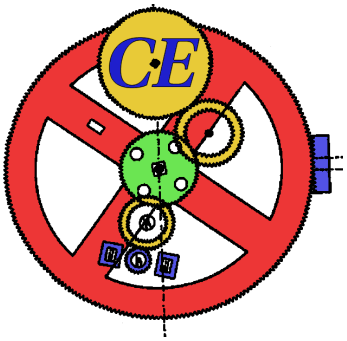


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Parallel H.264 Decoding Strategies for Cell Broadband Engine

C. C. Chi

CE-MS-2010-04

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 the TP implementation 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.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Parallel H.264 Decoding Strategies for Cell Broadband Engine

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

C. C. Chi
born in Rotterdam, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Parallel H.264 Decoding Strategies for Cell Broadband Engine

by C. C. Chi

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 the TP implementation 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.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-04

Committee Members :

Advisor:	B. Juurlink, CE, TU Delft
Advisor:	C. Meenderinck, CE, TU Delft
Chairperson:	Stamatis Vassiliadis, CE, TU Delft
Member:	Z. Al-Ars, CE, TU Delft
Member:	K. Goossens, CE, TU Delft
Member:	H. Sips, CE, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the Computer Engineering Department of the Delft University of Technology for giving me permission to commence this thesis. I am deeply indebted to my supervisors Prof. Dr. Ben Juurlink and PhD. Cor Meenderinck from the Delft University of Technology whose help and stimulating suggestions helped me in all the time of research and writing of this thesis. I want to sincerely thank Mr. Mauricio Alvarez from the Department of Computer Architecture (DAC) of the Technical University of Catalonia (UPC) for providing his earlier work on the FFmpeg code.

C. C. Chi
Delft, The Netherlands
February 1, 2010

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 [30] will soon create bottlenecks. The IBM Cell processor is a heterogeneous multi-core, which comes a long way in addressing these issues. However, the price is paid in programmability, 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 main goal of this thesis is to investigate how to leverage the full performance potential of future many-core architectures. The main obstacle in the move towards many-cores is the increased programming complexity. Recently there is a lot of activity in improving the parallel programming model. For this purpose, however, it is necessary to have insight on how to map potential parallelism on future many-cores. In this thesis we investigate this by parallelizing H.264 on a 16-SPE Cell Blade platform. The H.264 coder/decoder (codec) [1] is presently the most widespread and advanced video codec [20]. Until recently H.264 has been one of the main drivers in the need for more compute capabilities. Due to recent advances in ILP, this is no longer the case. However, H.264 is an evolving standard and future iterations require at least 10x as much compute capabilities to satisfy the never ending need for better visual experiences. Therefore, investigating efficient and scalable parallelization strategies of H.264 has great additional value.

In this work we implement two parallel approaches of an H.264 decoder, both exploiting MB-level parallelism. The first implementation, referred to as the Task Pool (TP), has been presented previously [2] and is based on the worker-server programming paradigm. The server issues work (in this case macroblocks (MBs)) to the workers and also keeps track of the dependencies between the MBs. The second implementation is a novel approach referred to as Ring-Line. In the latter the workers 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 and experimental results. The theoretical scalability is better for the TP implementation. However, both the scalability and the absolute performance measured in the actual implementations favor the RL approach. Most of the performance advantage originates from hiding the memory latency.

Independently, Baker et al. [7] have implemented a similar approach 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. Surprisingly, they do not fully exploit one of the key features of the Cell processor: explicit data management. In contrast to our implementation, they do not apply pre-buffering neither do they use the possibility to send data from one SPE to another directly. The latter is used to minimize memory latency and off-chip bandwidth utilization, while the former is used to hide the memory access latency. Compared to Baker's implementation, our RL implementation is between 50% to 100% faster, with the caveat that this is derived from comparing the results at 6 SPEs. Baker et al. only provided results for up to 6 SPEs due to limitations of their test platform (PS3).

The thesis is organized as follows. In Chapter 2 a brief overview of H.264 and the Cell architecture is provided. In Section 3 the experimental setup of the test platforms is presented along with several communication micro-benchmarks of the Cell Blade platform. Next, in Chapter4 the 2D-Wave and the novel Ring-Line algorithm is described. For both a theoretical scalability analysis is performed. In Chapter 5, the implementation of the parallel strategies is described. The focus of the implementation lies on porting the FFmpeg [12] code to work with the Cell memory hierarchy. This is described in detail. In Chapter 6 a resource requirements of the two implemented strategies is provided. We primarily focus on the memory and memory bandwidth requirements. The experimental results are presented and discussed in Chapter sec:perfcomp. After the analysis several conclusions are drawn in regards of the performance and scalability of the two approaches. In Chapter 8 several case studies of future H.264 applications are provided. Finally, in Chapter 9 the work is concluded and future work is discussed.

2 Background

2.1 Cell architecture overview

The Cell Broadband Engine [21] is a heterogeneous multi-core consisting of one PowerPC Element (PPE) and eight Synergistic Processing Elements (SPEs). The PPE is a dual-threaded general purpose PowerPC core with 512 kB L2 cache. Its envisioned purpose is to act as the control/OS processor, while the eight SPEs provide the computational power. Figure 2.1 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 [10]. The XDR memory can deliver a sustained bandwidth of 25.6 GB/s.

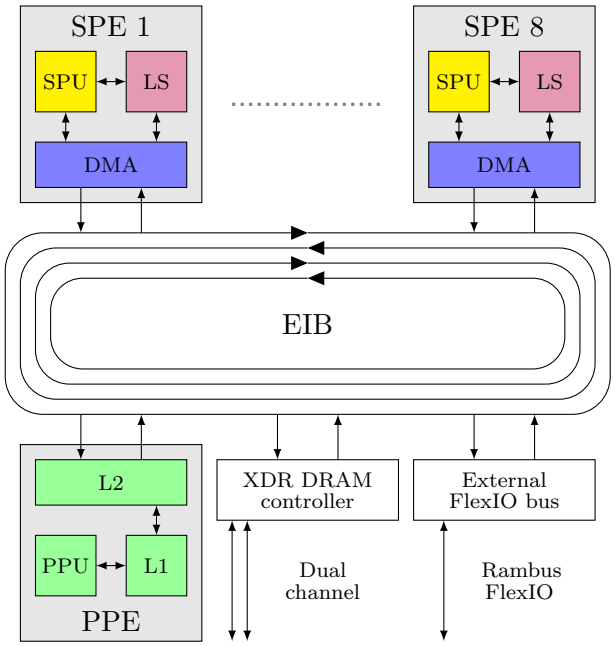


Figure 2.1: Schematic view of the Cell Broadband Engine architecture.

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 cache provides ample speedup, because it reduces the average latency and bandwidth usage of the external (off-chip) memory. With multiple cores there are multiple caches and coherency actions are required. The cache coherency actions grow with a complexity of $O(n^2)$ with increasing core count, 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 for access to 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 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 Overview of H.264 Decoding

Currently H.264 [1, 27] is the best video coding standard is terms of compression rate and quality [20]. Also, it is the most widespread standard for digital video. It is used in Blu-ray, 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.

This thesis focusses on the decoding part of H.264, of which the block diagram is depicted in Figure 2.2. 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 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 the parallelization opportunities.

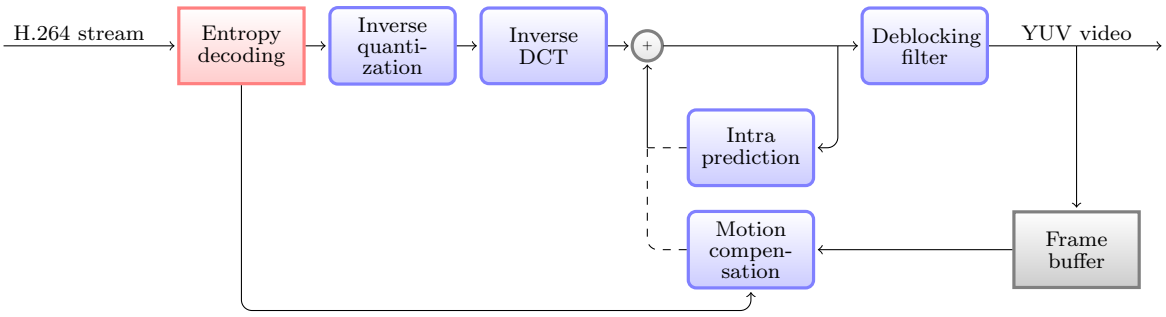


Figure 2.2: Block diagram view of the H.264 decoder.

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 Frames (GOP)-, frame-, and slice-level or apply function-level decomposition. Using the latter, Gulati et al. [15] described a system for encoding and decoding H.264. Data-level decomposition was applied by,

among others, the following. Rodriguez et al. [22] proposed an encoder that combines GOP- and slice-level parallelism. Chen et al. [11] proposed a combination of frame- and slice-level parallelism. Roitzsch [23] proposed a scheme bases on slice-level parallelism by modifying the encoder. Baik et al. [6] has 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 the parallelization strategies above are sufficiently scalable to efficiently utilize emerging many-cores. MB-level parallelization has proven to be much more scalable though and is, therefore, subject of this paper.

In this section we give an short overview of the previously named parallelization opportunities of H.264.

2.3.1 Task-Level Decomposition

In task-level decomposition the macroblock kernels are divided to run on different processors. An example of a possible task parallel solution is given in Figure 2.3.

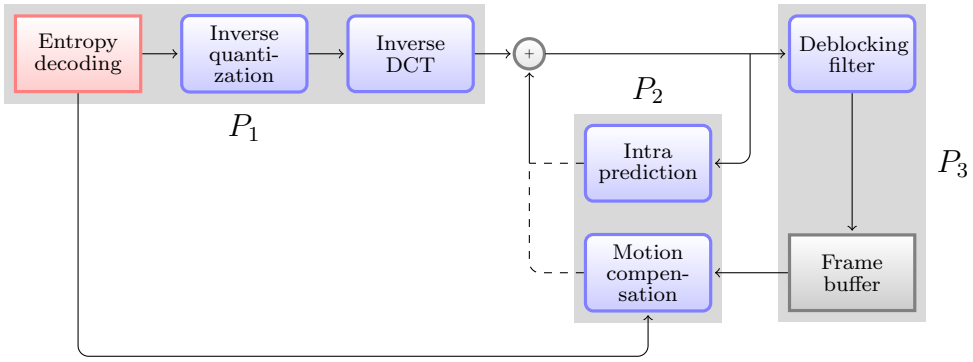


Figure 2.3: Task-level decomposition of the H.264 decoder in a data flow fashion.

The drawbacks of task-level decomposition are limited scalability and imbalanced load. The scalability is limited by the number of kernels that can run in parallel. In Figure 2.3 the parallelism is nearly depleted. Furthermore, not each task takes the same time to complete.

2.3.2 GOP-Level Parallelism

The coarsest grained data parallelism is at the group of pictures (GOP) level. A H.264 streams is build up in GOPs. A GOP always start with an I-frame and all frames in the GOP only have dependencies to other frames in the same GOP. A small GOP is considered to have 25 frames, while a large GOP has around 250. The problem with GOP-level parallelism is that memory requirements limit the scalability. For example a 16 core machine operating on a stream of GOPs with 100 frames would require 100x16x2 frames as a buffer. The additional factor 2 is required for double buffering. In case of a FHD sequence this totals in 10 GB. Also the latency of the stream rises with more processing elements. For 16 this is already 64 seconds.

2.3.3 Frame- and Slice-Level Parallelism

Inside a GOP there is also parallelism in form of frame parallelism. Figure 2.4 shows a typical sequence of frames in a GOP. In the figure the B-frames can be processed in parallel after a P-frame is decoded. In most streams there are only two to three B-frames between consecutive I- and/or P-frames. This limits the scalability to only a few processing elements. The main problem, however, is that H.264 allows B-frames to be used as reference to increase compression rate [13]. In the worst case there is no frame parallelism at all.

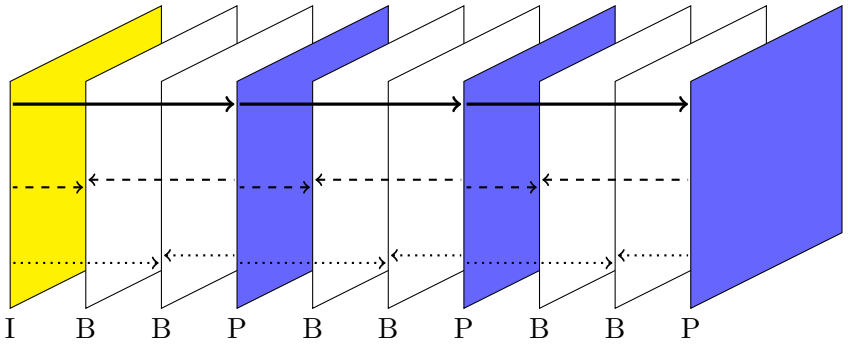


Figure 2.4: A typical GOP sequence with its frame dependencies. In H.264, however, it is allowed to use B-frames as reference.

A frame consists of one or more slices. These slices can be decodes independently. However, like the frame parallelism this is usually limited to a few slices. Also an additional deblocking step has to be performed over the slice edges after the decode, which impacts scalability. Also adding additional slices to a frame decreases compression rate. It is found that this is up to 10% using 8 slices and up to 35% for 64 [18].

Both frame- and slice-level parallelism are dependent on the encoder settings and have to sacrifice compression rate for parallel execution.

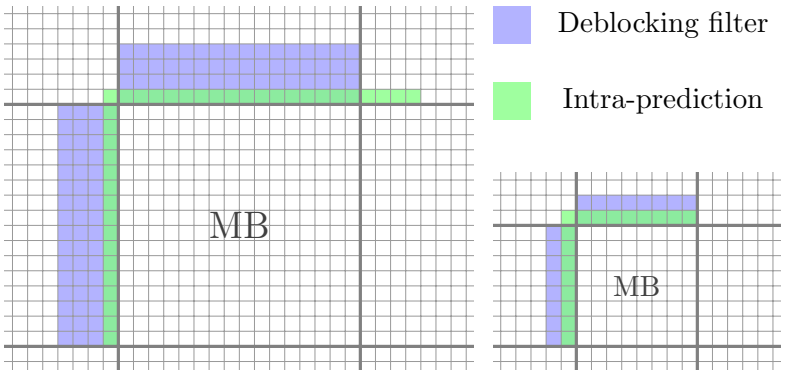
2.3.4 Macroblock-level Parallelism

MB-level parallelism can be exploited in the spatial (within a frame) and the temporal domain (among frames). Spatial MB-level parallelism has first been introduced by Van der Tol et al. [26]. Chen et al. [31] evaluated this approach on a Pentium machine with SMT and multi-core capabilities. Those 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. [18] 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.

2.3.4.1 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 2.5 shows all possible dependencies in the spatial domain. To decode the current MB, data from four surrounding MBs is used. Therefore, those must be fully decoded before the current MB can be processed.



(a) Luma (b) Chroma

Figure 2.5: MB dependencies within the spatial domain.

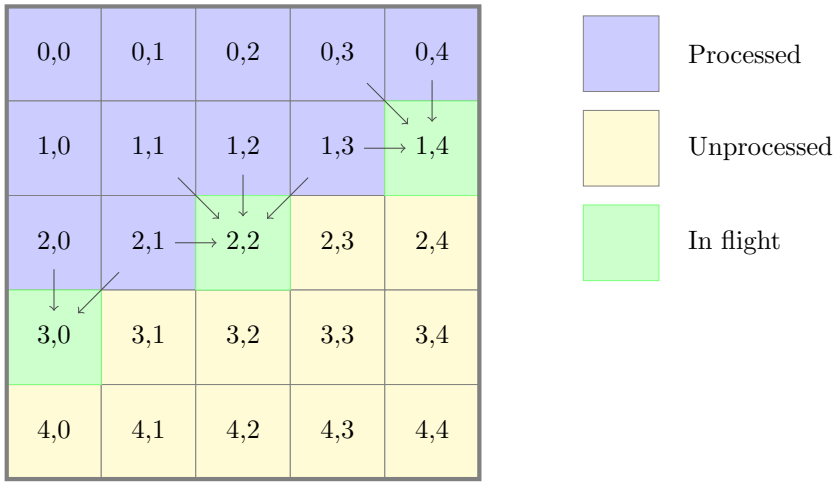


Figure 2.6: 2D-Wave parallelization: MBs on a diagonal are independent and can be decoded concurrently. The arrows represent the dependencies.

As a result of those dependencies, the MBs must be decoded in a proper order. As shown in Figure 2.6, MBs on a 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 the same time. Therefore, the maximum spatial parallelism can be defined as:

$$ParMB_{max,2D} = min(\lceil N_{MB,hor}/2 \rceil, N_{MB,ver}), \tag{2.1}$$

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 interchangeable with the number of concurrent MBs when addressing 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. The 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). Using more PEs increases this inefficiency. The dependency stalls are due to variable MB decoding times. Figure 2.7 illustrates both effects.

