Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine

Chi Ching Chi
Technische Universität Berlin
Franklinstrasse 28/29
10587 Berlin, Germany
cchi@cs.tu-berlin.de

Ben Juurlink
Technische Universität Berlin
Franklinstrasse 28/29
10587 Berlin, Germany
b.juurlink@tu-berlin.de

Cor Meenderinck
Delft University of Technology
Mekelweg 4
Delft, Netherlands
cor@ce.et.tudelft.nl

ABSTRACT

How to develop efficient and scalable parallel applications is the key challenge for emerging many-core architectures. We investigate this question by implementing and comparing two parallel H.264 decoders on the Cell architecture. It is expected that future many-cores will use a Cell-like local store memory hierarchy, rather than a non-scalable shared memory. The two implemented parallel algorithms, the Task Pool (TP) and the novel Ring-Line (RL) approach, both exploit macroblock-level parallelism. The TP implementation follows the master-slave paradigm and is very dynamic so that in theory perfect load balancing can be achieved. The RL approach is distributed and more predictable in the sense that the mapping of macroblocks to processing elements is fixed. This allows to better exploit data locality, to overlap communication with computation, and to reduce communication and synchronization overhead. While TP is more scalable in theory, the actual scalability favors RL. Using 16 SPEs, RL obtains a scalability of 12x, while TP achieves only 10.3x. More importantly, the absolute performance of RL is much higher. Using 16 SPEs, RL achieves a throughput of 139.6 frames per second (fps) while TP achieves only 76.6 fps. A large part of the additional performance advantage is due to hiding the memory latency. From the results we conclude that in order to fully leverage the performance of future many-cores, a centralized master should be avoided and the mapping of tasks to cores should be predictable in order to be able to hide the memory latency.

Categories and Subject Descriptors

I.4 [**Image Processing and Computer Vision**]: Compression (Coding); D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Algorithms, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.
Copyright 2010 ACM 978-1-4503-0018-6/10/06 ...$10.00.

Keywords

H.264, video, decoding, Cell, parallel, programming

1. INTRODUCTION

In the past performance improvements were mainly due to higher clock frequencies and due to exploiting Instruction-Level Parallelism (ILP). ILP improvements, however, have reached their limit and show diminishing returns in terms of area and power. Power limitations also prevent further frequency scaling. As a result, industry has made a paradigm shift towards multi-cores.

With the recent move to homogeneous multi-cores we have witnessed a doubling and quadrupling of processor cores. This approach, however, is not scalable to the many-core era. Slow inter-core communication, quadratic complexity of cache coherency, and shared memory bandwidth limitations [20] will soon create bottlenecks. The IBM Cell processor is a heterogeneous multi-core, which comes a long way in addressing these issues. The price is paid in programmability, however, as tasks previously handled by hardware, such as inter-core communication, are moved to software. Nevertheless, it is expected that future processor architectures will integrate Cell-like features because it is more scalable than homogeneous multi-cores. Investigating the programmability of the Cell processor is therefore important to gain insight in defining future programming models.

The goal of this paper is to investigate how to leverage the full performance potential of future many-core architectures. In order to do so we have implemented two parallel versions of an H.264 decoder, both exploiting macroblock-level parallelism, on a 16-SPE Cell Blade platform. The H.264 coder/decoder (codec) [1] is presently the most widespread and advanced video codec [13]. The first implementation, referred to as the Task Pool (TP), has been presented previously [2] and is based on the master-slave programming paradigm. Slaves request work (in this case macroblocks, MBs) from the master, which keeps track of the dependencies between the MBs. The second implementation is a novel approach referred to as Ring-Line (RL). In RL the cores process entire lines of MBs rather than single MBs, enabling distributed control. Furthermore, its static and predictable mapping of MBs to cores allows to overlap communication with computation and reduces the memory bandwidth requirements.

We analyze the TP and RL approaches based on the theoretical scalability as well as experimental results. The theoretical scalability is higher for the TP implementation. However, both the actual scalability and the absolute performance of the actual implementations favor the RL ap-

proach. Most of the performance advantage is due to hiding the memory latency.

Independently, Baker et al. [5] have implemented an approach similar to Ring-Line on the Cell platform. It is similar in the sense that SPEs process entire lines and a distributed control scheme is used. However, they do not fully exploit one of the key features of the Cell processor, namely explicit data management. In contrast to our implementation, they do not apply pre-fetching nor do they use the possibility to send data from one SPE to another directly. In our implementation inter-SPE communication is used to reduce off-chip bandwidth utilization, while pre-fetching is used to hide the memory access latency. Compared to Baker's implementation, our RL implementation is about 50% to 100% faster, with the caveat that this is derived from comparing the results at 6 SPEs, since Baker et al. only provided results for up to 6 SPEs due to limitations of their test platform (PS3).

This paper is organized as follows. In Section 2 a brief overview of H.264 and the Cell architecture is provided. In Section 3 the experimental setup is presented. In Section 4, the TP and RL implementations are discussed, while they are evaluated in Section 5. Section 6 concludes this paper.

2. BACKGROUND

2.1 Cell Architecture Overview

The Cell Broadband Engine [14] is a heterogeneous multicore consisting of one PowerPC Element (PPE) and eight Synergistic Processing Elements (SPEs). The PPE is a dual-threaded general purpose PowerPC core with a 512kB L2 cache. Its envisioned purpose is to act as the control/OS processor, while the eight SPEs provide the computational power. Figure 1 shows a schematic overview of the Cell processor. The processing elements, memory controller, and external bus are connected to an Element Interconnect Bus (EIB). The EIB is a bi-directional ring interconnect with a peak bandwidth of 204.8 GB/s [6]. The XDR memory can deliver a sustained bandwidth of 25.6 GB/s.

What makes the Cell such an innovative design is not its heterogeneity, but its scalable memory hierarchy. In conventional homogeneous multi-core processors, each core has several layers of cache. The caches improve performance, because they reduces the average latency and bandwidth usage of the external (off-chip) memory. With multiple cores the caches need to be kept coherent. The cache coherency actions grow with a complexity of $O(n^2)$, where n is the number of cores, which quickly become unpractical in many-core architectures. In the Cell architecture, the SPEs do not feature a cache and rely on a local store and DMA unit instead to access the memory. Each SPE has a local store of size 256 kB. The SPEs can only work on data in the local store. The programmer is responsible for the data transfers using explicit DMA operations. The programming style is that of the shopping list model. Instead of loading every data item separately at the time it is needed (as is the case with cache based systems), all data required for a task is brought in at once and before the execution of the task. Moreover, loading the data of one task should be done concurrently with the execution of another task in order to fully hide the memory latency.

