure 3.4 presents PPE execution time breakdowns for different resolution and motion estimation algorithms using 6 SPEs. We see that *issue* time is 1% and 3% of the PPE execution time in DIA and UMH respectively and that different input resolution has a limited

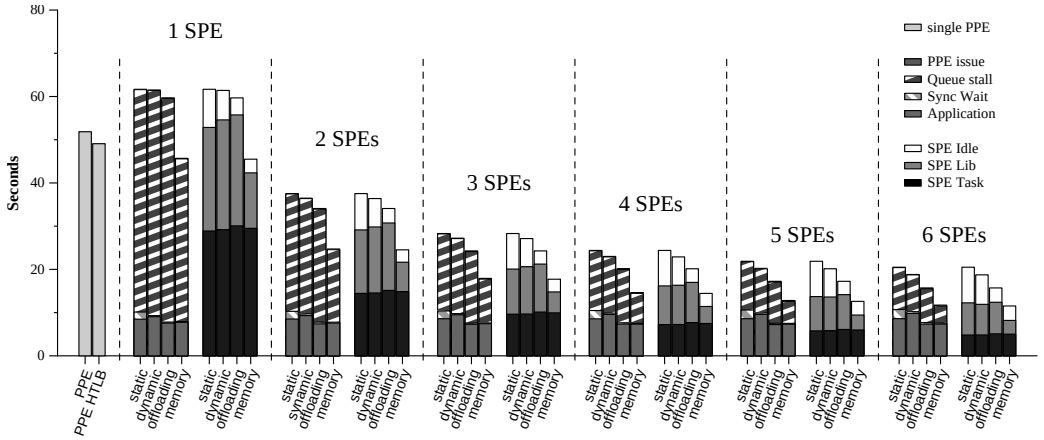

FIGURE 3.5: Impact of optimizations on *c264* execution time using the blue_sky input sequence, UMH motion estimation algorithm, and the large resolution.

impact. Task management costs take about 20% and 17% of PPE execution time using DIA and UMH for the large resolution. Memory allocation takes 3% and 4.5% of the PPE execution time when using small and large resolution respectively, for the UMH. In high resolutions the number of outstanding tasks and metadata increase, which increases overall memory utilization. The *metadata* section includes the management of motion vectors, number of coefficients, and calculating pointers to reference pictures. Finally, the metadata accounts for 9% to 12% of PPE total execution time. Although, the number of memory accesses in this module is small, the access pattern is irregular and causes many cache misses.

Overall, *c264* achieves significant speedups compared to the serial version running on the PPE. However, it still exhibits high task management overhead, that includes dependencies check, allocator and meta-data, from 30% up to 35%, and high communication overhead, up to 30% of the SPE exeuction time.

3.4 Impact of optimizations

In this section we evaluate the impact of each group of optimizations, using the baseline, selective, offloading, and memory versions, as described in our design.

Figure 3.5 shows the impact of each group of optimizations on execution time. We group optimizations in four different categories: (i) *Static* scheduling issues all tasks in an antidiagonal and then waits for all tasks to complete with a collective synchronization primitive; (ii) *Dynamic* scheduling includes the improved, selective dependence checking and issue; (iii) *Offloading* introduces offloading of serial tasks from the PPE in addition to dynamic scheduling on SPEs; (iv) *Memory* includes memory optimizations, dynamic scheduling and offloading. We use the blue_sky input with the high resolution, UMH motion estimation, and a 128×128 search range. We derive similar results with other video sequences and algorithms. Smaller resolutions usually decrease the benefits of memory optimizations.

Dynamic scheduling improves execution time over static scheduling on average by about 10% and reduces synchronization time 55%–80% on the PPE. Furthermore, idle time on the SPE decreases slightly compared to static scheduling. Dynamic scheduling issues tasks faster than the traditional 2D-wavefront algorithm and achieves better load balancing between SPEs. Overall execution time decreases by 9% using six SPEs compared with the static scheduling.

Offloading serial code from the PPE to SPEs increases available parallelism and reduces application and idle time on the PPE/SPE by 15% and 30% on six SPEs, respectively. However, the creation of more tasks slightly increases communication time on SPEs by about 5%.