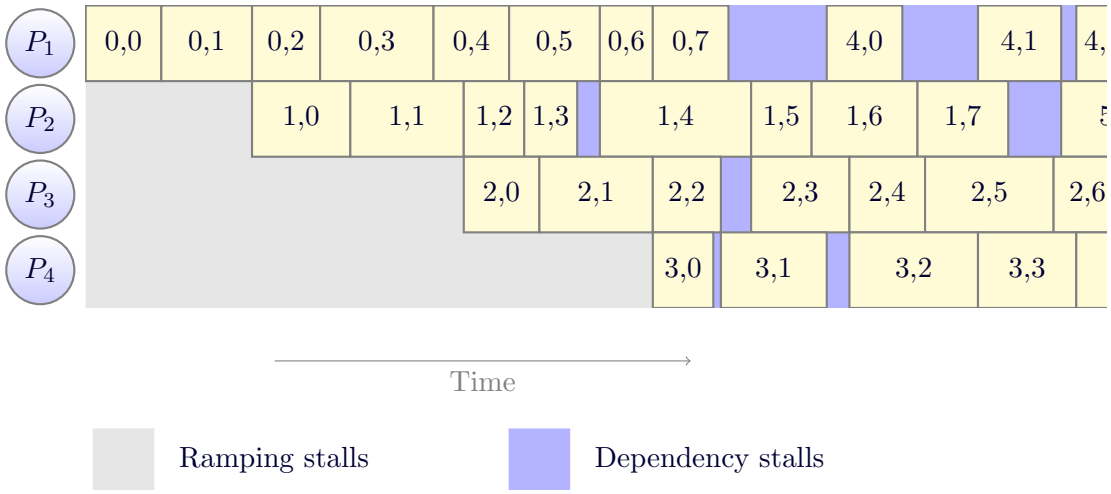


Figure 2.7: Ramping and dependency stalls reduce the scalability and performance of the 2D-Wave parallelization.

2.3.4.2 3D-Wave Parallelism

The concept behind temporal MB-level parallelism is that concurrency can be exploited in multiple frames at the same time. In addition to the intra dependencies, the macroblocks that require motion compensation also have inter-frame dependencies. Hence, data from reference frames is required to perform the decode. The exact reference areas are pointed by the motion vectors. Only these areas of the reference frames are required to perform the decode. MB-level parallelism among frames is referred to as temporal MB-level parallelism. Figure 2.8 illustrates this.

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. [18] showed that the maximum available parallelism for FHD sequences is between 4000 and 7000 while having more than 200 frames in flight.

Although the 3D-Wave approach improves the scalability it is not applied in this work. Introducing the third dimension in parallelization incurs additional overhead costs which only pay of at very large scale systems. Our platform has 16 cores, in which case the 2D-Wave approach provides sufficient parallelism.

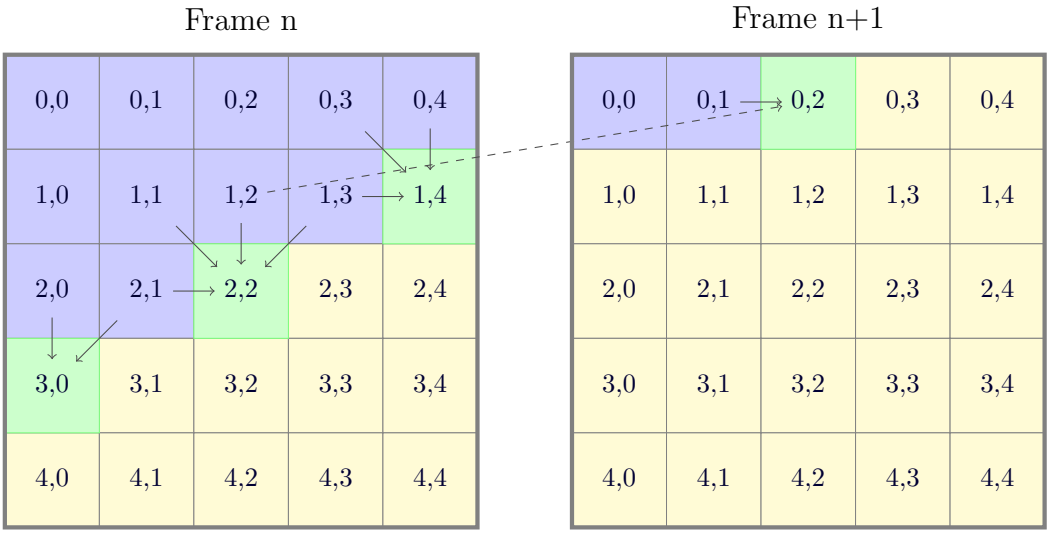


Figure 2.8: Temporal MB-level parallelism.

3 Experimental Setup

This chapter provides the details of experimental setup. The experimental setup is split in two parts, namely the development environment and the actual test setup on which the benchmarks are run. The details are presented in the first two sections of the chapter. Additionally, several micro-benchmarks are performed on the Cell Blade test setup. The focus of the micro-benchmarks is on the communication and synchronization performance. The micro-benchmarks greatly help in choosing the right communication and synchronization mechanisms for the parallel implementations. They will also be very useful when analyzing the results.

3.1 Development Environment

The development environment is a PS3 machine. The PS3 has a single Cell processor, however, only 6 SPEs are usable. The operating system running on the PS3 is the Yellow Dog Linux 6.2 distribution. The kernel version is 2.6.29. The IBM Cell SDK 3.1 [9] has been installed to enable developing Cell programs. Furthermore, the PS3 has a limited external memory of 256 MB XDR memory, of which only around 200 MB was usable. The operating frequency of the Cell processor is 3.2 GHz. The XDR memory can deliver up to 25.6 GB/s of bandwidth. On this platform the parallel H.264 decoders and micro-benchmarks are developed.

3.2 Test Setup

There are several test setups used. The most important one is the Cell Blade located in 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 via the FLEXIO bus. The amount of external memory is 1 GB and is fully usable. The rated memory bandwidth is 25.6 GB/s. The operating system is Fedora Core 7 with kernel version 2.6.22. With two fully accessible Cell processors up to 16 SPEs are usable. The micro-benchmarks and the benchmarks of the parallel H.264 decoders are performed on the Cell Blade. For the timing the hardware counters of the Cell are used. On the Cell Blade these timers have a resolution of 14.8 MHz and negligible call latency.

For the benchmarks the Full High Definition (FHD) sequences of the HD-VideoBench [4] are used. In particular we focussed on the sequences names BlueSky, Pedestrian, and RiverBed. The movies are encoded with X264 [29] 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.

To compare the Cell processor to modern x86 processors, three x86 based machines are used. The tests are run on an Intel Pentium 4 and Core2 Duo, and an AMD PhenomII x4. The platform specifications are listed in Table 3.1.

Specifications	Pentium 4	Core2 Duo	PhenomII x4
Frequency	3.4 GHz	3 GHz	3 GHz
Core count	1	2	4
Parallel threads	2	2	4
L1 Cache	16 kB	2x64 kB	4x128 kB
L2 Cache	1024 kB	6 MB	4x512 kB
L3 Cache	-	-	6 MB
NB Frequency	-	-	2 GHz
Memory size	2 GB	4 GB	4GB
Memory speed	400 MHz DDR	800 MHz DDR2	800 MHz DDR2
Memory interface	Dual 64-bit	Dual 64-bit	Dual 64-bit
Operating System	Fedora Core 8	Suse 10	Ubuntu 9.04
32/64-bit	32-bit	64-bit	64-bit
Kernel version	2.6.26	2.6.27	2.6.29

Table 3.1: X86 test platform specifications.

3.3 Communication Micro-Benchmarks

In this section the communication performance of the Cell Blade is analyzed. The analysis of the performance characteristics is required for choosing the right communication mechanisms. When interpreting the benchmark it should be kept in mind that the Cell Blade contain 2 physical Cell processors. Reduced performance is expected when both processors are used at the same time due to the off-chip latencies and shared memory bandwidth.

The micro-benchmarks can be split in three categories. The first category involves all benchmarks regarding sequential DMA transfers. The second category covers list DMA transfers. The third category explores the latency characteristics of the synchronization mechanisms available on the Cell platform. The overall goal is to investigate how the performance scales with multiple cores. A better understanding of the Cell communication subsystem is required to, on the one hand, make the right choices in the parallel implementation and, on the other hand, to be able to properly analyze the performance of the parallel H.264 algorithm. While prior analysis has been made [17][16], there is a need for more specific analysis. Therefore, the micro-benchmarks are custom made and tailored to specific usage scenarios of the parallel H.264 implementations.

In the benchmarks the hardware decrementer is used for timing purposes. Each SPE has access to an individual decrementing hardware timer, which has a very low latency access latency. The resolution of the timers is 14.8 MHz.

In the following sections the benchmarks performed in each category are presented.

Each benchmark starts with a introduction, followed with the results and the analysis. First, the sequential transfers benchmarks are investigated, followed by the list transfers. Then the synchronization benchmarks are performed.

3.3.1 Sequential DMA Throughput

The micro-benchmarks performed in this section reveal the behavior of the sequential DMA transfers. The benchmark scenarios are restricted to DMA transfers initiated from the SPEs. While the PPE has a DMA unit, it is less powerful than the SPE variants. A DMA unit is able to transfer data from the local store to a global memory space and vice versa. The two types of global memory targets are the external memory and local store spaces.

A sequential DMA transfer is restricted to continuous pieces of memory. The maximum size of a sequential DMA is 16 kB. For the transfer both the global memory and local store targets need to be aligned to a 16-byte boundary. Aligning to 128-byte results in maximum throughput [17]. Both cases will be investigated to determine the differences.

In each benchmark the results are deduced from timing 10,000 iterations of the specific DMA operation. Even though a barrier command is used to synchronize the start of the SPEs, some variations in actual start and end times cannot be avoided. In case of multiple SPEs the results could therefore be inaccurate due to less load on the memory subsystem at the start and the end of the run. To solve this, an additional 10,000 iterations are placed before and after the timed 10,000 iterations to maintain the load on the memory subsystem. The transfer sizes ranges from 16 to 16384 bytes. The number of SPEs ranges from 1 to 16.

3.3.1.1 Sequential DMA Characteristics - External memory

The results for the sequential DMA transfers from external memory are presented in Figures 3.1 and 3.2, for 128- and 16-byte alignment respectively. Both figures present the read performance. Write benchmarks have also been performed. However, the results are not presented due to its similarity with the read performance.

The first thing to notice is that the maximum throughput is higher than the memory is rated at. The XDR memory has a rated bandwidth of 25.6 GB/s, while the graph peaks at 30 GB/s. We suspect that the PPE cache is used as a buffer to speed up the transfers. The maximums are obtained at a rather low SPE number, which strengthens this suspicion. With more SPEs the cache could be too small to have an effect. However running the same benchmark on the PS3 did not exhibit this behavior and the numbers did not exceed 25 GB/s. Therefore, no conclusive explanation can be given.

Furthermore, we can see that up to a DMA size of 1024 bytes the throughput scales approximately linearly with the DMA sizes. The latency difference between a 16 and 1024 byte transfer is therefore quite small. It can be concluded that up to a transfer size of 1024 bytes the memory subsystem is not bandwidth limited, but rather limited in the number of memory operations per second (OPS). A calculation using the numbers from the graphs provides us that the maximum number of memory OPS is around 23 million.

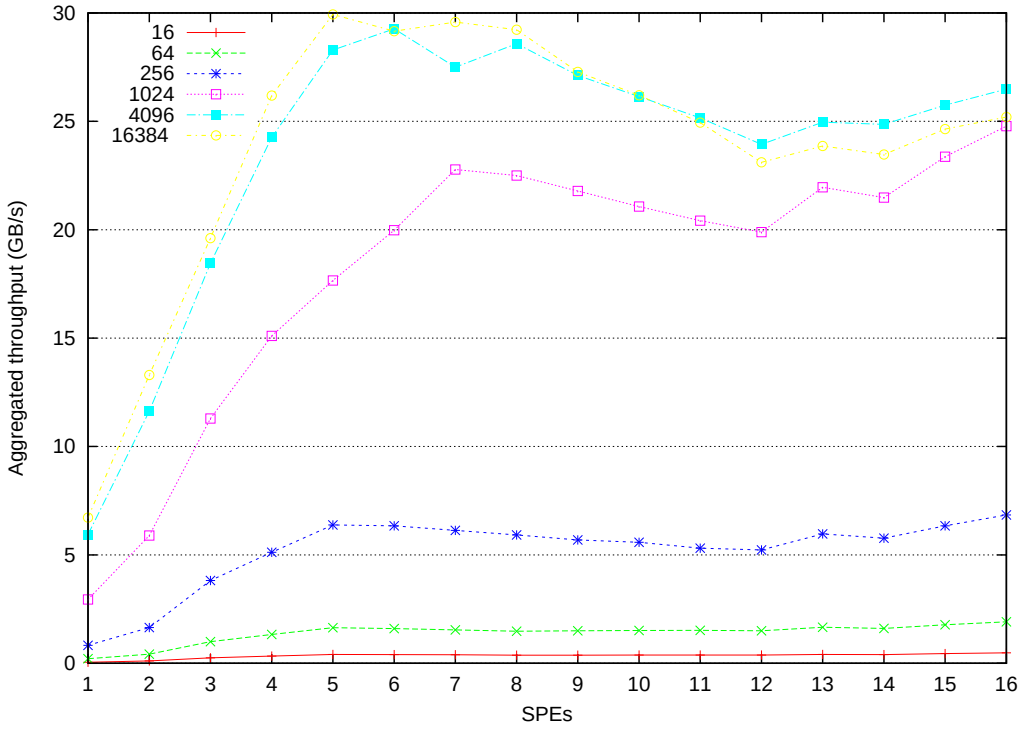


Figure 3.1: External memory throughput of sequential DMA transfers with a 128-byte alignment.

The throughput also scales linearly to about 5 SPEs. This scaling is exhibited for both alignment situations. The difference between the alignment is mainly observed at lower number of SPEs. The throughput with higher number of SPEs is roughly the same.

3.3.1.2 Sequential DMA Characteristics - Inter-SPE

In the inter-SPE benchmark both of the DMA transfer targets are local stores. Each SPE initiates DMA transfers to a pre-determined local store target. Because the SPEs are connected to the EIB, which is basically a ring interconnect, the location of the communicating SPE contexts has a big influence on the aggregated throughput [17]. If the source and the target SPE are neighbors, the contention is minimal. Neighboring SPEs can communicate without interference over the EIB. The runtime system, however, does not allow for direct control in the context placement. Instead it has an affinity option, which allows to specify the neighboring context. However, this is limited to a maximum of 8 SPEs, while our benchmark ranges up to 16 SPEs. Independently of using the affinity option it was discovered that the contexts occasionally are scheduled on different Cell processors when using 8 or less SPEs. In most of the cases when using 8 or less SPEs, the contexts would be scheduled on one Cell processor. However occasionally it uses two processors. More control on the scheduling behavior is therefore desired.

A solution that provides control on the scheduling to use a custom algorithm to deter-

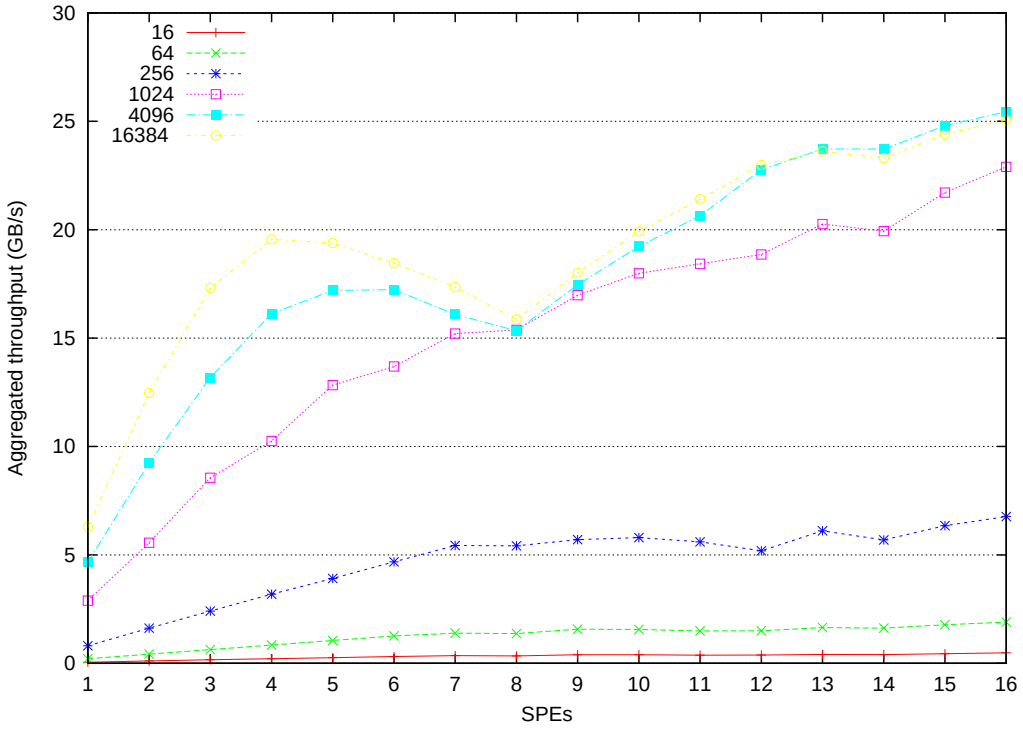


Figure 3.2: External memory throughput of sequential DMA transfers with a 16-byte alignment.

mine the SPE IDs. While this is implemented and considered as a working solution, it is a rather messy way. Even if it is possible to determine the SPE IDs, it is still not known what the relative placement of the SPEs are. This can only be determined by running throughput benchmarks with every imaginable ring configurations. Developers will most likely not use this because of the additional work overhead and code complexity. For the same reason the optimal ring-placed results are not presented. The ring placement is for the same reason not used in the implementation of the parallel strategies. However, it is expected that even with sub-optimal placement the on-chip bandwidth does not form the bottleneck as it is still is much higher than the memory bandwidth.

The results for the inter-SPE DMA transfers are presented in Figure 3.3 and 3.4, for 128- and 16-byte alignment respectively. Again both figures present the read performance, because it is similar to the write performance.