2.2 H.264 Decoding

Currently H.264 [1, 18] is the best video coding standard in terms of compression rate and quality [13]. Also, it is the most widespread standard for digital video. It is used in Bluray, digital television broadcast, online digital content distribution, mobile video players, etc. The compression rate is over two times higher compared to previous standards, such as MPEG-4 ASP, H.262/MPEG-2, etc. H.264 uses the YCbCr color space with mainly a 4:2:0 subsampling scheme. In this subsampling scheme the luma component (Y) has the same resolution as the frame, while the chroma components (Cb and Cr) are at a quarter resolution. Throughout the paper this subsampling scheme is assumed.

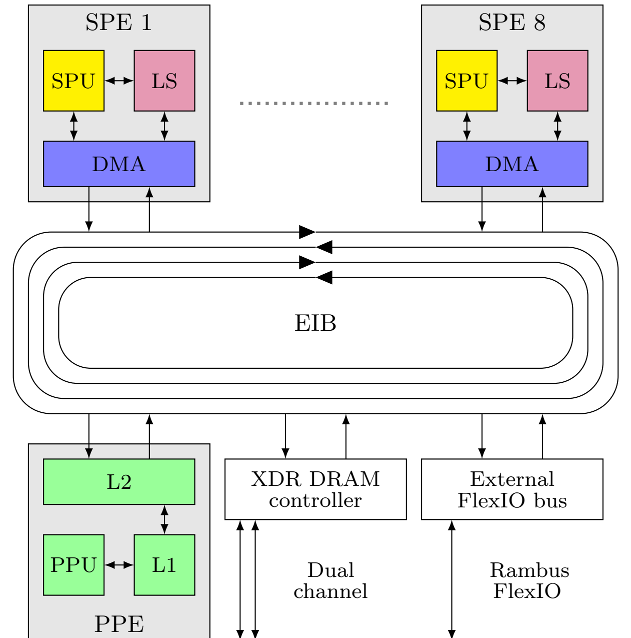

Figure 1: Schematic view of the Cell Broadband Engine architecture.

This paper focuses on H.264 decoding, of which the block diagram is depicted in Figure 2. In the entropy decoding the data of the MBs is extracted from the H.264 stream. The remaining kernels use the extracted data to decode the MB. The entropy decoding, motion compensation, and deblocking filter kernels require the most computation time with on average around 30%, 35%, and 25%, respectively. The intra prediction, inverse quantization, and inverse discrete cosine transform kernels account for the remaining 10%.

The entropy decoding can be parallelized on the frame/slice level using the frame markers. Parallelization of the MB kernels can be done on several levels. The next section briefly reviews existing parallelization strategies.

2.3 Parallelization Opportunities

A lot of work has been done to parallelize H.264 in general. However, most works exploit only coarse-grain parallelism at the Group of Pictures (GOP)-, frame-, and slice-level or apply function-level decomposition. Using the latter, Gulati et al. [9] described a system for encoding and decoding H.264 videos. Data-level decomposition was applied by, among others, Rodriguez et al. [15], who proposed an encoder that combines GOP- and slice-level parallelism. Chen et al. [7] proposed a combination of frame- and slice-level parallelism. Roitzsch [16] proposed a scheme based on slice-

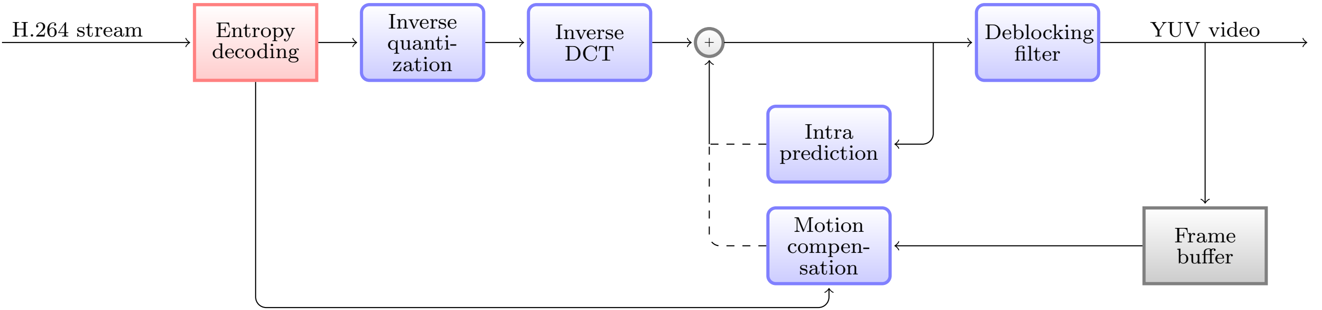

Figure 2: Block diagram view of the H.264 decoder.

level parallelism by modifying the encoder. Baik et al. [4] have implemented a parallel version of H.264 on the Cell processor. The design utilizes both data- and function-level decomposition by partitioning MBs from inter coded frames among the available SPEs in a load balanced fashion, and dedicating an additional SPE to deblocking. None of these parallelization strategies, however, are sufficiently scalable to efficiently utilize emerging many-cores. MB-level parallelization has proven to be much more scalable and is, therefore, subject of this paper. The remainder of this section describes the different strategies to exploit MB-level parallelism.

MB-level parallelism can be exploited in the spatial (within a frame) and the temporal domain (among frames). Spatial MB-level parallelism has been introduced by Van der Tol et al. [17]. Chen et al. [21] evaluated this approach on a Pentium machine with SMT and multi-core capabilities. These works also suggest the combination of MB-level parallelism in the spatial and temporal domains. This is explored further in the work of Meenderinck et al. [12] and was renamed to 3D-Wave parallelism. They showed that the amount of available parallelism is very large. They also renamed spatial MB-level to 2D-Wave parallelism. This naming scheme is also used within this paper.

2.3.1 2D-Wave Parallelism

The 2D-Wave parallelization exploits MB-level parallelism within a frame. The amount of parallelism is limited by the data dependencies in the spatial domain, referred to as intra dependencies. Figure 3 illustrates all dependencies in the spatial domain. To decode the current MB, data from four surrounding MBs is needed. Therefore, these must be fully decoded before the current MB can be processed.

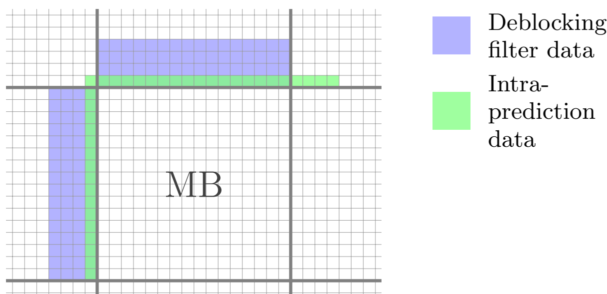

Figure 3: MB dependencies in the spatial domain.