Memory optimizations improve PPE execution time by up to 33% on six SPEs. We observe that memory optimizations reduce significantly communication time from 49% up to 60% on the SPE. Also stall time on the PPE decreases between 30% and 42%. Recycling application metadata and buffers does not have a significant impact on performance. However, it allows *c264* to run larger input resolutions. Huge pages have the biggest impact of all memory optimizations and reduce communication time by up to 50% compared to the offloading version. For example, using normal page sizes of 4 KB, SPE occurs about 9140 TLB misses using large resolution, 128×128 search range and blue_sky for input. In contrast using 16 MBytes pages the number of TLB misses decreases to 180. One side effect of the usage of large pages is that, for each TLB miss in SPE sends an interrupt to PPE for handling. Using larger TLB pages decreases the execution time of control code about 5% in large resolutions. Moreover, we observe the minor impact in the execution time, of the serial application running on PPE using large pages (PPE HTLB).

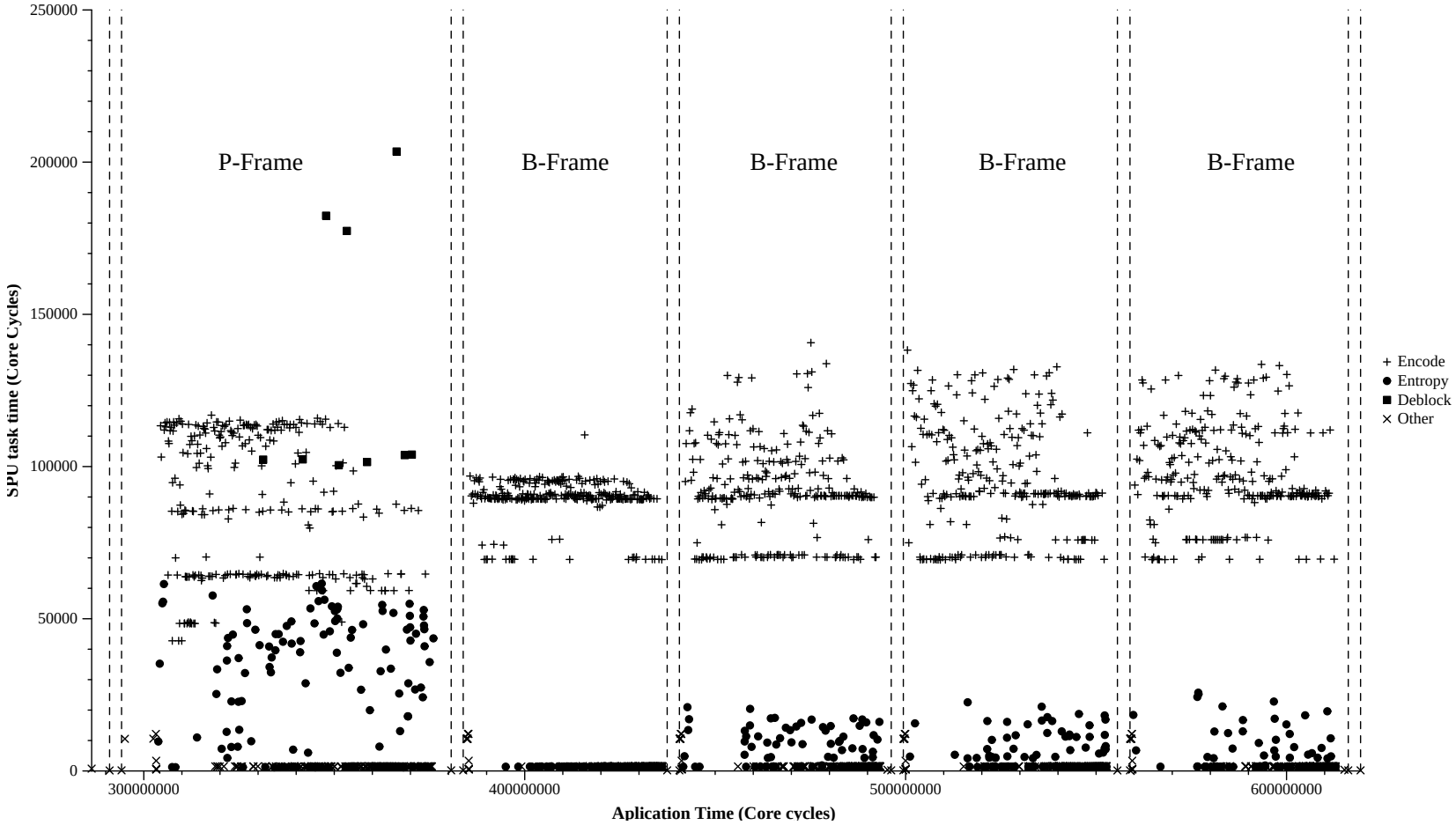

FIGURE 3.6: Timeline with all executed tasks and their respectively task execution time, collected from one SPE when all six SPEs are active.