As expected the aggregated throughput scales with the number of SPEs. However, the results vary significantly and the maximum is not nearly as high as the Cell is capable of in theory. This is caused by a combination of the following reasons. First, the EIB is used only in one direction by only reading or writing. The EIB is a bi-directional ring with total throughput of 12.8 GB/s per link. This explains why with a single SPE a maximum throughput of around 11 GB/s is observed. Second, the SPEs are scheduled on physically different Cell processors. The FLEXIO link has lower bandwidth and forms the bottleneck when multiple SPEs of different Cell processors communicate. The SPE

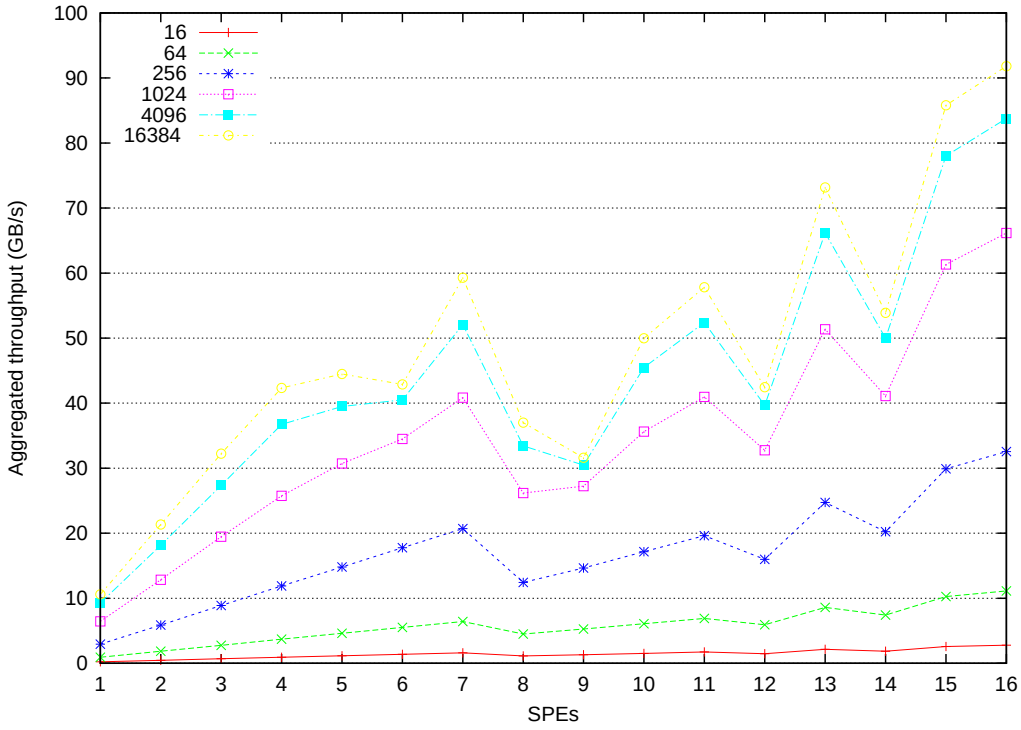


Figure 3.3: Aggregated inter-SPE throughput of sequential DMA transfers with 128-byte alignment.

contexts are scheduled on different processors with more than 8 SPEs and occasionally even when using 8 or less SPEs. The third reason is in the placement of the SPE contexts. Since we did not use affinity or specified the placement of the contexts, contention will occur on several links internally on the Cell processors.

The scaling graph of the 16-byte alignment has less variation. This is probably due to the more regular SPE placement for up to 8 SPE contexts. It is observed that the throughput is only a little lower than with a 128-byte alignment. Also the expected drop in performance when moving to the 9e SPE is observed clearly. The link between the two processors forms the bottleneck here. However, an unexpected drop also occurs when moving to 16 SPEs, which most likely is caused by link contention.

Increasing the transfer sizes imposes similar effects as observed with the external memory throughput benchmarks. The throughput scales approximately linear up to a certain DMA size. With the external memory this was around 1024 bytes, however, with inter-SPE communication this is already at 128-bytes. The memory subsystem is designed to transfer up to 128 bytes at once. Transferring 16 or 128 bytes involves the same amount of work. For the external memory benchmarks the scaling behavior was observed up to 1024 bytes instead of 128. Transfers involving the external memory have a higher initial latency, which causes the relatively small difference between 16 bytes and 1024 bytes.

From the performed micro-benchmark we conclude that sequential DMA transfers

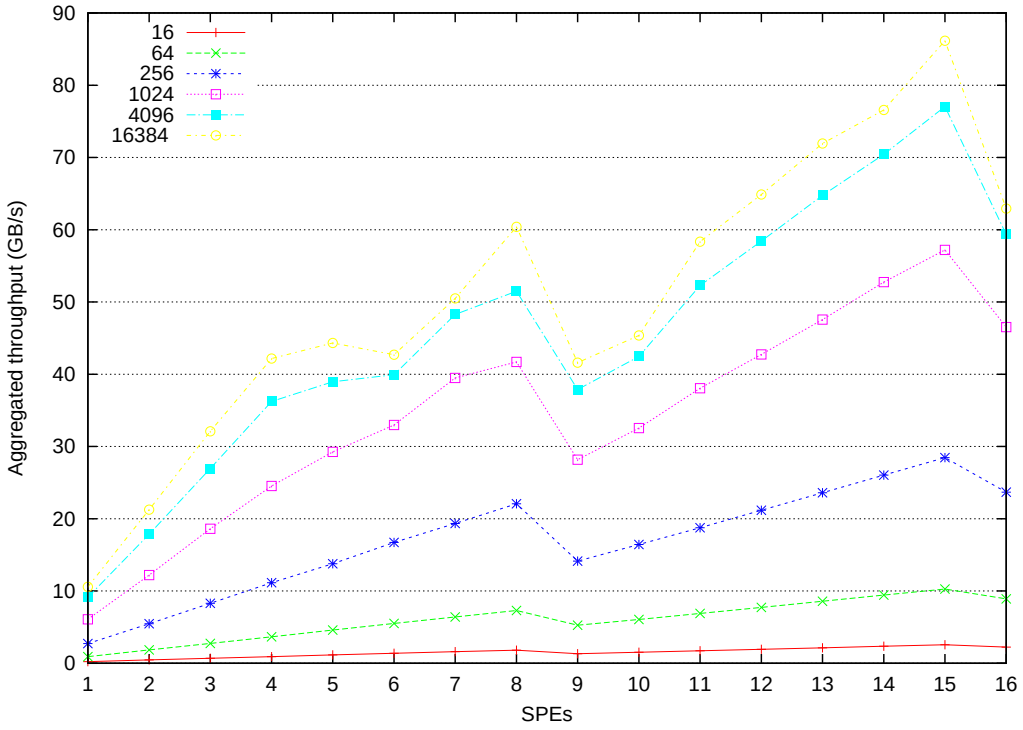


Figure 3.4: Aggregated inter-SPE throughput of sequential DMA transfers with a 16-byte alignment.

should be kept on-chip using inter-SPE transfers when possible. The number of DMA transfers to the external memory should be minimized by packing the data in a single large structure. The effect of alignment is most visible with low number of SPEs. While aligning the DMA targets to a 128-byte boundary optimizes the throughput, the penalty is not very significant. This especially holds for the inter-SPE transfers. It is therefore advised to only apply 128-byte alignment on external memory targets when the memory overhead is relatively small. For local store targets it is advised to use 16-byte alignment to conserve the limited local store space.

3.3.2 List DMA Throughput

In the previous section the characteristics of the sequential DMA transfers were investigated. In this section we continue with the list DMA transfers. In contrast to the sequential DMA, the list DMA uses multiple global address targets. An array of list elements is needed to specify each individual target and size. The number of list elements in a single list DMA transfer can range up to 2048.

Being able to specify multiple targets is very useful for strided memory access. For example, strided acceses are needed to transfer a macroblock from a frame. Normally it would require a DMA transfer for each line of the macroblock, but with the list transfer only a single operation is required. The maximal transfer size of 16 kB still holds for the list transfer. However, this restriction applies to a single list element, instead of an

entire transfer.

For the list DMA benchmark only the results for benchmarks with 16-byte alignment are presented. Also the list transfer element sizes are restricted to 16, 32 and 48 bytes. The purpose of this restriction is to focus only on the list DMA variant used in the implementation of the parallel H.264 strategies, presented in Chapter 5. The list DMAs are used for transferring blocks of data, e.g., a 48x20 picture block. The width of these blocks only vary in the options chosen in this benchmark. To further approach the settings of the implemented strided transfers, the number of list elements is set to 20.

To investigate the performance difference of performing several sequential DMA versus a single list DMA the sequential DMA transfers are also benchmarked with the same settings. The results of the external memory and inter-SPE micro-benchmarks are presented in Figure 3.5 and 3.6 respectively.

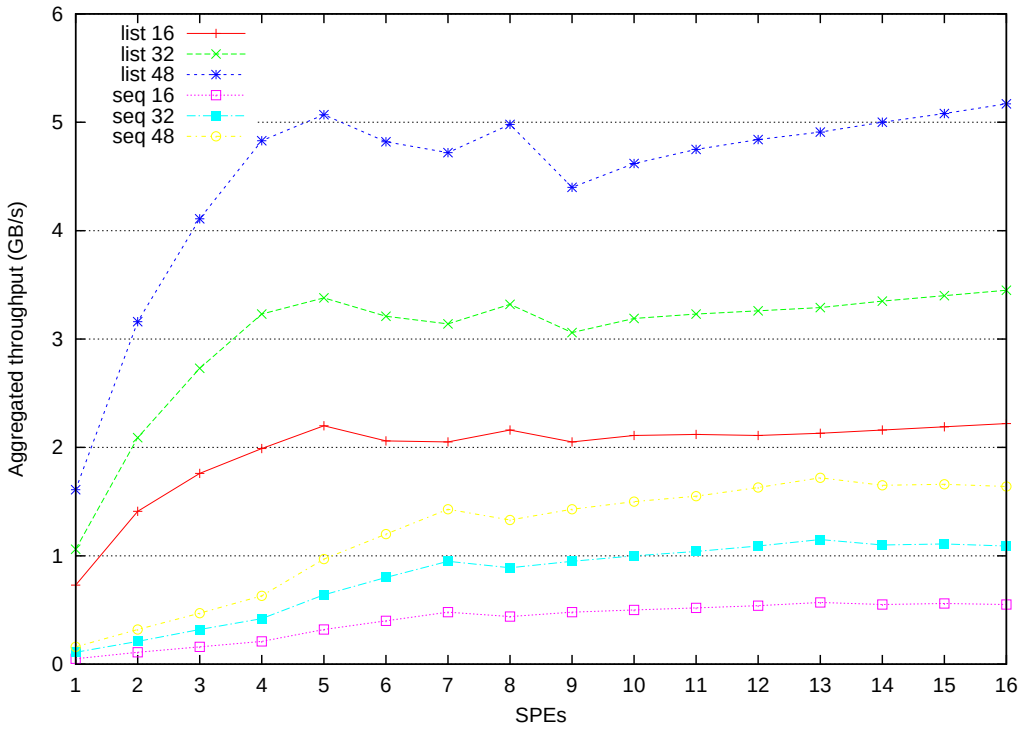


Figure 3.5: External memory throughput of list DMA transfers with a 16-byte alignment.

The first observation to make is that the maximum throughput in the external memory benchmark is much lower than the rated 25.6 GB/s. The results top out at around 5 GB/s for the 48 byte list transfers. In Chapter 6 the composition of the DMA transfers used in the parallel H.264 implementations is revealed to be mostly consisting of list DMA transfers. Therefore, the results of this benchmark give a good view of when the memory subsystem starts to become the bottleneck.

The difference between the list and small sequential DMAs is quite large and we can conclude that sequential DMAs are not an option. We also see that moving to larger widths increases the total throughput approximately linearly. Consequently, this means

that the latencies for different list widths are close to each other. Since the memory subsystem transfers 128-byte at a time, this is to be expected.

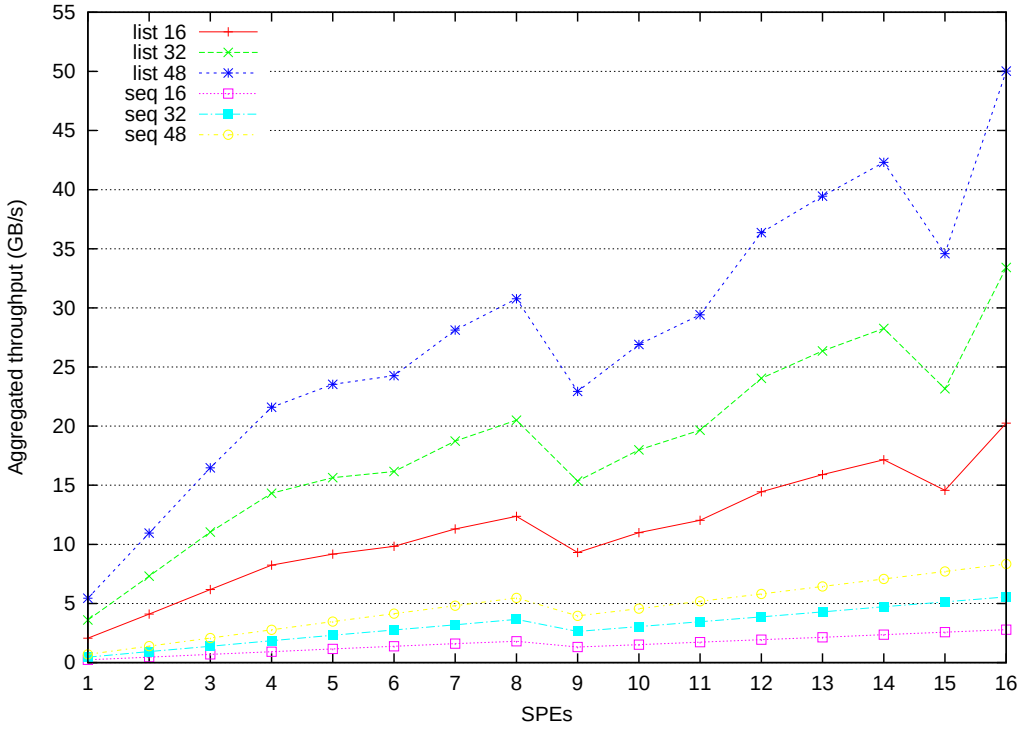


Figure 3.6: Aggregated inter-SPE throughput of list DMA transfers with a 16-byte alignment.

Moving to the inter-SPE benchmark results, a much higher throughput is observed. While the absolute throughput is not as high as obtained with the sequential DMA benchmarks, the ratio of inter-SPE to external memory is much higher for list DMAs. From the figure it is seen that the aggregated throughput tops out at 50 GB/s with 16 SPEs, which is a factor 10 higher than from using external memory. This indicates that the contention caused by sub-optimal context placement impacts the sequential DMA throughput more. In both figures the logical drop in performance is observed when moving from 8 to 9 SPEs. It is more clear in the inter-SPE benchmark since the external link is stressed more.

Perhaps a more interesting parameter than the throughput is the number of list elements transfered per second, shown in Table 3.2. In the table it is shown that the width of the list elements has relatively little influence on the number of list elements per second.

For the list DMA performance, the same conclusion can be drawn as for the sequential DMAs as the inter-SPE throughput is much higher. Therefore, it is advised to keep the transfers on-chip when possible. In case of the list DMAs, however, it has a much bigger than with the sequential DMAs. The relative throughput difference of external memory to inter-SPE is higher for list DMA transfers.

SPEs	External memory			Inter-SPE		
	16	32	48	16	32	48
1	49 M	36 M	36 M	138 M	121 M	122 M
4	134 M	108 M	108 M	553 M	480 M	483 M
16	149 M	116 M	116 M	1358 M	1121 M	1119 M

Table 3.2: Number of transfered list elements per second using DMA lists with 20 elements.

3.3.3 Synchronization Mechanisms

With the last micro-benchmarks the synchronization performance of the Cell processor is investigated. The 2D-Wave parallel algorithm, which is discussed in Chapter 4, has shared data structures which need synchronized access. The focus in this section lies on the synchronization mechanisms that can offer the synchronized access.

On the Cell, synchronized access can be implemented in several ways. First, it can be implemented via the mailbox facilities. The PPE can communicate with a SPE through mailbox messages. A mailbox message consist of a single 32-bit value. The PPE can send mail to each individual SPE context. The arrival of the mail will not generate a interrupt or any other notification. Checking the mailbox for messages has to be done by the programmer. Each SPE can also send a message to the PPE, but not other SPEs. On the PPE side also no notifications are given and the mailbox should be checked actively. With the mailbox facilities a traditional worker-server model can be implemented by using the PPE as the server and the SPEs as the workers. In this implementation only the server needs to access the shared structures, which effectively implements the synchronized access.

The second method is to use the mutex facility. The SPEs can use a mutex to acquire a lock on the shared structure and then perform the operation. However, the SPEs can only operate on data in its own local store. Implementing this will require two DMAs in addition to transfer the shared structure to and from the external memory.

The third method is to use atomic operations. Atomic operations are the only way the SPEs can work directly on the external memory. Atomic operations are always synchronized, but have limited functionality. Several atomic operations might be necessary to implemented the synchronized operation on the shared structures [14]. In some cases the synchronized operation might be too complex to implement with atomic operations. In those cases using atomic instructions is not a viable option.

The final method is to use the callback functionality. A PPE is able to register a function as a callback function. After registering the callback function, it can be called on the SPEs. The difference with regular functions is that the PPE executes them instead. When a SPE calls the callback function, execution halts until it is serviced by the PPE. This very similar to a classic interrupt mechanism in which a registered interrupt routine is performed after raising a certain interrupt.

In the benchmarks the latencies of the four synchronization mechanisms are mea-

sured. The mailbox has two variants, one that is context safe and one that is not. The difference lies in whether it is checked if the context is currently running on the SPE. When there are more SPE contexts than physical SPEs, context scheduling is performed. The context unsafe mailbox assumes that the SPE context will always be on the same SPE. Both variants are investigated in the benchmarks.

The mutex benchmarks also include two sequential DMA transfers in the timings, to simulate transferring a shared structure to the local store. The size of the structure is set to 256 bytes. This is sufficient to transfer the required part of the shared data structure.