As a result of these dependencies, the MBs must be decoded in a certain order. As shown in Figure 4, MBs on a

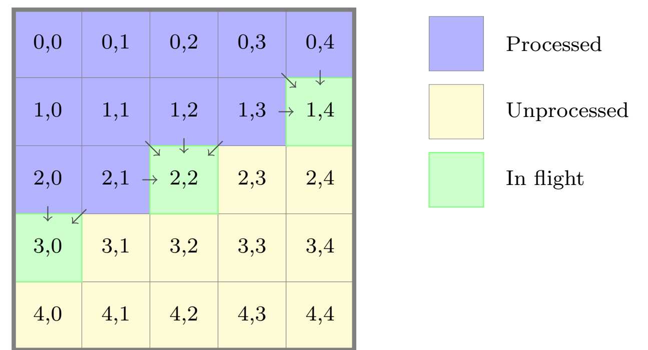

Figure 4: 2D-Wave parallelization: MBs on a diagonal are independent and can be decoded concurrently. The arrows represent the dependencies.

diagonal line are independent of each other and can therefore be processed parallel. The figure also shows that the number of parallel MBs is limited by the horizontal resolution. Only one available MB in an entire line exists at any time. Therefore, the maximum spatial parallelism is given by:

$$ParMB_{max,2D} = min(\lceil N_{MB,hor}/2 \rceil, N_{MB,ver}), \quad (1)$$

where $N_{MB,hor}$ and $N_{MB,ver}$ are the number of horizontal and vertical MBs in the frame. The maximum 2D-Wave parallelism for FHD is $min(\lceil 120/2 \rceil, 68) = 60$. The equation shows that the parallelism increases with the frame size. In this paper the term parallelism is used interchangeable with the number of concurrent MBs when discussing wave parallelization.

In the 2D-Wave the amount of available parallelism is not constant during the decoding of a frame as it suffers from ramping and dependency stalls. Ramping stalls occur at the start and the end of the frame when the number of available MBs is lower than the number of Processing Elements (PEs). The dependency stalls are due to variable MB decoding times. On the SPE the MB decoding time varies from about 5μs to 40μs. Figure 5 illustrates both effects.

2.3.2 3D-Wave Parallelism

MB-level parallelism among frames is referred to as temporal MB-level parallelism. Combining the spatial and temporal parallelism results in the 3D-Wave parallelization. The amount of parallelism it provides is very large and increases proportionally with the number of frames in flight. Meenderinck et al. [12] showed that the maximum available par-

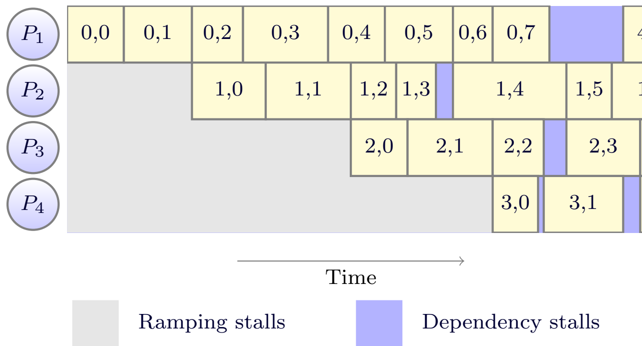

Figure 5: Ramping and dependency stalls reduce the scalability and performance of the 2D-Wave parallelization.

allelism for FHD sequences is between 4000 and 7000.

Although the 3D-Wave approach improves the scalability it is not applied in this work. The 3D-Wave approach incurs additional overhead which only pays off at very large scale systems. Our platform has 16 cores, in which case the 2D-Wave approach provides sufficient parallelism.

3. EXPERIMENTAL SETUP

Evaluation of the parallel implementations was done using a Cell Blade located at the Barcelona Supercomputing Center (BSC). The Cell Blade consists of two physical Cell processors linked via the FLEXIO bus, which has a peak throughput of 37.6 GB/s. The second Cell processor shares the memory controller of the first via a FLEXIO bus. The amount of XDR external memory is 1 GB with a rated bandwidth of 25.6 GB/s. The operating system is Fedora Core 7 with kernel version 2.6.22.

The FFmpeg [8] audio/video decoder is used as a base for our parallel implementations. The FFmpeg build is configured to use all optimizations, including the AltiVec PowerPc extensions. The SPE compiler optimization setting is `-O2`. The PPE and SPE hardware counters, which have a resolution of 14.8 MHz and negligible call time, are used to measure execution times.

The experiments are performed with the Full High Definition (FHD) sequences of the HDVideoBench [3]. These sequences are encoded with X264 [19] conform the High profile level 4.0 H.264 standard using CABAC as the entropy decoding scheme. More specifically, the streams are encoded using two B-frames between I- and P-frames with weighted prediction. The motion vector range is set to 24 pixels.

4. IMPLEMENTATION ON CELL

FFmpeg, which is used as a base for both the Task Pool and Ring-Line implementation, provides an highly optimized H.264 decoder in their libavcodec library. As with the other codecs in libavcodec, the H.264 decoder is instructed to decode on a per frame basis using `decode_frame`.

To implement the 2D-Wave parallel approaches on the Cell processor, the SPEs have to apply the MB kernels. The task of the PPE is to perform the entropy decoding and act as the controller. In the original `decode_frame` the MBs are processed in scan line order. This includes the entropy decoding, which extracts the parameters for one MB at a time. Therefore, for both implementations the entropy decoding has to be decoupled. Also the MB kernels need to be ported to SPE code. The motion compensation and IDCT kernel are ported to use the SPE SIMD engine using the FFmpeg Altivec code as a base. The deblocking filter and intra-prediction use scalar code. The difference between the TP and RL is mainly in the data flow.

4.1 Decouple Entropy Decoding

In this work, as in [10, 12, 17], only the MB processing is parallelized. We assume that either an entropy decoding accelerator exists or that the entropy decoding has been parallelized at the slice level.

In our implementations, the entropy decoding is decoupled from the MB processing. The output of the entropy decoding is stored in a work unit matrix of `H264mb` structures for each MB. A single `H264mb` has a size of 1.9 kB. Also a `H264slice` structure is used for all the slice information of 12kB. The `H264mb` and `H264slice` are both the minimal required subsets of the `H264Context` originally used by libavcodec, which has a size of 170 kB. To prevent the entropy decoding from influencing the results, it is performed entirely before starting the parallel MB processing in both implementations.

4.2 Task Pool Implementation

In the Task Pool approach a centralized master is used to distribute the MBs dynamically over the processing elements (PEs). A high level view of the implementations is shown in Figure 6. In the figure the master task is handled by the PPE and the SPEs are the slaves. The master keeps track of the state of the MBs using a dependency table and serializes the work available in a task queue.