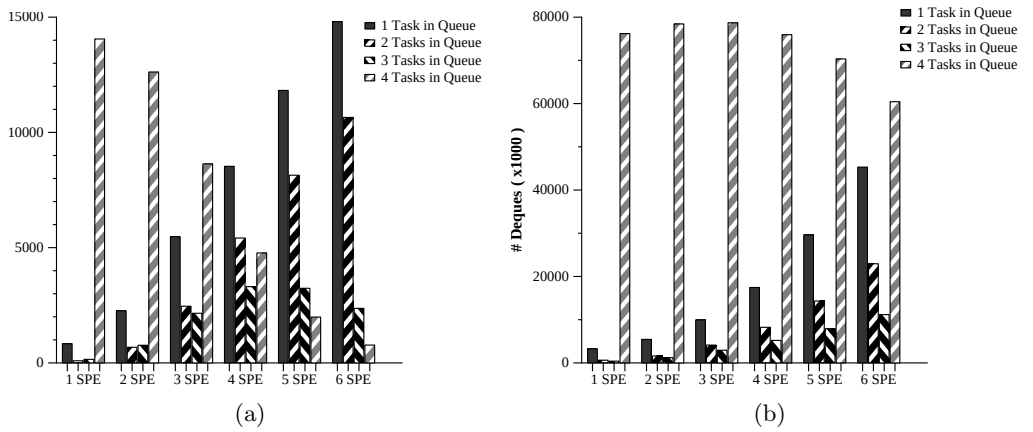

FIGURE 3.7: Average number of tasks in each SPE at task dequeue for blue_sky, small (left) and large (right) resolutions.

3.5 Available task parallelism

Next, we examine the number of tasks that are outstanding during different *c264* configurations. We are mainly interested in examining the impact of input resolution.

Figure 3.6 shows elapsed time in the x-axis and execution time of different task types in the y-axis. In this run we use the blue_sky video, small resolution, and 6 SPEs. Results are collected from only one SPE. File reading and writing is subtracted from the total time and is separated by dotted lines in the graph.

Between two frames the number of tasks executing on SPEs is limited. We issue some tasks, for example copying of frames in the encoder's buffers and initialization of different structures. About 5–10% of total execution time of PPE is spent in frame initialization and finalization. We see that the deblocking filter of the luminance component is the most computationally intensive task, taking up to 180000 core cycles. Next we see that analyze/encode tasks have significant variation in execution time. Furthermore, the application issues many small tasks, such as entropy encoding, that have execution time of about 1500 core cycles. These differences in execution time of tasks lead to load imbalance.

Figures 3.7(a) and 3.7(b) show the number of outstanding tasks in each SPE queue (average across all SPEs), when a new task is dequeued for execution. The SPE queue size has a maximum capacity of four tasks. At small resolutions and large SPE counts, there is typically one more task in the SPE queue when a task is dequeued for processing and rarely more than one. At larger resolutions, there is frequently three more tasks outstanding. At such resolutions, larger SPE queues would allow the PPE to create more tasks and could possibly reduce PPE task stall time as well.

3.6 Impact of task queue size

We observe that performance improves if the entropy encoding task is issued as soon as its dependencies are satisfied. The performance gain from entropy encoding task arises from running in parallel with other tasks. However, at the end of each frame, when there are no additional available parallel tasks to execute, encoder executes the entropy encoding on the PPE.

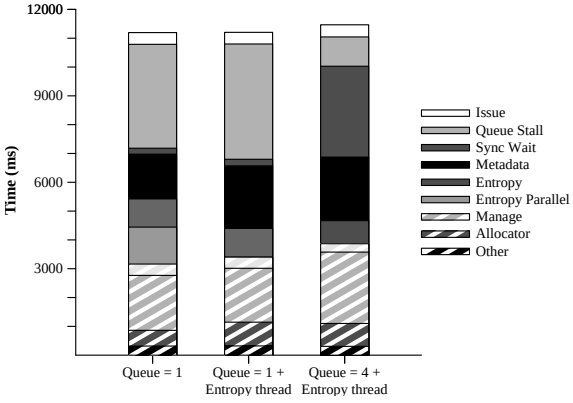

FIGURE 3.8: PPE execution time breakdown with one queue slot per SPE (left), with one slot and entropy encoding thread (middle) and with four slots (right).