Two types of atomic operations are benchmarked. The two types differ in whether they have a return value or not. It is expected that the atomic instructions with a return value are blocking calls and therefore have a higher latency. Both operations are needed to implement the 2D-Wave synchronized function.

Lastly, the contents of the callback function simply contains a `return`-statement, since we are only interested in the call and, thus, synchronization overhead.

The benchmark results are presented in Table 3.3. The mailbox, mutex and callback results should be interpreted as the entire synchronization overhead. The atomic operation numbers present the latency of one operation. Furthermore, the results are an average of 10000 iterations and represent the latencies when little to no contention on the synchronized access occurs.

SPEs	Mbox-safe	Mbox-fast	Mutex	Atomic-sub_and_test	Atomic-inc	Callback
1	1.75	0.43	0.83	0.13	0.13	15.78
4	1.75	0.41	1.50	0.26	0.26	14.58
8	1.76	0.41	1.57	0.26	0.26	9.28
16	2.31	0.86	1.76	0.36	0.26	9.07

Table 3.3: Average synchronization overhead of several synchronization mechanisms in μs

The benchmarks have been for different number of SPEs. It can be seen that the callback functionality is too slow for frequent use. The atomic operations have the lowest latencies among the results. As expected the latency of the atomic increment, which does not have a return value, is a little lower at 16 SPEs than the atomic subtract and test. However, both types could be needed multiple times in more complex operations.

For both mailbox variants and the mutex, the time on 16 SPEs is a little higher than the for 8 SPEs and less. This is logical as the PPE and several SPEs are located on different Cell processors. However, the latency remains within workable ranges. Therefore, the method using the fast mailboxes is considered the best choice.

The results in the table represent the situation when no to little contention occurs. In real applications of course contention could occur. When the work units are too small, the different SPEs contend with each other for the synchronized access.

A defining factor for scalability is the sequential and parallel execution ratio. The centralized synchronization belongs to the sequential part. The synchronization overhead cannot be accurately simulated since it depends on both the platform and the runtime

circumstances. It is not possible to give hard figures for this. What can be done is short investigation on what the synchronization to computation ratio should be to get achieve good scaling.

In case of the mailbox facilities, the PPE handles the requests. Whenever this becomes the bottleneck, the contention turns in to congestion and performance not scale any further. Therefore, the time spend in the work unit has to be larger than 16x0.86= 13.76 us to be able to scale to 16 SPEs. Still it is preferred that it is larger. The closer it gets to this execution time, the higher the average synchronization latency is due to contention.

3.3.4 Conclusions

In this section several communication micro-benchmarks are performed. With the Cell memory hierarchy, the communication is also different. The inter-core and external memory transaction have to be initiated explicitly through the DMA unit. There are two types of DMA operations, the sequential and list variant.

With the sequential DMA transfers the rated bandwidth of 25.6 GB/s is obtained only when using large DMA transfer sizes. With lower than 1 kB transfers the obtained throughput decreased roughly linearly. The inter-SPE bandwidth is much higher and more than 90 GB/s of aggregated bandwidth is achieved using 16 SPEs. However, this is much lower than the theoretical 204.8 GB/s per Cell processors. This has two main reasons. First, the SPE contexts are not optimally placed to each other in a ring structure. This causes link contention over the EIB. Second, two physical processors are used which are connected via the FlexIO bus, which has a maximum rated bandwidth of 37.6/GB. This causes a bottleneck when two or more SPEs of different Cell processors communicate. But the inter-SPE bandwidth is still much higher, especially for smaller DMA sizes. Therefore, the data should be kept on-chip when possible. If it is required to use the external memory it is much more efficient to transfer large blocks, instead of several smaller ones.

With the list DMA transfers strided memory accessed are possible. This is for example required to transfer a macroblock from a frame. In the performed list DMA benchmarks only stride sizes of 16, 32 and 48 were used. The maximum obtained throughput is 5.1 GB/s with a stride size of 48. Using the smaller stride sizes decreased the throughput approximately proportional. This also holds for the inter-SPE list DMA benchmarks. However, the obtained throughput is up to a factor 10x higher. Therefore, it is even more important to keep list DMA traffic on-chip.

The effect of 16- and 128-byte alignment is also investigated. While the alignment has a positive effect on the throughput, it was mostly visible when using lower number of SPEs. Therefore, it is advised not to use the 128-byte alignment with small inter-SPE transfers to conserve local store space. Also using 128-byte alignment on external memory structures is only advised when the memory overhead remains small.

Additionally to the DMA benchmarks, several synchronization constructs were investigated. These constructs could be used to implement the synchronized access required in the 2D-Wave strategy, discussed in the next chapter. The fast mailbox had the lowest synchronization overhead with 0.86 us for an average round-trip from PPE-SPE-PPE

using 16 SPEs. This number represents latency with little to no contention. The real synchronization overhead depends on the average contention and depends on the application. However, it is expected that this quickly becomes the bottleneck when using more than 16 SPEs in fine-grained parallel application.

The conclusion drawn in this section form the base the implementation decisions discussed in Chapter 5. Also it provides the necessary insight for analyzing and discussing the experimental results presented in Chapter 7.

Parallel Strategies for H.264 Decoding

4

Until recently H.264 has been one of the most important drivers in the need of more compute capabilities. Todays state-of-the-art microprocessors are powerful enough to run all variants of H.264. However H.264 is an evolving standard and in the near future it is projected to scale dramatically to fulfill the need of higher levels of visual experience. Future applications like 3D-TV and Super-Hivision are essentially an evolution of the current H.264 standard. The projected computational need of these applications will be more than 10x higher. Advances in ILP and technology alone will not be able to provide the required compute capabilities. With the industry moving towards multi- and many-core architectures, the need for a scalable and higher level of parallelization seems clear.

In Section 2.3 several levels of parallelization in H.264 are reviewed. Current decoders implement only frame- and/or slice-level parallelism. For future many-core processors this level of parallelism is insufficient. To solve H.264 parallelization for many-core architectures, macroblock-level parallelism needs to be exploited. When applying macroblock-level parallelism the entropy decoding is not considered. This is expected to be done beforehand. However due to the presence of frame markers this can be done in a data parallel fashion on the frame level. In this work the parallelization of the entropy decoding is, therefore, not considered and we only focus on the macroblock decoding.

As revealed in Section 2.3.4, the potential amount of parallelism at the macroblock-level is huge. To exploit this two strategies are presented: Task Pool (TP) and the novel Ring-Line (RL) approach. The TP approach was first proposed by Van der Tol [26]. The natural implementation of the scheme is a traditional centralized worker-server model in which the work unit is a individual macroblock. The server in this model has to keep track of the dependencies and dynamically provides the workers with the work units. Since the macroblock-level parallelism is quite high it is expected that this scheme scales as long the synchronization overhead remains low.

In contrast to the centralized TP approach, RL incorporates a fully distributed control mechanism. In this strategy the processing element process entire scan lines. Intra data control signals flows from one processing element to the next. The processing elements can therefore be connected in a traditional ring network. On the algorithm level the RL approach will prove to be not as scalable as the TP strategy due to its static characteristics. However this strategy is still a very interesting one as it maps well on the Cell architecture.

With TP and RL only the spatial macroblock-level parallelism is exploited. Meenderinck et al. [18] also proposed their 3D-wave strategy, which combines spatial and temporal macroblock-level parallelism. This approach exhibits massive amounts of parallelism. RL can also exploit the temporal parallelism, albeit to a lesser extend. RL is only able to have up to two frames in flight. In this work we will not fully review

these variants of H.264 decoding. The reason for this is of a practical nature. For the implementation of the algorithms the open source FFmpeg code was used as a base. The FFmpeg program structure does not allow for intuitive implementation of any strategy needing multiple frames in flight. This requirement is a necessity for exploiting parallelism in the temporal domain. Adding this feature could go as far as rebuilding the entire program structure. Furthermore the number of processing elements in the target platform is limited to 16. Up to 16 cores the difference between 2D and 3D-wave is fairly small as will be revealed further in the chapter. For RL the temporal variant is included in the theoretical analysis. With this it is possible to project real performances. RL and single-frame RL (SFRL) denote the spatial MB-level RL variant. The combined spatial and temporal variant is denoted by multi-frame RL (MFRL).

In this chapter the TP and RL strategies are discussed in detail on the algorithm level. First, we start with TP followed by a theoretical scalability analysis. After this we discuss RL and its differences to TP. This is also followed by a theoretical scalability analysis.

4.1 Task Pool Approach

The TP approach exploits 2D-Wave parallelism, described in Section 2.3.4, in a worker-server fashion. Only the part after the entropy decoding is parallelized. The entropy decoding has to be done before the macroblock decoding starts. In this section the TP algorithm is explained. After this scalability is analyzed in a theoretical analysis.

4.1.1 Task Pool Algorithm

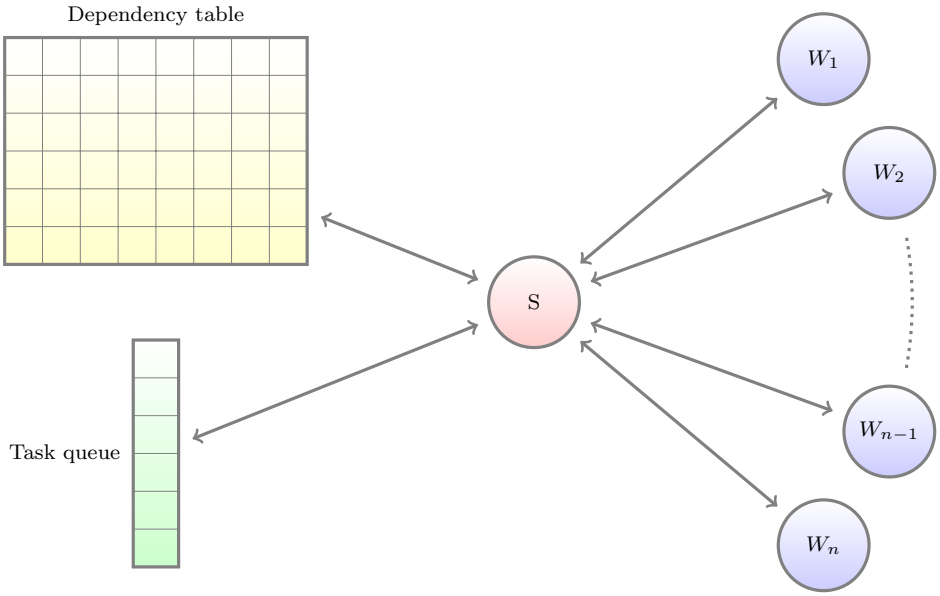


Figure 4.1: The TP algorithm is based on a worker-server model.

Figure 4.1 shows a high-level view of the TP implementation. From the figure clearly a worker-server model can be observed. The work unit in the TP algorithm is a macroblock. The server keeps track of the macroblock states and issues them to available workers when they are ready for processing. For this a dependency table and a task queue is used. The dependency table is a matrix with an entry for each macroblock. The entry contains the dependency count. The dependency counts are initialized with 0, 1 or 2 depending on the index. The top, left and right borders of the table are initialized to 1, while the rest initializes to 2. Only the entry with index (0,0) is initialized to 0. Having a dependency count of '0' means that the dependencies have been resolved and that the macroblock is ready to be processed. For each macroblock its corresponding dependency count reveals the number of macroblocks it depends on. Figure 4.2 shows the macroblock dependencies and dependency counts.

Figure 2.5, however, shows that each macroblock is dependent on the left, top left, top and top right macroblocks. This means that the number of dependencies is four and is consequently not correctly reflected in the dependency count. A more in depth look in the dependency structure reveals that the dependencies of the top left and top macroblock are redundant. On the left border there is no left macroblock. Likewise the top and right border do not have a top right macroblock. Therefore these have a dependency count of 1. The first macroblock (0,0) does not have dependencies at all and therefore is assigned a 0.

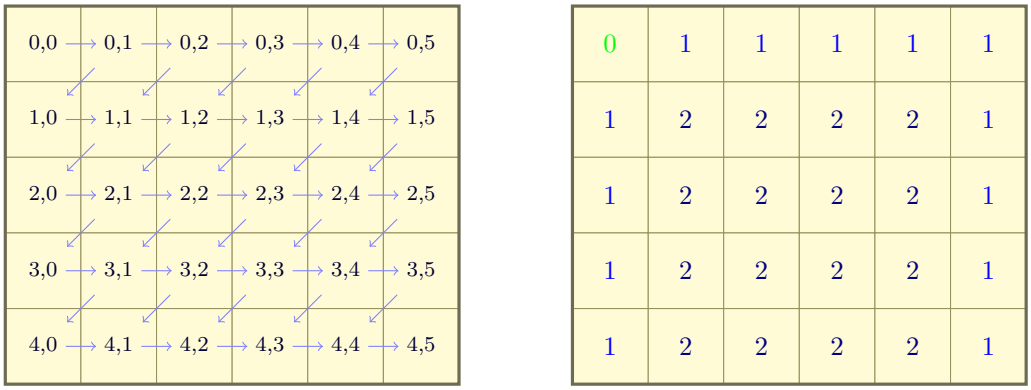


Figure 4.2: Left: Dependency flow. Right: Dependency counts.

When the dependency count is '0' the macroblock is appended to the task queue. The task queue can be described as a circular first-in first-out append/consume buffer, which contain the free macroblocks. The size of the task queue is equal to the maximum parallelism of 2D-wave defined by Equation (2.1), since the maximum parallelism is equal to maximum number of free macroblocks.

The algorithm can be described as follows. When there is work in the task queue and a worker is available the work units are assigned to the worker. When the work unit has been processed, the worker signals the server that its ready. The server first updates the dependency table by decrementing the right and down left entry corresponding to the processed work unit. If one or more of them become '0', they are appended to the task queue. Now the server will check again for available workers and work units and

assign the work units accordingly. This will repeat until all work is processed. The task queue is overwritten in a circular fashion.

The parallelism exhibited by this strategy is equal to the spatial macroblock-level parallelism. Figure 4.3 shows the parallel macroblocks as a function of the time with limited number of workers. The lines represents the parallelism of HD input. The graph shows that parallelism ramping is relatively less with low number of workers, indicating an efficiency loss with higher number of processing elements due to ramping stalls.

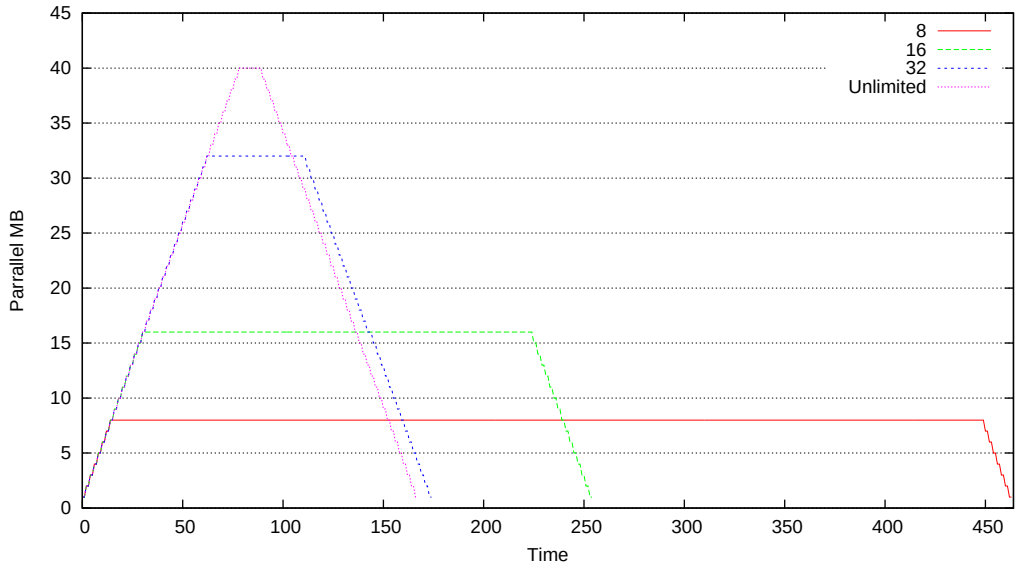


Figure 4.3: Parallelism ramping for a HD sequence with constant macroblock execution times.

In reality however, the macroblock execution times are far from constant. In H.264 the macroblock execution times are vary significantly. Figure 2.7 shows the possible deficiencies caused by ramping and dependency stalls. In the next section the impact on the scalability of the ramping and dependency stalls is investigated. The inefficiencies introduced by the synchronization overhead are not discussed as it a platform and implementation specific parameter.

4.1.2 Scalability Analysis - Task Pool

The TP algorithm follows 2D-Wave parallelism very closely and extracts as much parallelism as possible. However macroblock-level parallelism does not provide constant and infinite parallelism. Therefore, TP does not scale linearly, apart from implementation and platform inefficiencies. Inefficiencies are introduced by ramping and dependency stalls.

Ramping happens for every frame in the TP algorithm. This, however, can be avoided by extending the TP algorithm to exploit 3D-Wave parallelism. The ramping then only occurs once a video sequence, which is negligible. The efficiency loss due to dependency stalls, however, cannot be recovered. Therefore it is important to separate the

contribution of the individual sources to project theoretical performance of 3D-wave.

To investigate the inefficiencies at the algorithm level a TP simulator is used. The simulator does not simulate H.264, but rather the scheduling behavior of the TP algorithm. More specifically, the input of the simulator is not a H.264 stream, but the execution times of the individual macroblocks. The simulator uses the execution times calculate speedups for any number of workers. In the following experiments up to 64 processing elements are simulated. The input sequences have a length of 100 frames. The input resolutions are restricted to 1920x1080 and 3840x2160. In the remainder of the thesis these are referred to as FHD and QHD respectively.

To determine the impact of the dependency stalls, frame sequences with constant and variable execution times are simulated for each of the two resolutions. For the FHD resolution the variable MB execution times are extracted from three of the HD-VideoBench [4] sequences. The BlueSky, Pedestrian and RiverBed are 100 frames each. For variable QHD input Gaussian distributed random variables are used with a standard deviation of 5 and a mean of 20. On the FHD resoluiton these parameters provided comparable results with the HDVideoBench sequences. The speedup as a function of the number of workers is revealed in Figure 4.4.