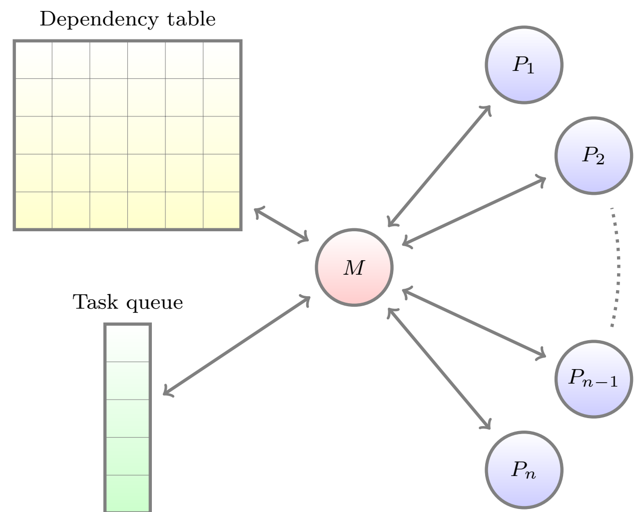

Figure 6: Centralized Task Pool implementation of 2D-Wave.

The dependency table contains a dependency count for each MB in the frame. Whenever an SPE has processed a MB, the dependency counts corresponding to the right and down-left MB are decremented. The MB state is free when its dependency count drops to zero. Free MBs are scheduled in a first-in-first-out manner using the task queue. Figure 7 shows the dependency flow and initial values of the dependency table. The latter equates to the number of arrows pointing towards each MB.

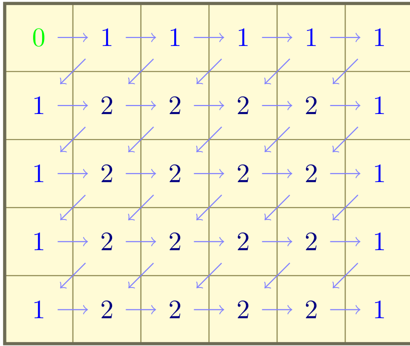

Figure 7: Dependency flow and inital values in the dependency table. After a MB is processed, the master decrements the dependency count of the MBs that are pointed to by the arrow(s).

The dependency table and task queue can only be accessed by the PPE, thus providing synchronized access. Experiments have shown [11] that the direct mapped mailboxes, which can send a 32-bit message, provide the fastest communication between PPE and SPE. To issue work to the SPEs, the PPE sends a MB ID in a mailbox message to the SPE. On completion, the SPE responds also with a mailbox message. Our own experiments, where we also considered mutexes and atomic instructions, show that this is the fastest way to communicate between a master and a slave. However, this scheme requires the PPE to continuously loops over the mailbox statuses.

On a shared memory system, the MB kernels have access to the entire frames located in the external memory. On the SPEs this is not the case, since the frames are too large to fit in the local store. Furthermore, to conserve memory bandwidth, only the relevant parts should be brought in. A block of 48×20 pixels is allocated in the local store to serve as the working buffer for the kernels. This block size is used to store the intra data shown in Figure 3. Although one row of the actual data is only 24 bytes wide, due to DMA restrictions a larger buffer is required. Each DMA must be 16-byte aligned with a size that is a multiple of 16 bytes. Therefore, the width of the buffer spans over three MBs. For similar reasons the size of the motion data buffer is larger than the actual data. Figure 8 depicts the DMA transfers between external memory and local store.

Between the DMA transfers the MB kernels are applied. Intra-prediction is applied after (2) and motion compensation after (3), followed by the IQ and IDCT. After (4) the deblocking filter is applied. In the TP implementations all the DMA transfers are blocking as the necessary data cannot be predicted beforehand.

4.3 Ring-Line Implementation

In the novel Ring-Line a more static approach is used to leverage 2D-Wave parallelism. Instead of individual MBs, the SPEs process entire scan lines. This allows to better exploit data locality (because adjacent MBs are processed by the same SPE), to overlap communication with computation (because it is known a priori which SPE will need the produced data), and to reduce synchronization overhead (because no central master is needed to keep track of the status of each MB). Figure 9 illustrates the mapping of MBs to SPEs for the case of 3 SPEs.

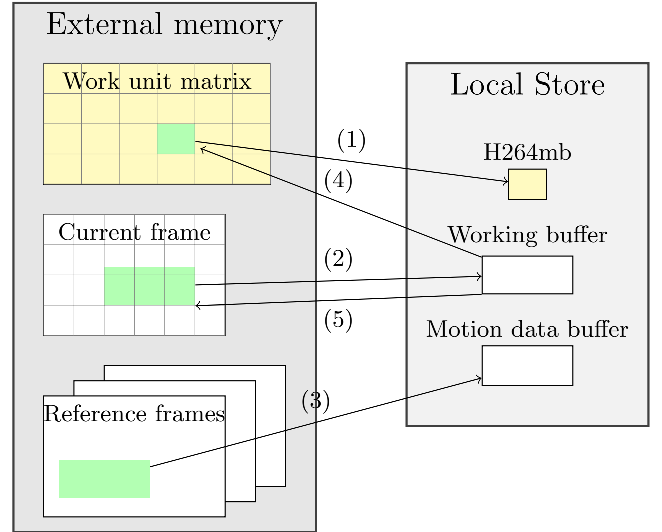

Figure 8: DMA transfers needed to decode a single MB. (1) After receiving the MB ID, the corresponding `H264mb` structure is transfered to the local store. (2) This is used to fill the working buffer with the correct intra data. (3) If motion compensation is required, the relevant part of the motion data is retrieved for each partition. (4) Before applying the deblocking filter, the unfiltered borders are stored in the surrounding `H264mb` structures. (5) After performing all the MB kernels the working buffer is written back.

The figure shows that only MBs on the same line and the next line depend on the MBs processed by an SPE. This static mapping of MBs to SPEs provides predictable targets for the intra data. Since the MBs are processed in line order, a part of the intra data is kept locally, while the other part is sent to the next SPE that processes the next line. The SPEs are, therefore, logically connected in a ring network, hence the name Ring-Line. Furthermore, the control is no longer centralized but distributed. SPEs signal that they have processed a MB by sending non-blocking, one-way control signals to the next SPE, together with the intra data, rather than to a central master.