An entropy task may be delayed behind other tasks in the SPE task queues, which are served FIFO. This is a problem of scheduling priority, however our runtime does not support different priorities in task execution. On the other hand, the PPE has a two-way SMT architecture. To compensate for priorities we spawn another thread for entropy encoding, to address the priority issue. The spawned thread is responsible only for entropy encoding. A macroblock is encoded as soon as the analyze/encode phases finish. For the synchronization we used a spinlock primitive.

Figure 3.8 shows the PPE execution time breakdown with six SPEs and varying queue sizes in three configurations: the optimized version with one queue slot on each SPE (left), using a PPE thread for entropy encoding (middle), and using a PPE thread for entropy encoding together with four task slots per SPE (right). In the right configuration, serial part of the entropy encoding is the time waitting in synchronization point the entropy thread to finish.

We observe that the allocator time increases due to locking, as a result of concurrent accesses to data structures. Metadata handling time increases because of memory accesses from the two PPE threads on the shared PPE L2 cache. Note that the issue time is lower due to overlapping and most queue stall time is converted in synchronization time. However, the total sum of synchronization, issue, and stall time is larger than when using one task slot per SPE queue. Finally, the serial part of entropy encoding is lower as expected, due to the additional thread that executes only this task. Overall, differences in task execution time can cause significant imbalance and decrease the performance of the encoder. We enforce load balancing, using one slot per SPE queue, although this increases issue time and stall time.

Resolution	*c264*	1 thread AMD SIMD	2 threds AMD SIMD
720x576	45.7	40.27	60.97
1920x1088	8.5	7.12	12.51

TABLE 3.3: *c264* and `x264` achieved fps using the blue_sky video sequence and the UMH algorithm with 128×128 search region.

3.7 Comparison with Other Platforms

To place our result in context, we also present `x264` results on two more traditional, x86-based multi-core platforms: A dual-processor system, with two 64-bit Dual-Core AMD Opteron Processor 2216 running at 2.4GHz, with 64KB L1-D cache, 64KB L1-I cache, and 1024KB L2 cache per core. The CMOS technology is at 90nm. We use six SPEs when we run *c264* on the Cell processor. Table 3.3 shows that the Cell outperforms the x86 processor by 14% and 20%, for small and large resolutions, respectively using one only thread. However, `x264` outperforms *c264* when two threads are active 33% and 45% for small and large resolutions respectively. We have similar results using different input video sequences.

3.8 Programmer effort

To estimate the programming effort associated with building *c264* we count the lines of code added or modified in *c264* compared to `x264`. Table 3.4 shows the programming effort measured in line counts and programming complexity. The programming complexity is related to the implementation time.

We category the modifications into:

- *SPE code for privatization of data structures* includes all code required in SPEs to access data structures in local memory
- *PPE task management* refers to dependencies management and the memory recycling mechanism.
- *PPE task metadata handling* refers to preparation of motion vectors and calculation of pointers to reference(s) frame(s).
- *Task PPE issue and SPE entry code* refers to preparing and issuing a task on the PPE and the SPE entry code that executes the appropriate task.
- *SPE kernel optimization* accounts for the code transformation of kernels, such as kernel vectorization and branch elimination.

The privatization of data structures was the most time consuming of all, because we must restrict access to global memory. The `x264` encoder, during to encoding process has memory read and updates global arrays that contain information about he current encoding frame. This forces us to modify almost all

Type	Lines of code	Programming effort
SPE privatization of data structures	4194	10–12 months
PPE task management	629	2–4 weeks
PPE task metadata handling	1299	4–6 days
PPE task issue and SPE entry code	1244	1–2 days
SPE kernel optimization	667	3–4 days
Total	8033	-

TABLE 3.4: Programming effort expressed in lines of code added to or modified from the original `x264` application.

source code that runs in SPEs. Task management required effort due to the implementation of dynamic scheduling using the data dependency table. Metadata handling, task issue, and entry code in SPE were trivial and minimal effort is needed. Kernel optimizations, such as vectorization or branch elimination, that are critical for exploiting the processing power of SPEs, has minor programming effort but higher performance gains. The programming effort for porting complex applications, such as `x264`, is significant despite the computational power of Cell processor.