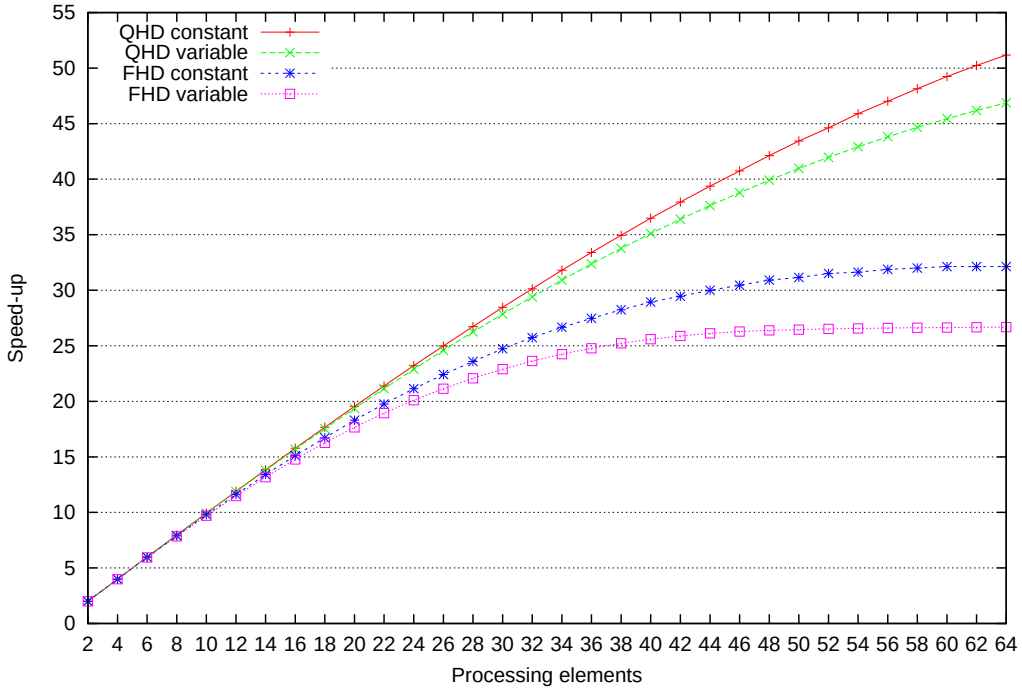


Figure 4.4: Scaling of sequences with constant and variable macroblock execution times.

From the figure can be seen that the QHD scales significantly better with larger number of processing elements. This is expected as the parallelism increases with the width of the resolution, which is in this case twice as large. The difference between the constant and variable variant of each resolution is due to dependency stalls. To

further point out the impact of variable execution times the normalized efficiency is plotted in Figure 4.5. The lines represent the efficiency of the variable execution times normalized to constant execution times. From the graph it can be seen that increasing resolution lowers the efficiency loss. This is expected since higher resolution provide more parallelism. The dynamic behavior of the algorithm counters the effect of variable MB execution times as long as there are other free macroblocks. Furthermore, the graph shows that the impact of the dependency stalls is not very significant. The lines plot stay above 90% efficiency most of the time, especially with small number of processing elements.

Figure 4.6 shows the normalized efficiency of constant macroblock execution times to perfect scaling. This efficiency loss is incurred by the ramping stalls. Since constant macroblock execution times are used the only the ramping stall remain. This is true until the number of processing elements hit the maximum parallelism defined by Equation (2.1). The additional efficiency loss introduced after this point is due to the lack of parallelism. Moving to 3D-wave resolves not only ramping inefficiencies, but also vastly increases parallelism. Up to 16 processing elements, however, the inefficiency losses are still small. Implementing the TP strategy on the Cell Blades gives a good representation of the 3D-Wave performance. The projected speedup of TP is 15x with 16 processing elements using a FHD sequence.

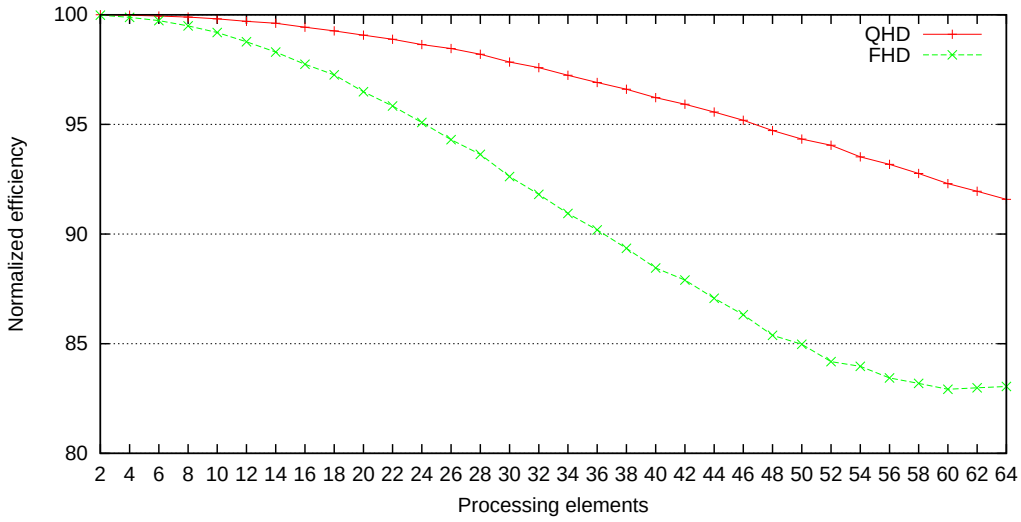


Figure 4.5: Normalized scaling efficiency of variable to constant macroblock execution times. The efficiency loss is caused by the dependency stalls.

4.2 Ring-Line Approach

In the previous section the TP approach has been discussed and analyzed. The TP algorithm exhibit excellent theoretical parallelism with a speedup of 15x with 16 processing elements. In this section the novel RL strategy is introduced. RL builds on a data flow principle and is expected to map well on the Cell architecture. First, the functionality of

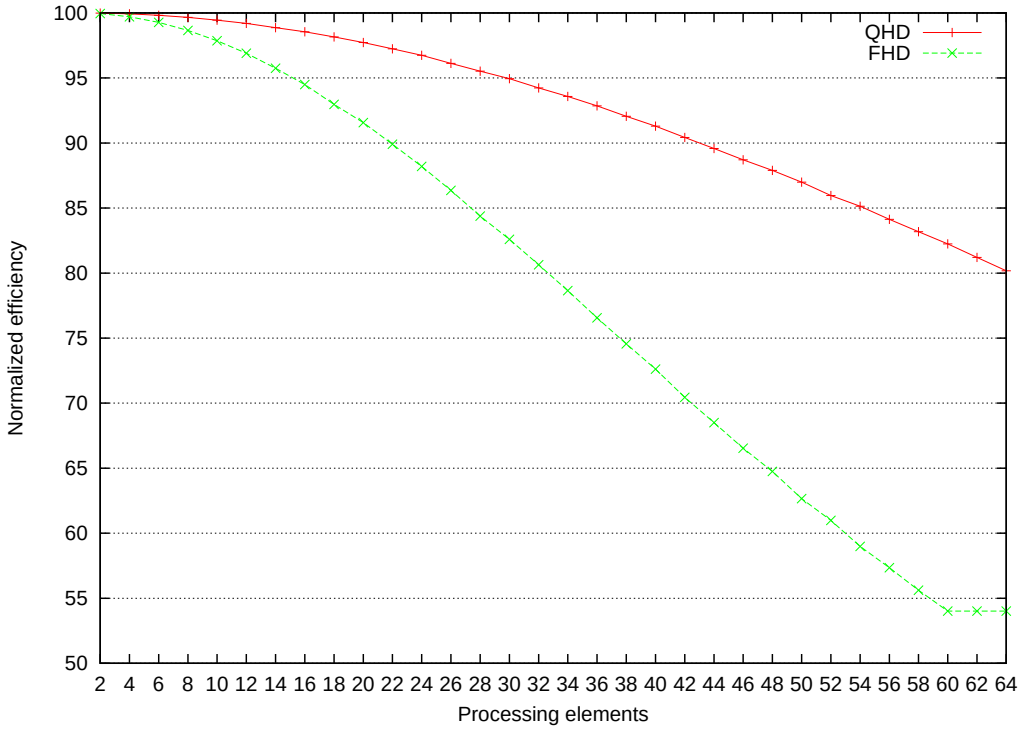


Figure 4.6: Normalized scaling efficiency of constant macroblock execution times to perfect scaling. The efficiency loss is caused by the ramping stalls.

the algorithm is explained. Following this the RL strategy is motivated and compared to TP. Finally, a theoretical analysis is presented to investigate the scalability in the same fashion as for TP.

4.2.1 Ring-Line Algorithm

In contrast to the TP algorithm, RL incorporates a fully distributed control mechanism. RL increases the granularity from individual macroblocks to macroblock lines, while maintaining the macroblock dependencies. Each processing element is assigned a scan line and processes the individual macroblocks in scan line order. Due to macroblock dependencies (Figure 2.6) the macroblocks cannot not be processed until all the dependencies are resolved. To satisfy this each processing elements has to communicate to its neighbor. Figure 4.7 illustrates four processing elements with each assigned a line. The arrows represent the dependencies. An important difference with TP is that instead of having two arrows out of each macroblock, only the one pointing to the next line requires synchronization actions. The arrow to the right has become implicit, since the macroblocks are processed in scan line order. The dependencies do not allow the processing elements to start execution until the previous processor has processed the depending macroblock. This can be satisfied by having each processing element communicate to its forward neighbor every time when it completes the execution of a macroblock. After the first line, processing element P_1 will continue with line five, the same way as the

second processing element will continue with line six. This continues for each processing elements until all the lines are processed.

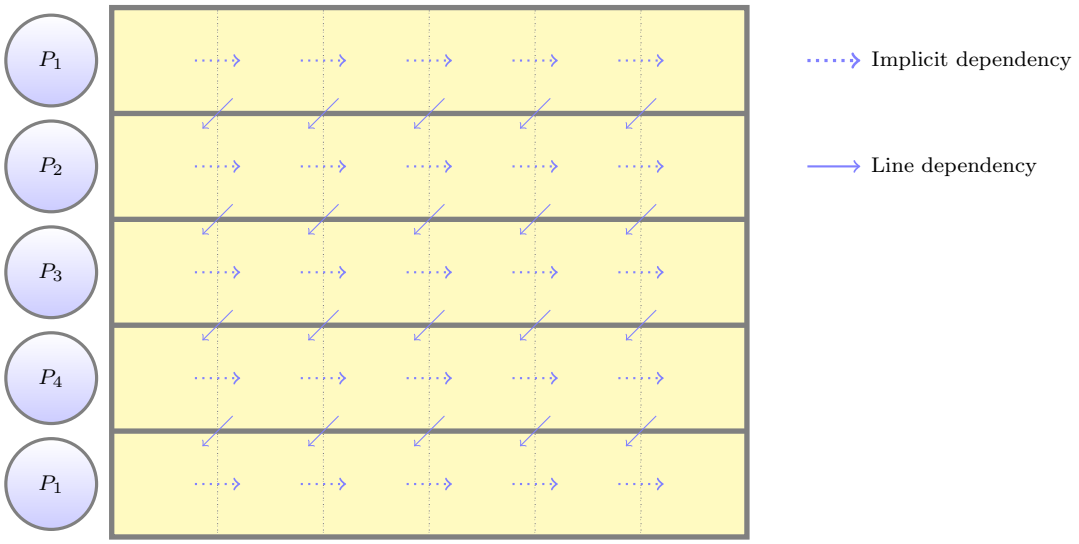


Figure 4.7: Simplified dependency flow of the RL algorithm. Dependencies flowing from one block to another on the same line are implicit.

From the figure only arrows to the next line are observed. Therefore, the processing elements only need to communicate with their forward neighbors. To allow this the processing elements can be mapped on a ring network. Figure 4.8 shows the mapping of a ring with the controller attached to one of the processing element. The C-processor is the *controller* and its task is to give start and stop signals to synchronize the macroblock processors with the entropy decoding. These signals only occur once for each slice/frame.

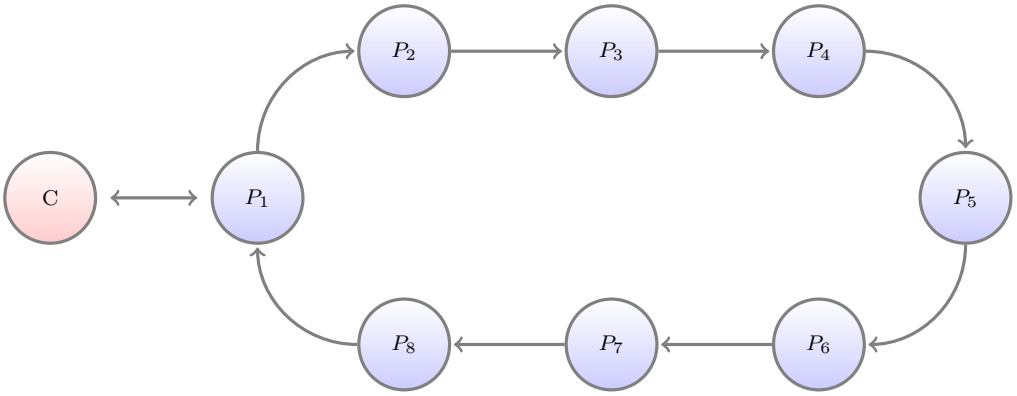


Figure 4.8: Uni-directional ring mapping of processing elements in the RL approach. The C-node is the control node which provides start and stop signals once a frame.

Since RL exploits 2D-Wave parallelism, it shows similar parallelism characteristics. The maximum parallelism is slightly different from Equation (2.1) and is described as:

$$ParMB_{max,RL} = min(\lfloor N_{MB,hor}/2 \rfloor, N_{MB,ver}) \tag{4.1}$$

When exploiting MB-level parallelism only one macroblock in a line can be processed at the same time. This also holds for RL as it assigns one processing element to a line. The only difference is that to avoid deadlocks the maximum parallelism has to reduced by one for uneven horizontal number of macroblocks, hence the floor operation.

Since it is based on macroblock parallelism, RL also exhibits ramping inefficiencies. Like in TP, temporal macroblock-level parallelism can be exploited to solve this inefficiency. In case of RL this is quite intuitive by viewing the input video as one large frame created by concatenating consecutive frames in vertical direction. Instead of stopping at the end of the frame and wait until all the processing elements are ready to move to the next frame, the processing element can directly continue with the next frame. Figure 4.9 illustrates this.

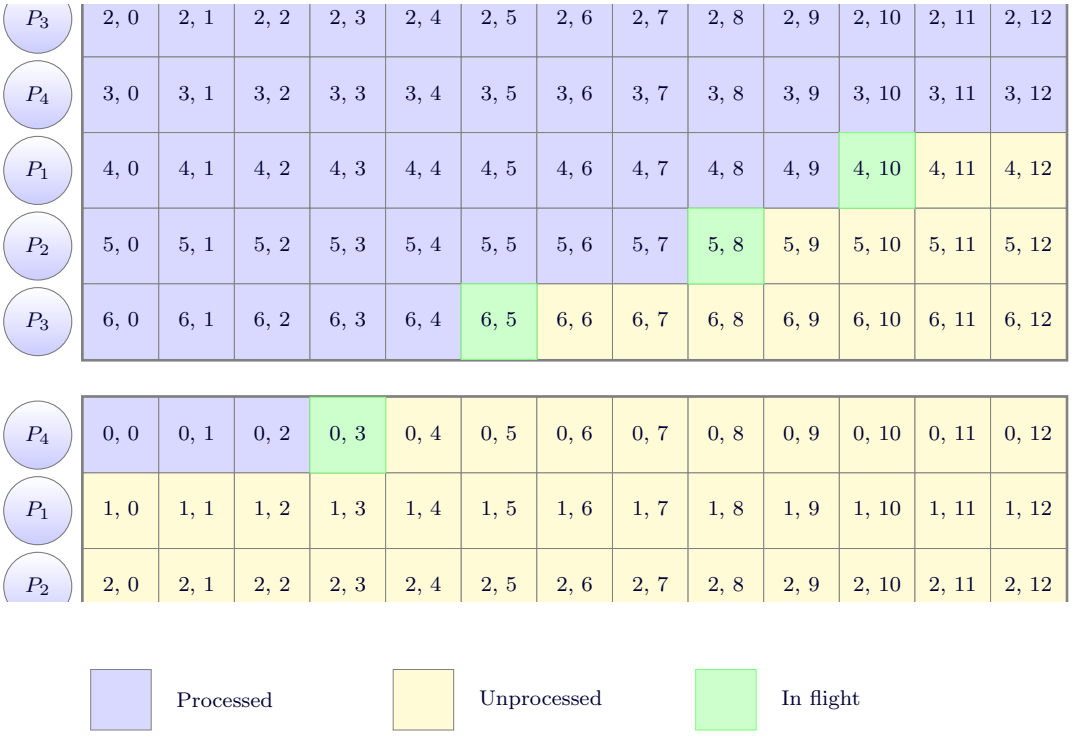


Figure 4.9: In multi-frame RL the decoding of the next frame start before the current frame end, which effectively negates the ramping stalls.

Looking at the figure shows that this approach has a maximum of two frames in flight for input sequences with a practical width to height ratio. This is quite different from what 3D-Wave allows in TP, which could have 200+ frames in flight with FHD resolutions. By using multi-frame RL the ramp-up and ramp-down can be neglected, since it now only occurs once in a video. The maximum parallelism, however, does not change. In exchange the average parallelism is now equal to the maximum. Even if this is not nearly as good as 3D-Wave, it still enables an increase up to two times depending

on the ramping characteristics of the input resolution. Furthermore only two frames in flight are required reducing the memory requirements and latency.