To support pre-sending of the intra data, local store buffers are required. In the chosen implementation a buffer with a

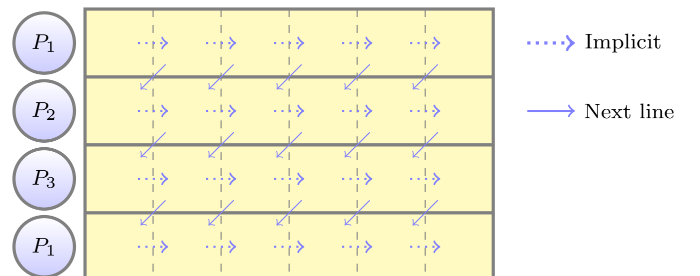

Figure 9: Dependencies in the Ring-Line algorithm. Dependencies to blocks on the same line are implicit, i.e., they are satisfied by the order in which each SPE processes the blocks.

width of an entire line and a height of 20 pixels is used. The line buffer is used as the working buffer and the target of the intra data. By directly writing the intra data in the right place of the line buffer, additional copy steps are avoided. In the RL implementation all intra data is kept on-chip.

Because of DMA size and alignment restrictions, two adjacent MBs are combined in the write back. Additionally, the write back step is delayed by one MB. This allows the deblocking filter to modify the data before writing it back to the frame. This results in writing each pixel in the frame only once, while the TP implementation writes it 3 to 6 times. The pre-sending and the write back step are shown in Figure 10 on the next page.

The static line assignment can also be exploited to perform pre-fetching. Since it is known in what order the MBs are processed the `H264mb` work units and motion data can be pre-fetched to the local store to hide the memory latency. Because the motion data depends on the work unit, triple buffering is used for the work units. The motion data remains double buffered. The communication parts of the motion compensation kernel had to be decoupled to allow for this pre-fetching. Figure 11 shows the scheduling of the DMA steps and the processing. Each DMA step (non-rounded rectangles) is non-blocking. Their completion is checked before issuing the same step in the next time slot. Using this scheme completely hides the combined DMA latency as long as its execution time is shorter than that of the MB processing.

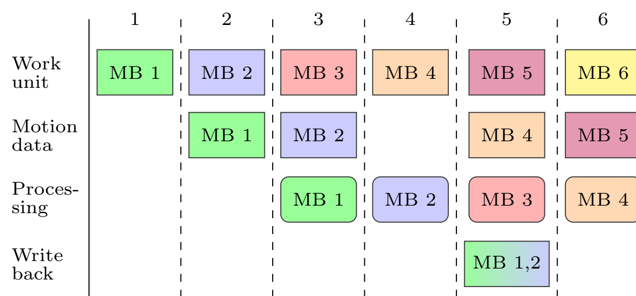

Figure 11: Order of DMA steps and processing. The DMA steps in the non-rounded rectangles are non-blocking. They are checked for completion in the next slot.

5. EVALUATION

In this section the performance of the two implemented solutions is analyzed in detail. First, we investigate the theoretical scalability limits of the TP and RL implementations. Second, in Section 5.2, the memory bandwidth requirements of both implementations are analyzed in order to determine when the memory subsystem becomes the bottleneck. Third, in Section 5.3, the actual performance results obtained on a 16-SPE Cell blade platform are presented and compared to the theoretical scalability. Finally, in Section 5.4, a qualitative comparison of TP and RL as well as other existing approaches is provided.

5.1 Theoretical Scalability

As discussed in Section 2.3.1, the amount of parallelism exhibited by the 2D-Wave approach is not constant as it suffers from ramping and dependency stalls. The impact of these stalls is analyzed in this section. Furthermore, we also investigate the scalability beyond 16 PEs. To derive the theoretical scalability a high level algorithm simulator is used that emulates the scheduling behavior of the TP and RL implementations. This includes the effects of ramping and dependency stalls. In the analysis perfect platform parameters, i.e., zero communication and synchronization overhead are assumed. The scalability is based on real MB decoding times, extracted from the 100-frame FHD HDVideoBench sequences. Figure 12 shows the scalability results, i.e., the speedup of both TP and RL over itself running on a single core, for 2 to 64 PEs.

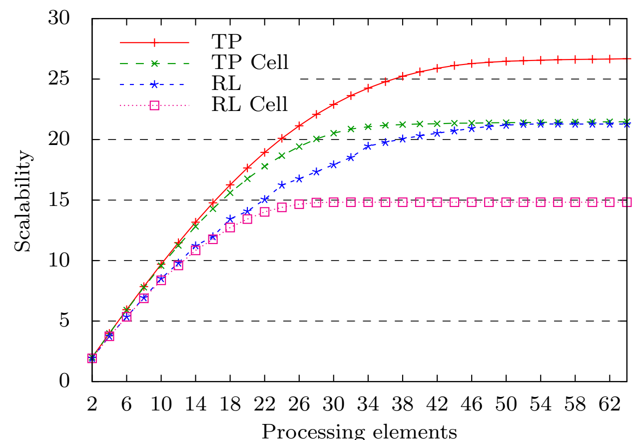

Figure 12: Theoretical scalability of the TP and RL implementations for FHD HDVideoBench sequences. The 'Cell' variants include the impact of the required larger horizontal spacing for satisfying the DMA alignment and size restriction. In all cases perfect platform conditions are assumed.

The figure shows two variants for each implementation. The 'Cell' variants simulate the actual implemented schemes and have lower scalability. The maximum 2D-Wave parallelism, defined in Equation (1), does not fully apply to the Cell implementations due to DMA alignment restrictions. In Figure 4 it is shown that in 2D-Wave, free MBs are two blocks apart in the horizontal direction. This is denoted as horizontal spacing. However, in the TP implementation a working buffer with a width of three MBs is written back. To avoid overlapping frame writes the horizontal spacing was increased to three MBs. In RL the write back step combines two MBs and in addition it is delayed a slot, which effectively increases the spacing to four. Equation (2) revises Equation (1), where the factor 2 is replaced with $Space_{hor}$.

$$RevParMB_{max,2D} = min(\lceil N_{MB,hor}/Space_{hor}\rceil, N_{MB,ver}) \qquad (2)$$

The equation shows that a larger spacing between MBs decreases the parallelism. This is expected as the parallelism is limited by the number of horizontal MBs in a line. The parallelism is, therefore, higher on architectures with smaller DMA constraints.