Chapter 4

Related Work

In this chapter, we examine related work concerning parallelization and performance of H.264 video encoding, decoding on the Cell processor and other platforms. Related work falls into three broad areas: parallelization of H.264 video encoding and decoding on Cell, SPE kernel vectorization, parallelizaion in other architectures, and hardware implementations.

Park and Ha [22] analyze the expected performance of `x264` parallelization at the macroblock-level for the Cell processor. They offload only the first phase of the encoder, the analysis to SPEs, whereas the rest of the encoder remains on the PPE. Wu et Al. [14] present a real-time encoder of H.264, however, their effort is focused on kernel optimization rather than parallelization. They ignore the entropy encoding and other auxiliary parts of the encoder, such as frame initiation and rate control. In contrast, we present a full implementation for the Cell processor that parallelizes or offloads every phase of the encoder and show that the task management can occur significant overheads. Xun et al. [34] use a decentralized pipelined parallel encoding algorithm to achieve real-time encoding for high definition H.264 streams using eight SPEs. They achieve to decrease communication overhead by carefully optimize DMA transfers for each module of the encoder and decrease the task management overhead using decentralized task creation. For efficient communication they use multi-buffering and on–chip communication for data transfers between different modules of encoder. Their encoder uses no runtime and they implement it using directly the Cell SDK library.

The impact of SPE-vectorization in the computational intensive kernels have bee extensively analyzed. Vectorization of kernels used in video encoding include: motion estimation [35], deblocking filter [9], discrete cosine transformation [5], and interpolation [12]. Vectorization usually produces additional overhead in for packing and unpacking data into vectors, but lower overall execution time than the scalar version of kernels. In *c264* we use SPE-vectorization for macroblock analysis and encoding, but not for deblocking. We vectorize SPE kernels for macroblock analysis and encoding and achived speedups from 1.39 up to 22x. Our profiling shows that impact of vectorization of deblocking filter yield comparably small

performance gains, given the portion of execution time spent in this specific part of the code.

Previous research has also examined video decoding of H.264 on Cell processor. Macroblock-level parallelism can be exploited in intra- and inter-frame and is extensively analyzed by Meenderinck et al. [11]. They use an master-worker programming model similar to ours. Meenderick et al. also investigates different scheduling policies for macroblock-level parallelism, including a static scheduling approach to preserve locality [10]. One of the problems of 3D-wavefront parallelism, is the artificial delays that creates, that can be solve using different priority to tasks [8]. Baik et al. [17] parallelize an H.264 decoder for the Cell processor based on per-module profiling, yielding a speedup of 3.5x on four SPEs, compared to the PPE-only version. Baker et al. [2] present a scalable parallel H.264 decoder for the Cell processor. In contrast to us, they do not parallelize or offload the entropy encoder, although they do present the computational requirements for entropy CABAC decoding. Other work that has looked into parallel decoding for the Cell processor includes [37, 15]

H.264 video encoding parallelization was presented for other platforms as well. Zhao et al. [38] present simulation results for wavefront parallelization of H.264 video encoding, however they ignore possible overheads that may occur from parallelization, such as of communication, management and synchronization. The authors in [24] propose a hierarchical parallelization for H.264 video encoding for large scale clusters, using message passing and multi-threading programming models. They use coarse-grain parallelism, at the grain of group of frames, combined with slice parallelization. Although real-time operation can be achieved with such an approach, the latency is very high. Slice-based parallelization using balancing algorithms int the encoding to improve scalability of decoder has been also proposed [25]. Parallelization of video decoder using Intel hyper-threaded architectures has been proposed by Chen et al [36]. They propose two implementations, one using slice queues and one using task queues, and achieve speedup ranging from 3.74x to 4.53x. Streaming techniques have been also proposed [32] for real-time H.264 video encoding,