When moving to multi-frame RL the inter-frame dependencies also need to be considered. Since the H.264 standard defines a maximum motion vector of 512 pixels in vertical and 2048 pixels in horizontal direction, it is possible to violate the dependencies by having too many lines in flight. Most of the time the motion vectors are much smaller than this. However, to avoid screen corruption at all times the standard must be obeyed. To meet this condition some parallelism must be sacrificed. Since the motion vectors are a maximum of 512 pixels in vertical length, the maximum number of lines in flight can be defined as:

$$Lines_{max} = N_{MB,ver} - VMVL_{max}/16 \tag{4.2}$$

where VMVL is the vertical motion vector length. This will ensure that when moving to the next frame the top 32 lines are processed. Therefore, the macroblocks in the next frame always have there inter-frame dependencies resolved. However this does sacrifice parallelism. By combining Equation (4.1) with Equation (4.2) the maximum parallelism for multi-frame RL is found:

$$ParMB_{max,MFRL} = min(\lfloor N_{MB,hor}/2 \rfloor,\ N_{MB,ver} - VMVL_{max}/16) \tag{4.3}$$

In case of FHD inputs the maximum parallelism will drop from 60 to 36. In Section 4.2.3.2 a scalability graph will be presented regarding the scalability loss of taking inter-frame dependencies into account.

When looking at the dependency structure of RL it can be concluded that the processing elements cannot 'gain' on each other in horizontal direction. The processing elements cannot process more macroblocks than the processing element it is dependent on. As in TP the variable macroblock execution times lead to dependency stalls.

On the other hand, the distance between two processing can also get larger. This could pose a problem as a buffer is needed to store the intra data of the line directly above. If the distance between two processing element becomes larger more buffer space is needed. When memory constraint dictates a smaller buffer, buffer stalls could occur. Figure 4.10 illustrates these two situations. Processing element P_2 had a dependency stall at macroblock (1,9). P_1 has not processed (0,10) yet, which is required to satisfy the intra dependency. For the buffer stalls it is assumed that P_4 has a buffer size which fits intra data of 6 macroblocks. Because P_4 has not processed macroblock (3,1) yet, the intra data of macroblock (2,0) to (2,5) is still needed. This means that all the data in the buffer of P_4 is still needed for processing macroblocks. P_3 is not allowed to write the intra data of macroblock (2,6) and is stalled until a buffer slot comes free in P_4, which happens after P_4 processes macroblock (2,1).

Both TP and RL 2D-Wave parallelism. In contrast to TP, RL exhibits minor parallelism increase when extending to temporal macroblock-level parallelism. Furthermore the buffer and dependency stalls can be seen as additional drawbacks compared to TP. Taking these into account one might abandon further investigation of RL decoding.

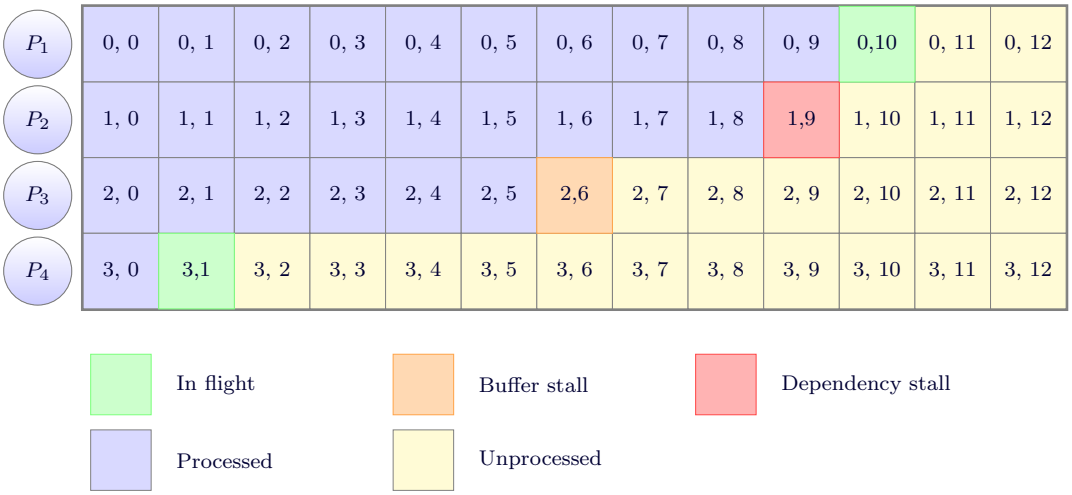


Figure 4.10: Dependency and buffer stalls in the RL algorithm.

However, RL potentially has several advantages. In the next section the advantages of RL are motivated. A scalability analysis follows afterwards, in which we investigate the impact of ramping, dependency and buffer stalls in detail.

4.2.2 Motivation

In terms of theoretical parallelism RL is inferior to TP due to buffer stalls and increased dependency stalls. Also when considering temporal macroblock-level parallelism, multi-frame RL is not as scalable as 3D-wave. However, it is well known that parallelism and (theoretical) scalability do not equal performance. The possible performance improvement lies in the distributed and scalable control mechanism, intuitive architecture mapping, and concurrent communication and computation.

4.2.2.1 Distributed and Scalable Control

The TP approach has a centralized control mechanism. The dependency table and task queue are both shared data structures which requires synchronized access. In Section 3.3.3 several possible Cell implementations for synchronized access to the shared data structures were investigated. It was concluded that none of these are highly scalable solutions. This is not surprising as centralized models inherently have limited scalability due to contention on the single synchronization point. Additionally MB-level parallelism is fine-grained with relatively small work units. The synchronization overhead could become the bottleneck with relatively low number of processing elements.

RL does not suffer from this as it features a distributed control mechanism. Also the control signals are one-way and non-blocking as no responses are necessary. The synchronization overhead stays constant and small independent of the number of processing elements. In TP the contention and ultimately congestion on the synchronized access of shared structures impacts efficiency. The constant and small overhead can be considered as a big advantage. However, RL deals with increased dependency stalls instead. It can

only process the macroblocks in the assigned line. It is interesting to see which of the two effects have more impact.

4.2.2.2 Architecture Mapping

As described in Section 2.1, the SPEs on the Cell are connected via the EIB, which can be described as a bi-directional ring. Combining this with the fact that each SPE has dedicated memory in form of the local store results in the ability to send the intra data control signals directly to the SPE. Being able to explicitly exploit data locality is one of the key feature of the Cell processor. In Section 3.3.1 and 3.3.2 it was found that on-chip bandwidth on the Cell was much larger than off-chip bandwidth.

The RL algorithm maps very well on the Cell memory hierarchy. The Cell offers explicit on-chip data management, which can be used to offload the external memory bandwidth. Compared to TP, RL requires less external memory bandwidth to do the same work. This also improves scalability in bandwidth constraint scenarios.

Because of the explicit memory management it is capable to pre-send the intra data directly to the target SPE. The memory latency is hidden since the DMA unit can do memory operations in parallel to the computation. In the next section we further discuss the use of the DMA unit and how this benefits RL.

4.2.2.3 Concurrent Computation and Communication

In the Cell processor the SPEs do not have a traditional cache. Instead this is replaced by a DMA unit and a local store. The SPEs communicate via their DMA units. As described in Section 2.1, the DMA unit can best be described as a memory co-processor. The DMA unit can move data from external memory to the local store and vice versa. It can also transfer data between local stores. The strength of the DMA unit is that it can process operations concurrent to the SPE. When properly exploited, the memory latency can be hidden behind the computation. To properly exploit the DMA unit, further examination of the application is required. More specifically, implicit memory accesses in H.264 must be uncovered.

Before a macroblock can be decoded, the output of the entropy decoding is required. This contains among others the DCT coefficients, macroblock types and subtypes, prediction coefficients and (arguable) the motion vectors. Also the intra data is needed for the intra-prediction and deblocking filter. Furthermore, motion data is required to perform the motion compensation. Finally, the resulting picture data has to be written back to the frame.

To create concurrent communication and computation behavior the required data must be predicted and requested before it is needed. While the programmer can do a much better job than the hardware there is a requirement that must be met. The algorithm must have a predictable nature.

After taking a better look at TP, we can conclude that it does not fall in this category. The specific macroblocks that are processed by each worker cannot be predicted at the algorithm level. This depends on the number of processing elements and the individual macroblock execution times, which cannot be determined at compile time. A static

schedule could solve this, but this impacts scalability and performance. Therefore, pre-buffering data on the macroblock-level is not possible. The DMA unit can also not be used to hide the latency of writing back data to the frame. The resulting picture data contains the intra data for processing other macroblocks. The data must be written back before resolving the dependency of these macroblocks. In short, the dynamic macroblock assigning of TP results in unpredictability. Therefore, it is not possible to exploit the Cell architecture in this regard.

In contrast, the RL algorithm is very predictable since the processing elements know which lines they need to process. This information can be used to pre-buffer the entropy output of the next macroblock. If the motion vectors are pre-calculated, the motion compensation reference data can also be pre-buffered. The intra data should already be present due to pre-sending of the previous processing element in the ring. Also the latency of writing back the resulting output to the frame can be hidden. The intra data is explicitly sent to the next processing element in the ring. It is not needed to request this from the shared memory as is the case for TP.

Figure 4.11 shows the memory request patterns. In this figure it is assumed that the work unit is a single macroblock. While it is possible to request a group of blocks [26] or multiple free macroblocks, it reduces parallelism.

For RL the memory accesses are concurrent to the computation. For TP this is not possible since it cannot predict the next work unit. The figure shows a much more efficient usage of the resources. In RL two times as many blocks are processed in the same time slots. However, keep in mind that the size of the blocks hold no value in terms of real communication and computation timings. In Chapter 6 the ratios are investigated further. Depending on the actual ratios the speedup exploiting concurrent communication and computation ranges from 1x to 2x.

4.2.3 Scalability Analysis - Ring-Line

In this section a scalability analysis is presented for the RL approach in the same fashion as for TP. Again a simulation program is written, this time to simulate the RL algorithm. The interface of the simulation program is fairly similar to the one used with TP. The input of the simulator is macroblock execution timings. The simulator calculates the resulting speedup with any number of processing elements. All the algorithm specific effects are taken into consideration. This includes the ramping, buffer, and dependency stalls. Additionally the simulator simulates performance of multi-frame RL.

As in TP the input sequence is again 100 frames in length. The number of processing elements ranges up to 64. The video resolutions used are FHD and QHD. For obtaining the variable execution times of FHD the HDVideoBench videos are used, while for the QHD Gaussian random variables are used. For RL a standard deviation (std) of 15 is used. This is a lot higher than the std of 5 used in TP. This is necessary because the results for the FHD with std 15 were similar to the one obtained using the variable macroblock execution times. The RL is relatively better at dealing with Gaussian like distributions than TP. The effects of the variable execution times are not felt if it does not cause dependency or buffer stalls. The standard deviation of a Gaussian distribution becomes lower when it is averaged with more samples. The H.264 macroblock execu-

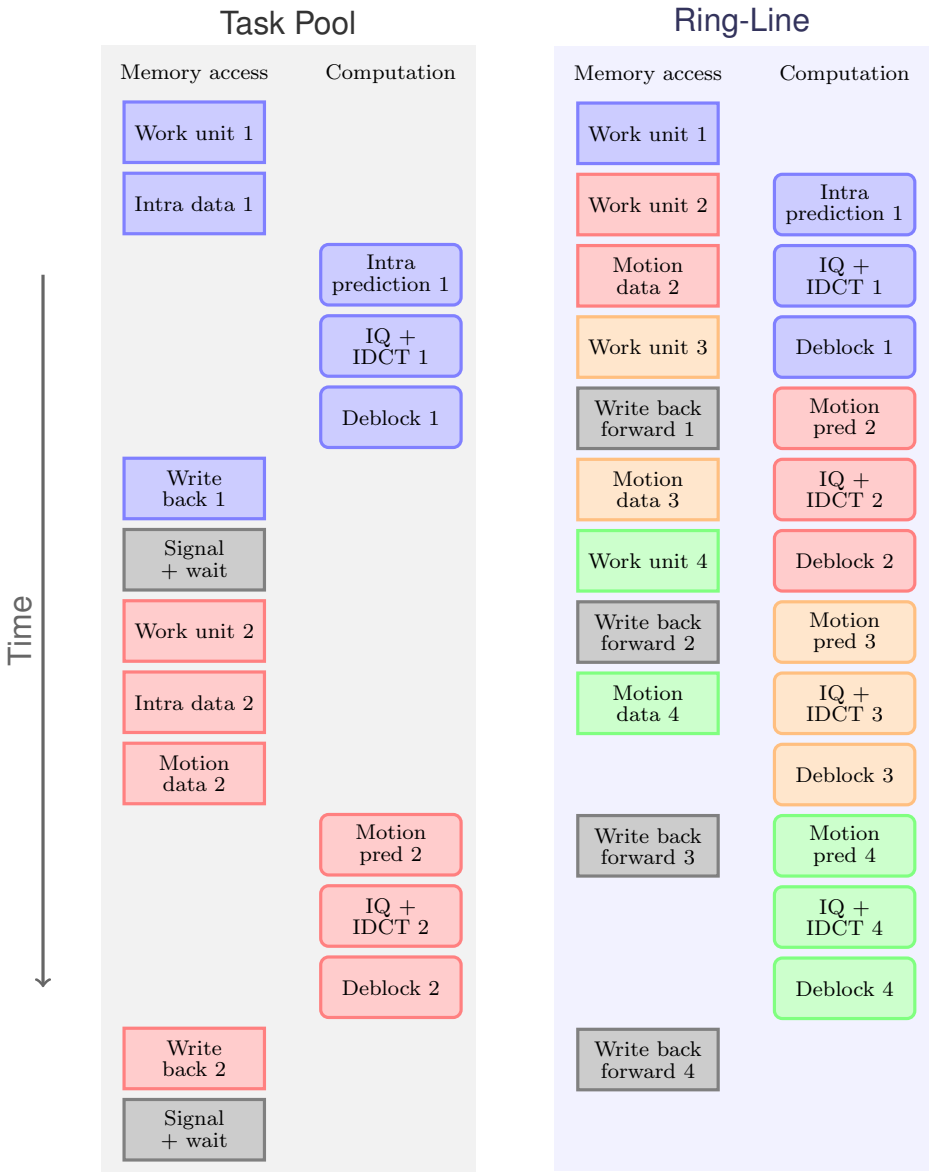


Figure 4.11: Difference in communication and computation patterns between Task Pool and Ring-Line.

tion times, however, do not match a Gaussian distribution as the variations are more clustered.

As in the scalability analysis of TP, the synchronization overhead is neglected. The purpose of the analysis is to determine platform independent limits of the RL strategy. The results presented should be interpreted as best case scenarios. However, we expect that the results obtained for RL will to be closer to reality than is the case for TP. Due to the distributed nature of the RL algorithm the synchronization latency remains constant and hidden behind computation.

In TP two sources of inefficiency were identified, the dependency and ramping stalls. However for RL a third source is present. Recall that in RL the intra data is pre-sent to the next processing element. When the buffer size is insufficient, the sending processing element has to wait until a slot opens up. This effect is referred to as a buffer stall. The analysis of the performance impact the buffer size will be preceded with an analysis of the theoretical lower and upper limit of the buffer size.

We start with the scalability impact of variable macroblock execution times. This is followed by the impact of ramping and finally the buffer stalls are discussed.

4.2.3.1 Dependency Stalls

The dependency stalls originate from variations in macroblock execution times. If all macroblocks have identical execution times the ramping is the only source of performance loss. However real videos the macroblocks have a lot of variation in execution time. Figure 4.4 shows that TP is not effected significantly due to its dynamic nature. It is expected that RL is affected more since it has a relatively static nature.

To investigate how much RL is affected by the dependency stalls the effect must be isolated. This can be done by comparing two variants of multi-frame RL (MFRL), one with constant and one with variable macroblock execution times. Since the simulation takes 100 frames long video sequences, the ramping of multi-frame RL can be neglected. The buffer width is also taken sufficiently large to avoid buffer stalls. The only remaining inefficiency is the caused by the dependency stalls.

Figure 4.12 shows the scaling of constant and variable multi-frame RL. Observing the plot reveals that the plot lines corresponding to input with constant macroblock execution times scale perfectly up to a point after which they do not scale anymore. Since with constant multi-frame RL no inefficiencies are present the only limitation is the macroblock-level parallelism described in Equation 4.1. As expected the lines stop scaling when the maximum parallelism equals the number of processing elements. The lines with variable macroblock execution times scale far from perfect. The FHD asymptote is about a factor 2x lower than its constant counterparts. It is expected that this also holds for QHD when using more than 64 processing elements.

A more accurate look at the efficiency loss is shown in Figure 4.13. The scalability with variable execution times is normalized to their constant counterpart. Since constant multi-frame RL exhibits perfect scaling the normalized efficiencies can be considered as real efficiencies, until they reach the maximum parallelism. To stay more than 90% efficient the processing elements needs to stay below 18 and 38 respectively for FHD and QHD. The cross-over points are about 1/3 of the maximum available parallelism and 1/6 of the horizontal macroblock resolution.