In both cases the TP implementation achieves higher scalability than RL. The TP is less effected by dependency stalls due to its dynamic scheduling. In other words, because the TP schedules more fine-grained tasks (single MBs), it

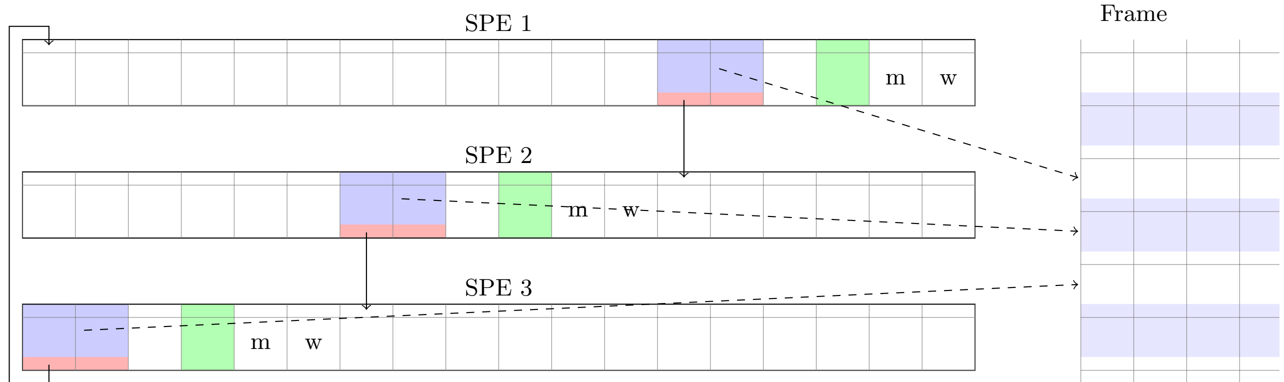

Figure 10: The intra data (red) is *pre-send* to the line buffer of the next SPE. The transfer is issued together with the write back to frame (blue). The motion data (m) and the workunit (w) of the next MBs are *pre-fetched*. All DMA transactions take place concurrently to the processing of the current MBs (green).

achieves a better load balancing than RL, which uses coarser tasks (entire lines of MBs). The Cell variant of RL suffers relatively more due to its larger spacing. Up to 16 PEs, however, the effects are small. For 16 PEs, due to DMA constraints, the scalability decreases from 14.8x to 14.3x for TP and from 12.0x to 11.7x for RL.

5.2 Memory

5.2.1 Local Store Requirements

The SPE local store of 256 kB is shared by the program image and the local data structures. The program image sizes are quite similar with 115 kB for the TP implementation and 118 kB for RL. This is mostly occupied with the intra-prediction and motion compensation functions, with 28 and 64 kB, respectively.

The size of the local data structures, however, is very different. While the TP implementation requires only 25 kB, Ring-Line requires 138 kB which is almost all the remaining local store space. For resolution higher than FHD, larger line buffers are required, which will not fit in the local store. This can be solved using a less memory hungry variant, which only requires 84 kB of local store memory. The memory requirements can be reduced by separating the line buffer in a dedicated intra data buffer for pre-sending and a working buffer whose size is equal to the size of the working buffer in the TP implementation. In this variant, however, additional local store copy operations are required, but a local store size of 256 kB is sufficient to scale up to Super HiVision resolutions ($8k \times 4k$). The performance of this variant is slightly worse (at most 5%) than the presented RL implementation. Therefore and for brevity, these results are omitted.

5.2.2 Memory Bandwidth

The memory bandwidth requirements of the SPEs are determined by the DMA operations. Most of them are list DMAs, which provide strided access to main memory. The throughput of list DMA transfers is around 5 times lower than normal DMAs [6]. Furthermore, small sized DMA transfers also have a much lower throughput compared to large size transfers. Therefore, when measuring the bandwidth utilization, we do not count the number of bytes transferred, but use a relative metric instead. This metric represents the load on the memory subsystem per second in percentages. To determine the load of a particular DMA step, we measured the throughput of such DMA operations using micro-benchmarks. Table 1 shows the bandwidth utilization per frame.

Table 1: Relative memory bandwidth requirements per frame per second.

DMA	List	**Task Pool**	**Ring-Line**
Intra data	L	0.28%	-
Write back	L	0.26%	0.10%
Motion data	L	0.07-0.50%	0.07-0.50%
Unfiltered borders	-	0.07%	-
Work unit	-	0.07%	0.07%
Total		0.75-1.18%	0.24-0.67%

The tables shows that RL is between 1.75x to 3x more efficient in terms of memory bandwidth requirements. This factor depends on the number of MBs in which motion compensation is used. The BlueSky sequence (0.50%) uses a lot of motion compensation while the RiverBed sequence (0.07%) uses only very little. The bandwidth utilization puts a limit on the maximum achievable performance of the decoders. The BlueSky sequence uses 1.18% (TP) and 0.67% (RL) of the bandwidth per frame on average. Thus a performance of 84.7 and 149.2 fps (frames per second) is the maximum achievable for TP and RL, respectively. For the RiverBed sequence this is much higher at 133.3 and 416.7 fps. These values should be regarded as optimistic. The effects of imbalanced memory load due to parallelism ramping and variable requirements of P- and B-frames are neglected.

5.3 Experimental Results

The experimental results have been obtained on the Cell Blade platform described in Section 3. The performance in frames per second (fps) of both implementations are shown for up to 16 SPEs in Figure 13.

The figure shows that the RL implementation achieves a much higher performance. On 16 SPEs, RL achieves on average 139.6 fps while the TP implementation attains only

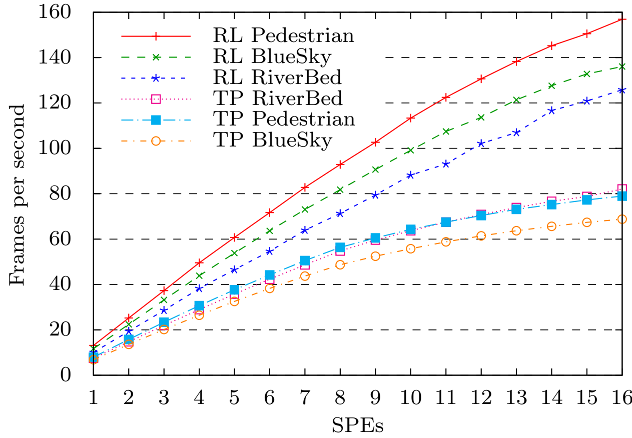

Figure 13: Performance in frames per second of the HDVideoBench FHD BlueSky, Pedestrian and RiverBed sequences.

76.6 fps. We also see that the base performance of a single SPE is higher, which partly explains the performance difference. The other factor is the obtained scalability, which is depicted in Figure 14. In this figure the average scalability of TP and RL over their respective single SPE performance results and the theoretical expected scalability are shown.

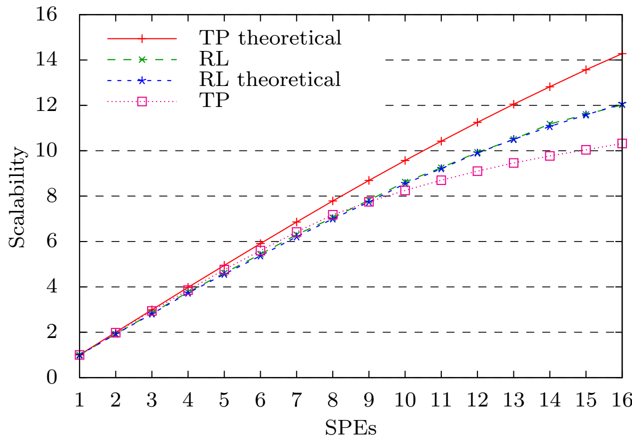

Figure 14: Average scalability of the TP and RL implementations compared to the theoretical scalability results obtained using the simulator.