Finally, previous work has proposed specialized hardware for H.264 video encoding, while exploiting many levels of parallelism. The authors in [33] present a parallel, fine-grain implementation of the CAVLC entropy encoding for a 167-core (MIMD) processor. They partition entropy encoding into two phases, called scanning and encoding, and fine-grain parallelize the encoding phase. In *c264* we do not parallelize the encoding phase of entropy encoding, but rather offload it from the PPE to a single, dynamically chosen SPE. We find this to be an effective approach that reduces significantly the code complexity, while it results in good scalability. Authors in [27] present an architecture for de-blocking filter in H.264. They use several techniques to avoid redundant data transfers and reduce processing latency, such as classification of de-blocking filters and usage of bus interleaved architectures. The single chip encoder for H.264 presented in [3] uses a four-stage macroblock pipeline architecture that encodes 720p 30fps HDTV videos

in rea-time. The biggest drawback of the hardware approaches, is that they usually offer limited flexibility compared to the software approach *c264* employs. For instance, new motion estimation algorithms can not be easily introduced, because this would require architecture changes.

Chapter 5

Discussion and Conclusions

In this chapter we discuss some issues that programmers or programming model should deal with. We also present some future work and possible directions for continuing research. Finally we discuss our overall conclusions.

5.1 Discussion

Besides the issues we address in this work, we believe that programmers will need to deal with three additional, broad issues: Minimization of data transfers, scheduling of fine-grained tasks, local store memory size limitations, and code management.

Fine-grain task parallelism may increase the amount of data transferred between local store and global memory. Small tasks may require transferring multiple times the same data that could otherwise be transferred only once for each coarser-grained task. For example, in *c264* the data of search window for motion estimation, are overlapping between tasks of neighbouring macroblocks. Thus, there is a need for mechanisms that will minimize unnecessary data transfers transparently. Although, solution for software management cache in SPEs exists [20], requires major code modifications and have some computational overhead.

Fine-grained task-based scheduling can be used to avoid synchronization and task management overhead in the application side. With proper scheduling, task dependencies can be forced using without occurring additional overhead in the PPE side. Dynamic scheduling creates significant overhead in *c264*, caused mostly from the dependencies checking.

Local store memory size limitation has undesirable effects in many aspects on the *c264*. First, we have quality decrease of output stream, because the search window for motion estimation is reduced and the disabled advanced algorithms for analysis and encoding. Second, the performance of SPE code suffers, beacause we cannot enable advanced compiler optimizations and practically eliminates the prefetching of arguments. Finally, programming effort increase to address the challenges of limited memory.

Code management for future multicores with explicitly managed memories is an important issue. The large number of cores may require multiple copies of a

code region wasting memory resources. Storage management for code and data is undesirable for programmers and significantly increases complexity and effort. Although transparent solutions to these problems exits [13, 26, 4], they rely on knowledge of the programmer for efficient usage.

5.2 Future Work

In our work we examine many aspects the performance of the cell processor, however there are still areas for further research that include:

A detail analysis of trade-offs between the local-store and cache-based approached using complex and bandwidth demanding applications such as `x264`. Our preliminary results show that multi-core x86 processors can provide computational and communication resources to overcome the Cell processor. Moreover, an detailed comparison architectural comparisons across a broad range of multi-core systems and parallelization algorithms and parallelization approaches, is lacking. Finally, in our work we do not address the power considerations for varying technologies. A detailed analysis that compared the local-store and cache-based processors should be interesting.

Increase parallelism and scalability using inter-frame parallelism. This requires for the programmer to address the challenges, that we report in section 2.2. Moreover hierarchical parallelization, using more coarse grain methods such as group of pictures, can be used to exploit further more the data parallelism that exists in video encoding.

5.3 Conclusions

Our intention is to explore the available parallelism and possible overheads of a parallel H.264 video encoder on Cell processor. We start from an existing, thread-based parallel encoder, `x264`, that uses frame-based parallelism. We redesign the encoder to use master-worker, task-based parallelism with appropriate scheduling. We employ fine-grained macroblock-level parallelism and we deal with memory limitations in the Cell processor.

Our results show that our implementation achieves speedup of about 4.5x on six SPEs for many realistic scenarios compared to the PPE-only execution time. However, PPE and SPE still incur significant task management and communication overheads, up to 35% and 30% of PPE and SPE execution time, respectively.

We conclude that the task management overhead, communication and the size of local store memory is significant and affects the overall performance and scalability despite of optimizations. Overall, we believe that is possible to improve *c264* performance significantly, however it is still challenging to scale to large numbers of cores due to control flow complexity and task management overhead.

Bibliography

[1] ITU-T H.264: Advanced video coding for generic audiovisual services, November 2009.