A way to view this is that the processing cores need a few macroblocks in between them to act as a buffer for the variations in execution times. When the buffer is too small the processing elements constantly bump in to each other. An analogy is to view the processing elements as cars on a road. The cars cannot catch up on each other and have the same average speed. However the cars do have a acceleration behavior that varies significantly. Imagine that the cars cannot be further away from each other than 10m. This will cause a lot of stalling to avoid bumping in the front car. If the cars would

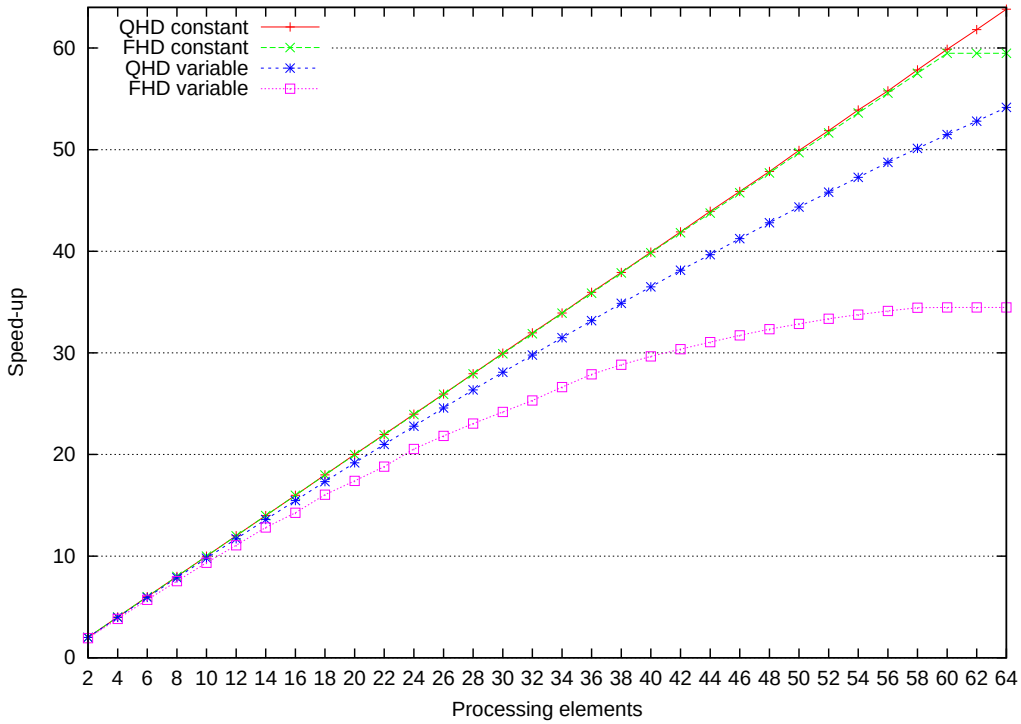


Figure 4.12: Scaling of MFRL with constant and variable macroblock execution times. The difference in scalability is caused by dependency stalls.

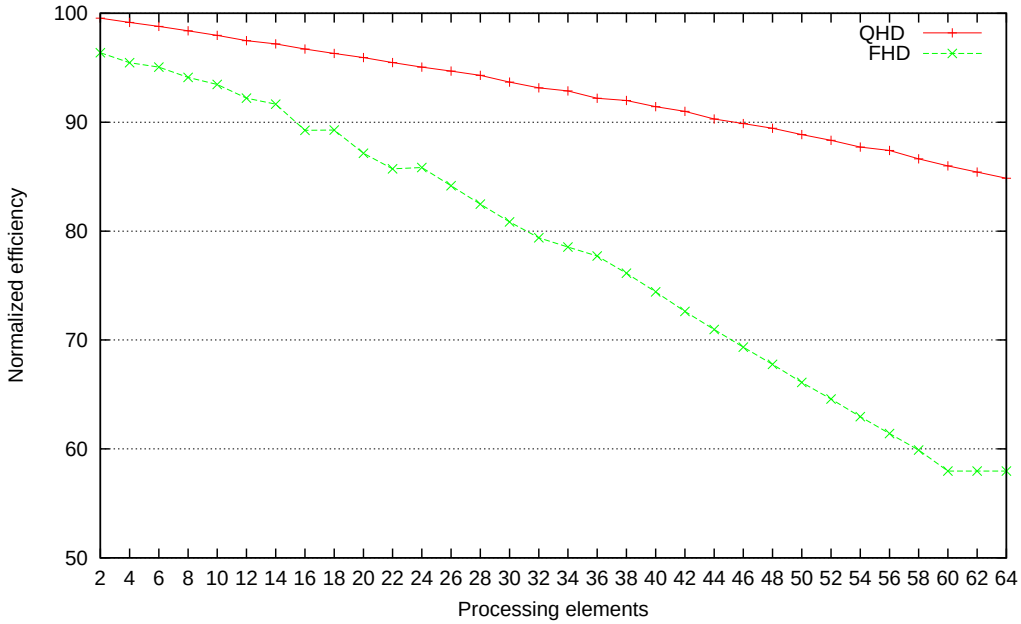


Figure 4.13: Normalized efficiency of sequences with variable to constant execution times using MFRL. The efficiency loss is caused by the dependency stalls.

be 100m apart this would become much less of an issue. In the case of RL a gap of six macroblocks will avoid most "bumping".

4.2.3.2 Ramping Stalls

To determine the impact of ramping the effect must be isolated. This can be accomplished by comparing multi-frame RL (MFRL) to single-frame RL (SFRL). The difference of these two simulations can be completely accounted to the ramping inefficiency. The impact of ramping is shown in a normalized efficiency graph of the SFRL and MFRL scalability in Figure 4.14. FHD is affected more than QHD, which is in line of the expectations since in the higher resolutions relatively more time is spent between the two ramping phases with the same number of processing elements.

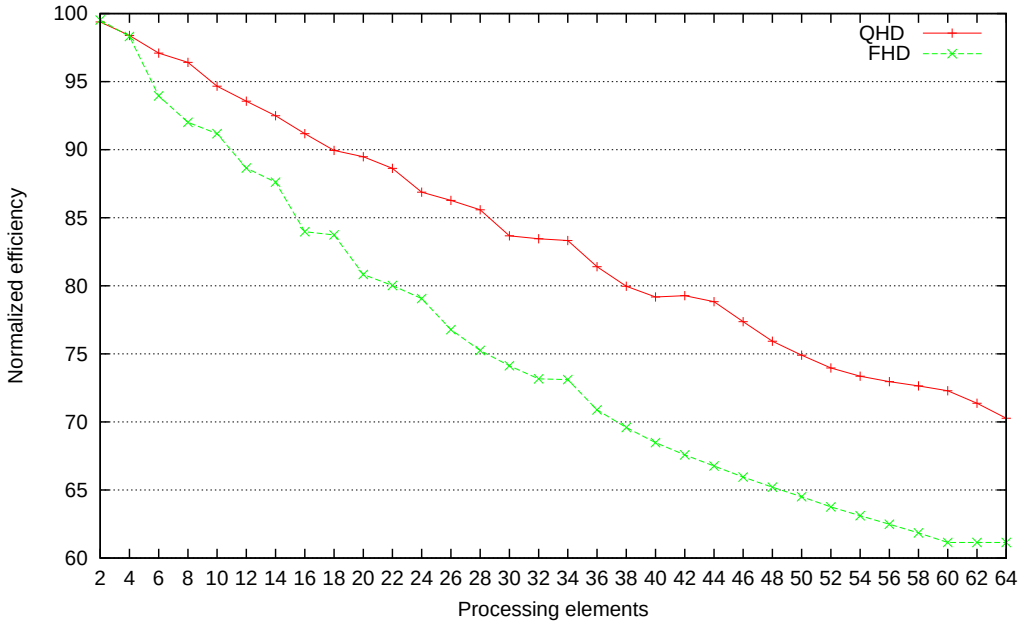


Figure 4.14: Normalized efficiency of SFRL to MFRL. The efficiency difference is caused by the ramping stalls

The efficiency drop is quite significant. To stay above 90% efficiency the number of processing elements should not exceed 10 and 18 respectively for FHD and QHD resolutions. It needs to be noted that this efficiency loss can be recovered by moving to MFRL. Implementing MFRL improves the efficiency significantly with higher number of processing elements.

4.2.3.3 Buffer Size and Buffer Stalls

In RL the intra data is sent from one processing element to the next. By keeping the data on-chip the off-chip bandwidth requirements are reduced. Furthermore the latency and synchronization overhead are also lowered considerably. However, the on-chip memory is very limited in size, which could pose as a problem. Optimizing the use of on-chip

memory is important. In embedded or application specific designs, using less memory directly impact the necessary die size. Also for the implementation of the Cell processor, which has a fixed on-chip memory size, it is important to know the requirements and the effects of the the buffer size. For this reason the buffer size requirement and its impact on the scalability are investigated.

To stay platform and implementation independent no hard memory requirements in byte quantities are provided. Instead the derived relations are defined in terms of buffer slots. A buffer slot can contain the dependency data of a single macroblock.

First let us consider the minimum and maximum buffer requirements. The maximum buffer size is defined as the number of buffer slots required to totally avoid buffer stalls. To determine this the cause of buffer stalls needs to be understood. Buffer stalls occur when the gap between the macroblocks is be larger than the buffer size. The gap is defined by the difference of the horizontal positions of the macroblock that is currently processed by two adjacent ring elements. When the gap reaches the limit of the buffer size, the buffer is filled with relevant data. When a macroblock is processed a buffer slot is freed and the gap grows smaller. Therefore, the buffer size determines the maximum allowed gap.

One of the implicit rules of RL is that the processing elements cannot catch up on each other in horizontal direction. From this can be concluded that the largest possible gap is always less or equal to the number of macroblocks in a scan line. The number of processing elements also affects the possible gap size. The minimum distance between processing elements is two macroblocks due to intra-dependencies. Therefore, for each processing element the maximum possible gap size is reduced by two. Putting this together results in the maximum buffer size:

$$BufferRL_{max} = N_{MB,hor} - 2(p-2) \tag{4.4}$$

, where p is the number of processing elements for $p >= 1$. For $p = 1$ the buffer needed is two slots larger than the MB_{width}. To process a macroblock, intra data of the top three blocks are required. The buffer needs to be able to support this and, therefore, the $Buffer_{max}$ is two slots larger than the maximum macroblock gap. Remember that the maximum buffer size is specified so that even the maximum possible macroblock gap does not cause a buffer stall. Therefore, applying the maximum buffer size fully avoids buffer stalls.

The minimum buffer size is also dependent on the same factors as the maximum buffer size. The relation between them differs however. The minimum buffer size is defined as the smallest buffer size to avoid deadlocking of the algorithm. Figure 4.15 illustrates deadlocking due to a too small buffer. The figure shows that all the buffer slots in all processing elements are occupied. Processing element P_4 cannot continue since P_1 has not yet processed $(3, 0)$ and $(3, 1)$. Because P_4 is stalled P_3 is also stalled. A stall chain can be observed up to P_1. When this occurs the algorithm is in a deadlock state.

There are two ways to solve this problem. First the buffer size of each processing element can be increased from four to five. Each processing element is able to move further apart and deadlocking is avoided. Another way to solve this is to introduce

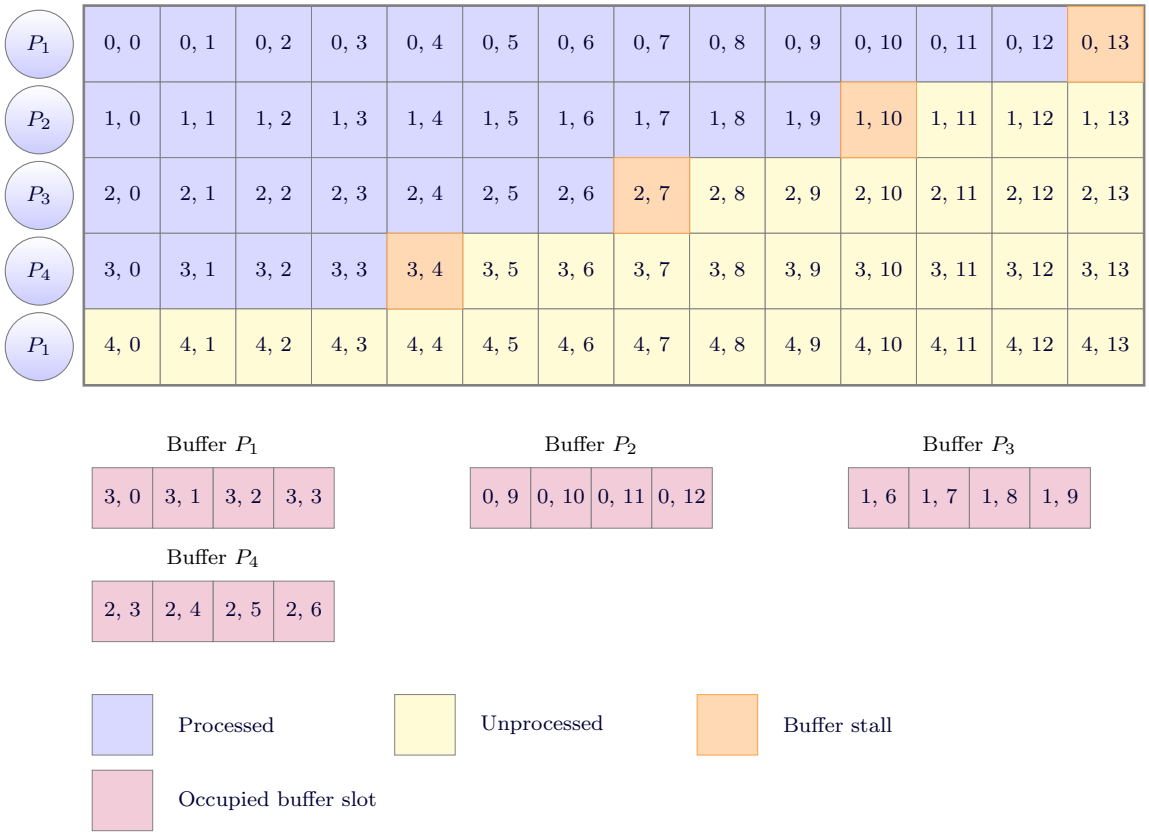


Figure 4.15: Deadlock incurred due to insufficient buffer size. All processing elements cannot write data to the next, since all the buffers are full.

another processing element, P_5. The additional processing element bridges the gap between P_4 and P_1, effectively avoiding deadlocks.

The common ground for the two solutions is that the total number of buffer slots increases. To avoid the deadlocks the total number of buffer slots must be enough to cover an entire line. Putting this together results in the minimum buffer size relation:

$$BufferRL_{min} = \lfloor N_{MB,hor}/p \rfloor + 2 \tag{4.5}$$

The +2 term is needed to ensure a large enough buffer to prevent overwriting of the top left and top macroblock dependency data.

Using the maximum and minimum buffer sizes both have their advantages and disadvantages. Using the maximum buffer size fully avoids buffer stalls. However, the amount of buffer slots may be too large to implement. For FHD this would mean having more than a hundred buffer slots. Furthermore, the buffer size needs to scale with horizontal resolution. On the other hand the minimum buffer size forces the macroblocks to be processing in semi-lockstep. The scalability and efficiency clearly suffer from this.

To form a hypothesis about the ideal buffer size, a closer look at the effects of the minimum buffer size on the algorithm behavior is required. When using the minimum buffer size the processing elements are forced to maintain a certain distance between

each other. Consequently dependency stalls cannot occur since the processing elements cannot bump into the forward processing element. Therefore, all the efficiency loss is incurred by the buffer stalls. This reveals a correlation between the efficiency loss incurred by buffer stalls and dependency stalls. Only the more dominant effect is visible in the efficiency loss.

In the situation of the minimum buffer size the buffer stalls are certainly dominant. And with the maximum buffer size the dependency stalls are dominant. We cannot improve on the dependency stalls since it is part of macroblock parallelism. Therefore, we only need to decrease the effect of the buffer stalls to the degree of the dependency stalls.

Imagine the minimum buffer size situation with three processing elements. The three processing elements are processing the macroblocks in lockstep and have an equal macroblock gap. A lot inefficiency is introduced since the execution time is equal to the maximum of the three macroblocks in flight. Now if the middle processing element would be able to move freely in the between the next and the previous processing element all the buffer stalls would be resolved for the middle processing element. Increasing the buffer size would not help since the processing elements cannot catch up. This effect can be realized by using a buffer size equal to Equation 4.6.

$$BufferRL_{hyp,opt} = \lfloor 2 * N_{MB,hor}/p \rfloor \tag{4.6}$$

In the hypothetical optimal buffer size is defined by two times the lockstep distance between two macroblocks. This buffer size is always larger or equal to the minimum buffer size. Also note that the attractive properties of the minimum buffer size are retained. The buffer sizes scale down when using more processing elements. Furthermore this also allows to scale the number of processing elements when higher resolutions are used without changing the buffer sized. With increases in resolutions additional computational capabilities are required. By increasing the number of processing elements the buffer size per processing element can remain the same, while addressing the compute requirements.

To put the hypothesis to the test a simulation is performed with the different buffer size relations. Among the them are the maximum, minimum and optimal buffer equations. Also an additional buffer size relation is simulated, which has a buffer size in between the minimum and optimal. To determine the effect of the buffer stalls the input needs to have a variable macroblock execution time. To take the ramping out of the equation multi-frame RL is used. Figure 4.16 shows the normalized efficiency of the buffer size relations to the maximum buffer size.