The average scalability of RL at 16 SPEs is about 15-20% higher than the scalability of TP. This is somewhat surprising as the theoretical scalability obtained using the simulator shows that TP is more scalable than RL. In reality, however, the TP implementation cannot achieve the theoretical scalability due to shared memory contention and the central master, which provides synchronized access to the shared data structures. The actual scalability of RL, on the other hand, almost exactly follows the theoretical scalability due to hiding the memory latency and a distributed control mechanism.

We further investigate the performance advantage of RL over TP in detail by profiling the execution of the BlueSky sequence, which uses a lot of motion compensation. Figures 15 and 16 show the profiling results for TP and RL, respectively. The figures break down the average MB decoding time into processing time, DMA startup cost and waiting time, time needed for synchronization and dependency stalls, and time lost due to parallelism ramping. In both figures the MB processing time stays constant, as expected. The relative time lost due to parallelism ramping is also as expected and similar for both implementations. The DMA cost and the time needed for synchronization and dependency stalls, however, are much higher for TP and, furthermore, increase with the number of SPEs.

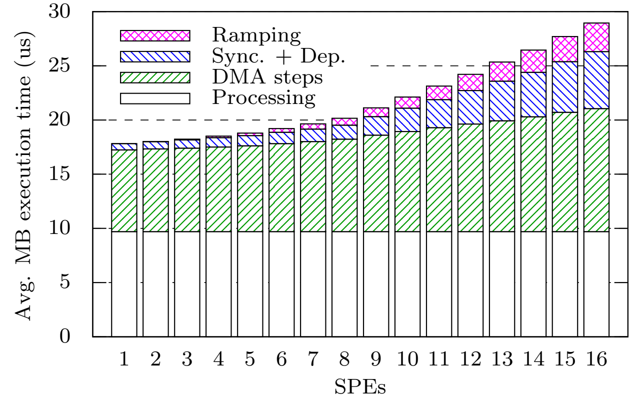

Figure 15: Breakdown of the average MB execution time in the TP implementation using BlueSky.

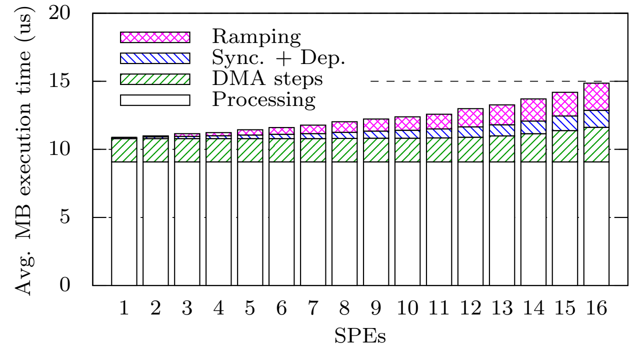

Figure 16: Breakdown of the average MB execution time in the RL implementation using BlueSky.

The DMA cost consists of the time needed to set up DMA operations and the time waiting for completion. In RL, the DMA waiting time is hidden by the computation so the DMA cost depicted in Figure 16 is due to time needed for setting up DMA operations. For more than 13 SPEs, the DMA cost starts to increase slightly, which indicates that the DMA latency has increased to a point where it cannot be completely hidden. To put the results into perspective, the obtained performance for 16 SPEs of 136.1 fps is at 91% of 149.2 fps, at which point the memory bandwidth is saturated. With 68.8 fps, TP has saturated 81% of the bandwidth. Since for the TP implementation the DMA latency is not hidden, the memory contention adds directly to the average MB execution time.

The time needed for synchronization and the time lost due to dependency stalls could not be measured separately. A good estimate of the synchronization overhead of TP is the difference between TP and RL in this part, as in RL the synchronization overhead is close to zero. The synchronization overhead incurred by TP increases with the

number of cores due to its centralized execution model. A larger increase can be noticed after 8 SPEs, partly caused by the additional off-chip latency to the second Cell processor. Due to the rapid increase in synchronization overhead, the TP implementation is not able to scale much further with more SPEs. In RL, on the other hand, the synchronization overhead does not increase at all because only consecutively numbered SPEs need to synchronize. The observable increase is solely due to dependency stalls.

Table 2: Single-core performance in frames per second (fps) of the original sequential FFmpeg and parallel implementations. The original sequential FFmpeg results are obtained using the PPE. For the TP and RL a single SPE is used.

Sequence	**Sequential**	**Task Pool**	**Ring-Line**
BlueSky	11.51 fps	6.84 fps	11.69 fps
Pedestrian	14.82 fps	7.98 fps	13.14 fps
RiverBed	11.50 fps	7.45 fps	10.08 fps

Finally, the base performance of the parallel implementations is compared to the original sequential FFmpeg running on the PPE. The results in Table 2 show that the sequential FFmpeg running on the PPE is faster on average than the parallel implementations running on a single SPE. The RL performance is, however, very close and even surpasses the PPE performance for BlueSky. This is quite good considering that in terms of chip area the PPE is about three times as large as a SPE.

5.4 Qualitative Evaluation

In the previous section two parallel H.264 implementations have been analyzed. The TP implementation is based on a centralized master-slave model, while RL has a static distributed execution pattern. However, several other solutions are available that have features of both. In this section we broaden our evaluation by including three of them.

The first two of the three are variants of the TP implementation, Tail Submit (TS) [10] and a strategy referred to as Multiple Task (MT), which add opportunities for pre-fetching. In TS it is assumed that the next MB to process is the MB to the right of the current. Often the dependencies of the right MB are resolved after processing the current. In this approach the data for the next MB is speculatively pre-fetched. In MT multiple free MBs are sent to an SPE when available. This allows for pre-fetching when there is more than one MB available for the SPE. Compared to TS it does not speculate and, therefore, does not increase memory bandwidth requirements. There is, however, less opportunity for pre-fetching as more than one MB has to be available before the processing starts. In both cases the chances for pre-fetching diminish when increasing the number of SPEs.

The third is the implementation of Baker et al. [5]. Baker et al. developed independently and concurrently, an approach that resembles our RL approach. Similar to RL consecutive SPEs synchronize and thus the control is distributed. However, no pre-fetching of motion data and pre-sending of intra data is implemented, which highly increases the memory bandwidth requirements.

In Table 3 a qualitative comparison of the five implementations is provided. Adding TS or MT to our TP implementation requires additional local store space for the pre-fetching. While they would improve performance, less scalability than TP is expected due to a higher single core base performance. Baker's implementation, on the other hand, has the same theoretical scalability as our Ring-Line. The obtainable scalability and performance, however, is lower as it does not hide the memory latency by pre-fetching.