[2] Michael A. Baker, Pravin Dalale, Karam S. Chatha, and Sarma B.K. Vrudhula . A scalable parallel H.264 decoder on the cell broadband engine architecture. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS '09)*, pages 353–362, New York, NY, USA, 2009.

[3] Yu-Wen Huang, Tung-Chien Chen, Chen-Han Tsai, Ching-Yeh Chen, To-Wei Chen, Chi-Shi Chen, Chun-Fu Shen, Shyh-Yih Ma, Tu-Chih Wang, Bing-Yu Hsieh, Hung-Chi Fang, Liang-Gee Chen . A 1.3 TOPS H.264/AVC single-chip encoder for HDTV applications. In *Proceedings IEEE International Solid-State Circuits Conference*, volume 1, pages 128–588, February 2005.

[4] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault and Y. Gao and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[5] A. Shahbahrami and B.H.H. Juurlink . Performance Improvement of Multimedia Kernels by Alleviating Overhead Instructions on SIMD Devices. In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies (APPT '09)*, pages 389–407, August 2009.

[6] S. M. Akramullah, I. Ahmad, and M. L. Liou. A data-parallel approach for real-time MPEG-2 video encoding. *Journal of Parallel and Distributed Computing*, 30(2):129–146, 1995.

[7] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. HD-VideoBench. A Benchmark for Evaluating High Definition Digital Video Applications. In *Proceedings of the 10th International Symposium on Workload Characterization (IISWC '07)*, pages 120–125, Washington, DC, USA, 2007.

[8] A. Azevedo, B. Juurlink, C. Meenderinck, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, and M. Valero. A Highly Scalable Parallel Implementation of H.264. In *The 4th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC'09)*, 2009.

[9] A. Azevedo, C. Meenderinck, B. H. H. Juurlink, M. Alvarez, and A. Ramírez. Analysis of video filtering on the cell processor. In *International Symposium on Circuits and Systems (ISCAS '08), 18-21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA*, pages 488–491, 2008.

[10] Chi Ching Chi, Ben Juurlink, Cor Meenderinck . Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proceedings International Conference on Supercomputing (ICS '10)*, June 2010.

[11] Cor Meenderinck and Arnaldo Azevedo and Ben H. H. Juurlink and Mauricio Alvarez, Alex Ramirez. Parallel Scalability of Video Decoders. *Signal Processing Systems*, 57(2):173–194, 2009.

[12] Cor Meenderinck and Ben Juurlink. Intra-Vector SIMD Instructions for Core Specialization. In *Proceedings of the IEEE International Conference on Computer Design (ICCD '09)*, October 2009.

[13] I. Corp. An introduction to compiling for the Cell Broadband Engine architecture, Part 5: Managing memory, 2006. http://www.ibm.com/developerworks/edu/pa-dw-pa-cbecompile5-i.html.

[14] Di Wu, Boonshyang Lim, Johan Eilert and Dake Liu. Parallelization of High-Performance Video Encoding on a Single-Chip Multiprocessor. In *IEEE International Conference on Signal Processing and Communications*, 2008.

[15] Erik B. Van Der Tol, Egbert G. T. Jaspers, Rob H. Gelderblom E.B. van der Tol, E.G.T. Jaspers and R.H. Gelderblom. Mapping of H.264 decoding on a multiprocessor architecture. In *Image and Video Communications and Processing*, pages 707–718, 2003.

[16] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *11th International Conference on High-Performance Computer Architecture (HPCA '05), 12-16 February 2005, San Francisco, CA, USA*, pages 258–262, 2005.

[17] Hyunki Baik, Kue-Hwan Sihn, Yun-il Kim, Sehyun Bae, Najeong Han, and Hyo Jung Song. Analysis and Parallelization of H.264 decoder on Cell Broadband Engine Architecture. In *IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 791–795, Washington, DC, USA, 2007. IEEE Computer Society.

[18] IBM. Cell Broadband Engine resource center, 2008. http://www.ibm.com/developerworks/power/cell/downloads.html.

[19] Intel Corporation. Intel Single-Chip Cloud Computer. http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC-Overview.pdf.

[20] Lee, Jaejin and Seo, Sangmin and Kim, Chihun and Kim, Junghyun and Chun, Posung and Sura, Zehra and Kim, Jungwon and Han, SangYong. COMIC: a coherent shared memory interface for cell be. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*, pages 303–314, New York, NY, USA, 2008. ACM.