Table 3: Qualitative comparison of four 2D-Wave parallelism implementations. Obtainable scalability refers to how much of the theoretical scalability is obtainable on the Cell. Portability describes in what manner it is possible to use the implementation on different architectures.

Criterium	**TP**	**TP (TS)**	**TP (MT)**	**Baker**	**RL**
LS usage	++	+	+	−	0
BW usage	−	−−	−	0	++
Theoretical scalability	++	+	+	0	0
Obtainable scalability	−	0	0	+	++
Performance	−	0	0	0	++
Portability	+	+	+	0	−

In terms of obtainable scalability and performance RL is the best solution, because it is almost unaffected by contention effects. As long as the parallelism increases, RL can scale to any number of cores. For the TP implementations to scale, the synchronization latency needs to reduced proportionally. From the results we have seen that the scalability is already less than ideal. It can be concluded that the TP implementations are not able to leverage much performance from more cores, unless the synchronization overhead is reduced considerably. In addition to more cores, RL is also able to leverage faster cores. Faster cores imply lower MB execution times, which in turn increases the significance of the synchronization performance, much like it is the case with more cores. In both cases the memory subsystem capabilities need to improve proportionally.

The TP variants are portable as they can be implemented on both cache-based and local store architectures, with minor effect on the other characteristics of Table 3. RL, however, requires a local store and explicit inter-core communication as provided by the Cell processor. Without these elements both performance and scalability are expected to suffer. This also hold true for Baker's implementation although the effects are not as severe, since the Cell memory hierarchy is exploited to a lesser extend.

6. CONCLUSIONS

In this paper we have presented and analyzed two parallel H.264 decoding strategies on the Cell architecture. Both the Task Pool (TP) and the novel Ring-Line (RL) approach exploit 2D-Wave parallelism, but TP relies on a centralized task pool, while RL has a static but distributed control.

While the TP implementation exhibits higher theoretical scalability, RL is superior both in terms of performance and obtainable scalability. The actual scalability of RL follows the theoretical expectation, resulting in a scalability of 12x on 16 SPEs. The obtained scalability of TP is, on the other

hand, 10.3x where 14.3x is theoretically expected. This efficiency loss originates from memory contention and synchronized access contention, both of which increase with the number of cores. The contention effects limit the scalability of TP independent of the amount of parallelism and, therefore, also limits the applicability of TP to many-cores. The RL implementation does not suffer from these contention effects and thus scales effortlessly to many-cores with a Cell-like memory hierarchy, provided larger resolution inputs are used and proportional memory subsystem improvements are made. Whereas the scalability of the two implementations is rather similar on 16 SPEs, with 12x for RL and 10.3x for TP, the actual performance of RL is almost two times as high as that of TP. The former achieves a throughput of 139.6 frames per second (fps) while the latter achieves only 76.6 fps. The additional performance difference in favor of RL is due to hiding the memory latency by pre-fetching.

We conclude that in order to exploit the full performance of a Cell-like architecture, the following programming principles should be followed. First, a distributed control scheme should be used. A central control thread easily bottlenecks the system and therefore limits scalability and reduces performance. Second, a data oriented programming style should be used in order to minimize off-chip bandwidth requirements and to hide the memory latency. This not only improves performance but also scalability. Our RL implementation shows that for applications with predictable tasks, static approaches outperform dynamic approaches. Static approaches reduce the cost of dependence checking and task distribution, and allow to exploit locality better.

7. ACKNOWLEDGMENTS

The authors would like to thank Barcelona Supercomputing Center (BSC) for making their Cell Blades available. Also we would like to thank Mauricio Alvarez of UPC for making his initial implementation available and for the discussions. Finally, we would like to thank the anonymous reviewers for their remarks.

8. REFERENCES

[1] International Standard of Joint Video Specification (ITU-T Rec. H.264| ISO/IEC 14496-10 AVC), 2005.

[2] M. Alvarez, A. Ramirez, A. Azevedo, C. Meenderinck, B. Juurlink, and M. Valero. Scalability of Macroblock-level Parallelism for H.264 Decoding. In *Proc. Int. Conf. on Parallel and Distributed Systems*, 2009.

[3] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In *Proc. IEEE Int. Symp. on Workload Characterization*, 2007.

[4] H. Baik, K. Sihn, Y. Kim, S. Bae, N. Han, and H. Song. Analysis and Parallelization of H.264 Decoder on Cell Broadband Engine Architecture. In *Proc. Int. Symp. on Signal Processing and Information Technology*. Samsung Electron. Co., 2007.

[5] M. Baker, P. Dalale, K. Chatha, and S. Vrudhula. A Scalable Parallel H.264 Decoder on the Cell Broadband Engine Architecture. In *Proc. IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis*, volume 7, 2009.

[6] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its First Implementation: a Performance View. *IBM Journal of Research and Development*, 51(5), 2007.

[7] Y. Chen, X. Tian, S. Ge, and M. Girkar. Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures. In *Proc. Int. Parallel and Distributed Processing Symposium*, volume 18, 2004.

[8] The FFmpeg Libavcodec. `http://ffmpeg.org`.

[9] A. Gulati and G. Campbell. Efficient Mapping of the H.264 Encoding Algorithm onto Multiprocessor DSPs. In *Proc. SPIE Conf. on Embedded Processors for Multimedia and Communications*, 2005.

[10] J. Hoogerbrugge and A. Terechko. A Multithreaded Multicore System for Embedded Media Processing. *Transactions on High-Performance Embedded Architectures and Compilers*, 3(2), 2008.

[11] F. Khunjush and N. Dimopoulos. Extended Characterization of DMA Transfers on the Cell BE processor. In *Proc. 13th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-08), held in conjunction with IPDPS*, 2008.

[12] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel Scalability of Video Decoders. *Journal of Signal Processing Systems*, 57(2), 2009.

[13] T. Oelbaum, V. Baroncini, T. Tan, and C. Fenimore. Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard. In *Proc. Int. Broadcast Conf.*, 2004.

[14] D. Pham et al. The Design and Implementation of a First-Generation CELL Processor. In *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, 2005.

[15] A. Rodriguez, A. Gonzalez, and M. Malumbres. Hierarchical Parallelization of an H.264/AVC Video Encoder. In *Proc. Int. Symp. on Parallel Computing in Electrical Engineering*, 2006.

[16] M. Roitzsch. Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding. In *Proc. IEEE Real-Time Systems Symposium*, volume 27, 2006.

[17] E. van der Tol, E. Jaspers, and R. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.

[18] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[19] X264. A Free H.264/AVC Encoder. `http://www.videolan.org/developers/x264.html`.

[20] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell. Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design. *Proc. Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[21] X. Zhou, E. Q. Li, and Y.-K. Chen. Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.