[21] L. Merritt and R. Vanam. Improved Rate Control and Motion Estimation for H.264 Encoder. In *Proceedings of the International Conference on Image Processing (ICIP 2007), September 16-19, 2007, San Antonio, Texas, USA*, pages 309–312, 2007.

[22] J. Park and S. Ha. Performance Analysis of Parallel Execution of H.264 Encoder on the Cell Processor. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '07)*, pages 27–32, 2007.

[23] Redbooks, IBM. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*, chapter 4, page 325. 2008.

[24] A. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical Parallelization of an H.264/AVC Video Encoder. In *Proceedings of the international symposium on Parallel Computing in Electrical Engineering (PARELEC '06)*, pages 363–368, Washington, DC, USA, 2006.

[25] M. Roitzsch. Slice-balancing H.264 video encoding for improved scalability of multicore decoding. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT '07)*, pages 269–278, New York, NY, USA, 2007.

[26] Ron Cytron and Paul G. Loewner. An automatic overlay generator. *IBM Journal of Research and Development*, 30(6):603–608, 1986.

[27] Shih-chien Chang, Wen-hsiao Peng, Shih-hao Wang, Tihao Chiang. A Platform Based Bus-interleaved Architecture for De-blocking Filter. In *Proceedings IEEE International Conference on Consumer Electronics (ICCE '05)*, volume 51, pages 249–255, 2005.

[28] S.M. Akramullah, I. Ahmad, M.L. Liou. Performance of a software-based MPEG-2 video encoder on parallel and distributed systems. In *IEEE Transactions on Circuits and Systems for Video Technology (TCSV '97)*, volume 7, pages 687–695, 1997.

[29] T.Olivares, F.Quiles, P.Cuenca, L.Orozco-Barbosa and I.Ahmad. Study of data distribution techniques for the implementation of an MPEG-2 video

encoder. In *Proceedings IEEE International Conference on Parallel and Distributed Computing and System (PDCS '99)*, pages 537–542, November 1999.

[30] G. Tzenakis, K. Kapelonis, M. Alvanos, K. Koukos, D. S. Nikolopoulos, and A. Bilas. Tagged Procedure Calls (TPC): Efficient runtime support for task-based parallelism on the Cell Processor. In *The 2010 International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC '10)*, Jan. 2010.

[31] Videolan. x264: A free H.264/AVC encoder, http://www.videolan.org/developers/x264.html.

[32] N. Wu, M. Wen, W. Wu, J. Ren, H. Su, C. Xun, and C. Zhang. Streaming HD H.264 encoder on programmable processors. In *Proceedings of the 17th ACM international conference on Multimedia (MM '09)*, pages 371–380, New York, NY, USA, 2009.

[33] Z. Xiao and B. M. Baas. A High-Performance Parallel CAVLC Encoder on a Fine-Grained Many-core System. In *International Conference on Computer Design (ICCD '08)*, pages 248–254, Oct. 2008.

[34] Xun He, Xiangzhong Fang, Ci Wang, and Satoshi Goto. Parallel HD encoding on cell. In *International Symposium on Circuits and Systems (ISCAS '09)*, pages 1065–1068, 2009.

[35] Xun He, Yunfei Zhang, Xuewen He, Haicheng Wu, Yao Zou, Shanghai Jiao Tong. An efficient block motion estimation method on CELL BE. In *International Conference on Audio, Language and Image Processing (ICALIP '08), July 7-9, Shanghai, China*, pages 1672–1676, 2008.

[36] Yen-Kuang Chen and Xinmin Tian and Steven Ge and Milind Girkar. Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures. In *18th International Parallel and Distributed Processing Symposium (IPDPS '04), 26-30 April, Santa Fe, New Mexico, USA*, page 63, 2004.

[37] Yuan, Yu and Yan, Rong and Li, Huoding and Liu, Xing and Xu, Sheng. High definition H.264 decoding on cell broadband engine. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 459–460, New York, NY, USA, 2007.

[38] Zhuo Zhao and Ping Liang. A Highly Efficient Parallel Algorithm for H.264 Video Encoder. In *International Conference on Acoustics, Speech and Signal Processing, 2006*, volume 5 of *Acoustics, Speech and Signal Processing*, pages 489–492, May 2006.