The plot should be interpreted as additional performance loss due to buffer stalls. Observing the plot reveals that the hypothetical optimal buffer size performs exactly the same as the maximum buffer size. The other two perform below optimal. Therefore, the hypothesis can be viewed as proven by experimental results. Furthermore, it can be seen that the three plot lines all converge to 100%. This is to be expected, since with increasing the number of processing elements the effect of the dependency stalls become more dominant to the point where the the macroblocks are processed in lock step. When this happens the effect of buffer and dependency stalls are equal and will result in equal performance. A more accurate way to view this is that at the crossover point at 60 processing elements all the buffer size relations become equal. The same explanation

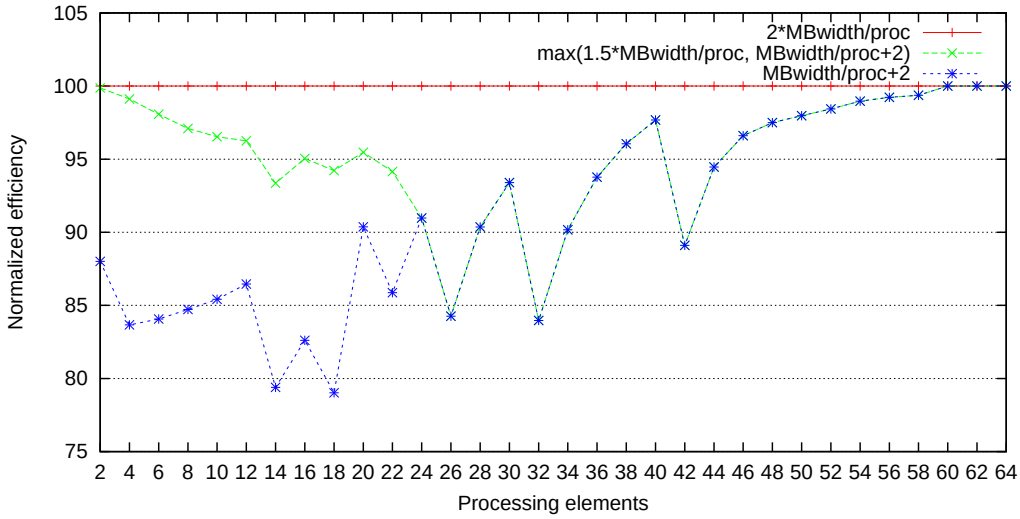


Figure 4.16: Normalized efficiency of several buffer size relations to the maximum buffer size relation of Equation (4.4).

can be given for the fact that at 24 processing elements the minimum and "in between" buffer size relations cross-over.

However, the figure also shows some irregular spiking of efficiencies for the minimum buffer size relation. The graph only presents results of even number of processing elements. If uneven results were to be included, the spiking would be more frequent. Also experimental results for the QHD resolution showed deeper downward spikes.

The spiking occurs because the total number of buffer slots is not monotonically increasing with the number of processing elements. Because the floor operator rounds down the number of buffer slots, the total number of slots could become lower. For example a downwards spike is observed at the transitions of 12 and 14 processing elements. The total number of processing elements is $(\lfloor 120/12 \rfloor + 2) * 12 = 144$ for 12 processing elements, while this is only $(\lfloor 120/14 \rfloor + 2) * 14 = 140$ for 14 processing elements.

With the minimum buffer size relation it is assumed that every processing element has the same amount of buffer slots. In a lot of cases the total number of buffer slots allocated is more than absolutely necessary, than when unequal buffer sizes are allowed for the processing elements. Having more redundant buffer slots improves performance. When the floor operation in Equation 4.5 performs a larger rounding, the performance is also lower.

4.3 Algorithms Compared

In previous sections the scalability of TP and RL were investigated. The focus lied on identifying the contributions of the ramping and dependency stall effects. For RL also the buffer stalls were included in the analysis. In this section the scalability of the two strategies are compared. In Figure 4.17 shows the scaling of the two strategies. For the FHD resolutions the same variable macroblock execution times are used as

previously. For the QHD resolution the Gaussian random variables are used with a standard deviation of 5 for TP and 15 for RL. As mentioned in the previous section these standard deviations are used because they provided similar results with the FHD resolutions. The mean is set to 20 in both cases. The buffer size for RL was set to the optimal buffer size relation of Equation (4.6).

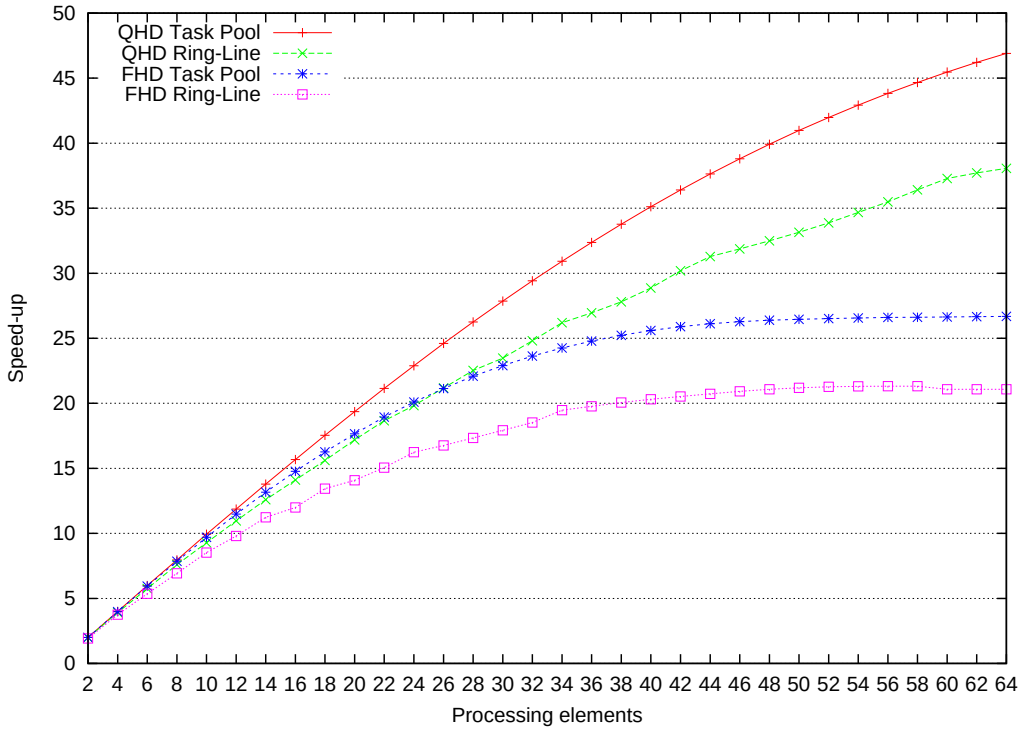


Figure 4.17: TP and RL scaling with variable macroblock execution times.

The results are in favor of TP. This is not surprising as in the previous section it was observed that due to the static behavior of RL it is has more dependency stalls. At 16 processing elements a speedup of 14.8x is observed for TP and 12x for RL using the FHD sequence. The maximum speedup is also in favor of TP with 26.7x to 21.1x. To have a better look at the efficiency behavior an efficiency plot is shown in Figure 4.18.

The efficiency quickly diminishes for RL. For FHD at 16 processing elements the efficiency is already at 75%.However a large part of the efficiency loss can be recovered by using multi-frame RL(MFRL). According to Figure 4.14 at the same number of processing elements the normalized efficiency of SFRL is at 84% of MFRL. A quick calculation gives us that with MFRL the efficiency would be 90%. This is still less than the efficiency of TP, however those results are only attainable with zero synchronization overhead.

4.4 Conclusions

In this chapter two parallel strategies for exploiting MB-level parallelism are discussed. The 2D-Wave parallelism introduced by Van der Tol [26] can be naturally implemented

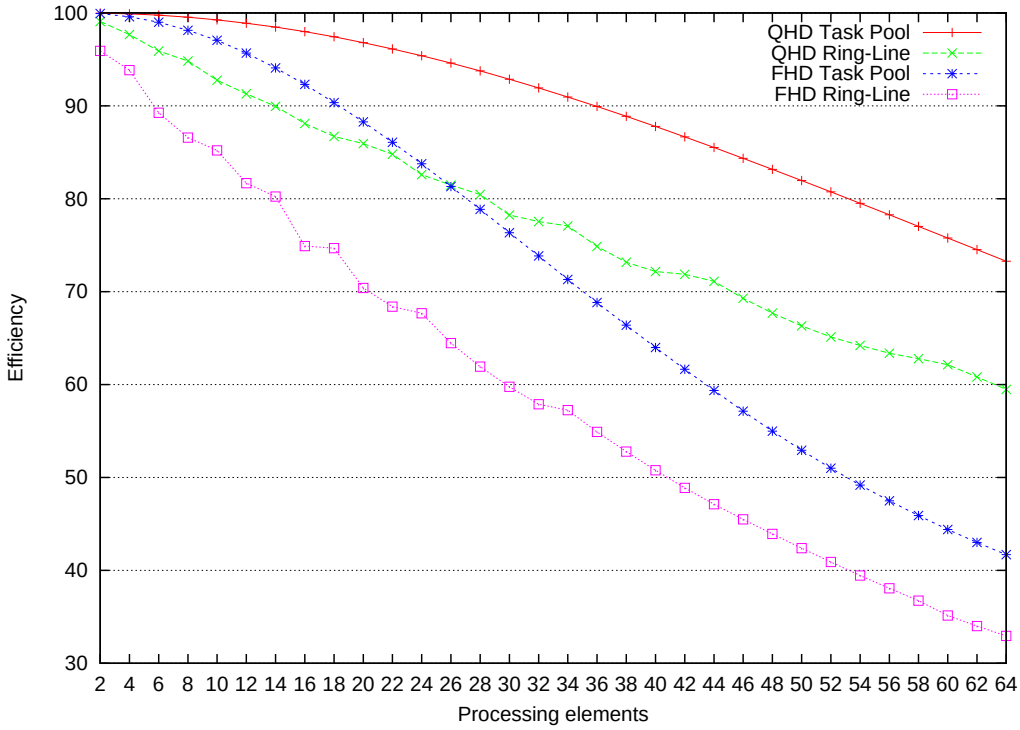


Figure 4.18: TP and RL efficiency with variable macroblock execution times.

as a worker-server model using the TP approach. The work units in this model are the macroblocks and the server synchronizes the access to the shared dependency table and task queue. The theoretical scalability for this approach is found to be excellent. With 16 processing elements a 14.8x speed-up is realized with FHD sequences.

In the novel RL approach, the processing elements are assigned entire scan lines instead of the individual macroblocks. It features a distributed control mechanism where only communication with the neighboring ring elements is necessary. Because of its more static nature the theoretical scalability is less attractive than TP. Only a 12x speedup is seen with 16 processing elements working on a FHD sequence. However, there are opportunities to map the RL approach efficiently on the Cell architecture. Due to its predictability concurrency in communication computation can be exploited on the Cell architecture by using the DMA units. Therefore, it is expected that the RL approach, if properly implemented, has a better performance per core. Furthermore the theoretical analysis performed in this Chapter has neglected the synchronization overhead, since this is a platform specific parameter. It is expected that the synchronization effects are more visible in TP, which will have a bigger impact on the performance and scalability.

To compare the theoretical scalability to the real performance, the two approaches have to be implemented. In the next chapter the implementation of TP and RL for the Cell architecture is presented. As stated before, the Cell architecture allows for an efficient mapping of the RL algorithm, therefore the actual performance may be higher despite showing less theoretical scalability. It is believed that future architectures will

also use a Cell-like memory hierarchy. Therefore, it has great value to use the Cell architecture for the implementation. In Chapter 6 the memory usage and bandwidth requirements of the implemented solution are investigated. Following this in Chapter 7 the performance results are presented and compared with the theoretical results revealed in this chapter.

Implementation of H.264 on the Cell Processor

5

Advances in ILP have slowed down and technology cannot scale performance further due to the power wall. Nowadays multi-core architectures are quite common and the focus has shifted to exploiting TLP. As Moore's Law continues to hold true, adding more cores is a logical step to effectively use the increasing transistor budget.

Modern x86 compatible multi-core architecture fall in the category of shared memory homogeneous design. The cores are identical and are general purpose in the widest sense. Programmers can easily create threads and have them execute among multiple cores, while maintaining all the advantages of the abstracted memory hierarchy. This approach, however, does not scale well.

With the move to multi-core, caches will increasingly pose problems. Since each core has individual layers of cache, coherency actions need to be performed in multi-core architectures. The complexity to perform the cache coherency actions with increasing number of cores is $O(n^2)$. The second problem is the memory bandwidth. In shared memory multi-cores the external memory is accessible for all cores. This also implies sharing the memory bandwidth. Thus, memory bandwidth should scale with the number of cores or it forms a bottleneck. In the near future it will be important to increase the data locality. The third problem applies to cache based architectures in general. Caches were introduced to reduce the average penalty of an external memory access. Cache misses nowadays can have a penalty of up to 300 cycles and is projected to increase further.

Instead of a regular multi-core platform, the Cell platform is chosen for implementing the parallel H.264 decoder. The Cell architecture solves the problems of modern x86 multi-core architectures by explicit control in memory traffic. The Cell processor has one PPE and eight SPEs. The PPE can be considered as a traditional general purpose processor, which features the traditional memory hierarchy abstraction. The SPEs can best be described as SIMD processing elements without a cache. This is replaced with a 256 kB local store and a DMA unit. In contrast to the PPE, the SPEs cannot implicitly access the shared memory. Instead the DMA unit has to be explicitly issued to move data to and from the external memory. The Cell memory hierarchy completely solves the cache coherency problems by having only one cache in the system. Since the memory management is done explicitly on the Cell, memory bandwidth can be conserved by keeping data on-chip. The DMA unit can be used to directly send data from one local store to another. Also the increasing effect of the memory latency is no longer an issue when using predictable algorithms. The downside is that the responsibility has shifted from the hardware to the programmer to find and exploit this predictability. The performance and efficiency of a Cell program depends more heavily on the programmer.

Due to its attractive properties, it is expected that future architectures will have a Cell-like memory hierarchy. Therefore, it has a lot of added value to implement the

H.264 decoder on the Cell platform. In the Chapter 4 the Task Pool (TP)and the novel Ring-Line (RL) strategy were discussed. The TP strategy is naturally implemented in a traditional worker-server model. In contrast, the novel RL strategy build on a data flow like principle. In this chapter we discuss the implementation of both strategies on the Cell platform. To avoid implementing the H.264 decoder from scratch the FFmpeg [12] open source code is used as a base. The FFmpeg code is widespread and used in almost every personal computer for decoding video.

The implementation discussed in this chapter supports the H.264 High Profile except for interlaced sequenmces. Both parallel decoders support level 4.x input video with the condition that CABAC is used for the entropy decoding. For research reasons it has no added value to implement all variants of H.264. Since H.264 supports a lot of settings and profiles not all videos decode with our implementation. Still, with High Profile - level 4.x - CABAC most videos are supported since it is the most common used profile and setting. Note that the High Profile does not support slices and therefore the FFmpeg base code has to handles these type of videos sequentially.

Before starting the discussion of the parallel implementations, an overview is given of the original FFmpeg code structure. This provides the necessary insight for implementing the parallel strategies. Afterwards we discuss the common changes to the FFmpeg structure needed for both parallel strategies. Finally the implementation of the parallel strategies is revealed starting with the TP followed by RL.

5.1 Original FFmpeg Code Structure

FFmpeg is a complete, cross-platform solution to record, convert, and stream audio and video. It includes the libavcodec audio/video codec library [12]. FFmpeg is the market leader for audio and video decoding on consumer PCs. In this section the FFmpeg code structure is presented. For obvious reasons we only focus on the video decoder part of FFmpeg. This includes the interface with libavcodec and the structure of the libavcodec H.264 decoder. The FFmpeg code is actively maintained by the community. The FFmpeg revision used in this thesis dates from May 2009 with revision number 19030.

5.1.1 Libavcodec Interface

The libavcodec library should be seen as the core of FFmpeg. The actual decoding and encoding is done by the codecs contained in this library. The task of FFmpeg is to parse the video container and call the appropriate codec in libavcodec for the video content. The interface of FFmpeg to the codecs in libavcodec consists of three functions. These functions are `decode_init`, `decode_frame` and `decode_end`.

The codec is initialized with the `decode_init` call. This initialization step consists, among others, of initialization of the lookup tables and allocation of the necessary memory structures. To decode a frame, the `decode_frame` function is called, supplied with a start marker of a frame. The `decode_frame` function returns the decoded frame as the result. To decode the entire video this function is called for each frame. After all frames

are decoded, the `decode_end` function is called to clean up the codec by freeing all the memory allocations made in the initialization.

The interface is quite straightforward and provides the necessary abstraction. However, there is another issue that requires attention. The codec needs to be initialized before the frames can be decoded. In FFmpeg the `decode_init` function is used for this, which has to be called before any `decode_frame`. In H.264, certain size information is needed to allocate the structures in the initialization which is embedded in the slice header. This poses a problem since the slice header is decoded in `decode_frame`, which can only be called after `decode_init`. FFmpeg and libavcodec solves this by doing a dummy decode run of the first frame, before the actual decoding.

The objective of the dummy decode is to extract the necessary slice parameters. In the following real decode the saved slice parameters are used to do a full decode. Instead of opting to update the interface, this (hacked) approach is most likely preferred for its compatibility. This is observed quite often in the software world, since the programmers are usually not very fond of recreating code, especially in programs as old and large as FFmpeg. Updating the interface would require changes to every codec in the entire libavcodec.

In our case the limitations of the libavcodec interface poses great difficulty in implementing 3D-Wave and multi-frame RL, discussed in Chapter 4. FFmpeg and libavcodec is a good example of code that is build without considering a parallel approach. The only way to decode the video stream is to call `decode_frame` for every frame in the stream. The interface of `decode_frame` dictates that it produces a frame as output. Since no other frame is able to start due to frame dependencies the number of frames in flight is limited to one. Changing the interface of FFmpeg and libavcodec requires such a quantity of work that it is wise to consider a complete restart. Since this is not a desirable solution, we decided not to implement 3D-Wave and multi-frame RL.

Because no changes are made to the interface, all changes are restricted to the libavcodec library. In the following section the structure of the H.264 decoder of libavcodec is analyzed in order to decide which parts of the code needs to be modified.

5.1.2 Libavcodec H.264

The libavcodec library consists of a large variety of audio and video codecs. An H.264 codec is also included in this library. In this section a simplified overview is provided of the libavcodec implementation of the H.264 decoder. After this the necessary changes are identified and grouped as common and/or strategy specific.

As stated before the interface of libavcodec is build up with a sequential mindset. This is also the case for the libavcodec H.264 decoder. Listing 5.1 is a pseudo code of the H.264 `decode_frame` function.

In libavcodec each codec has its own `Context`-type which holds all the shared information in a central structure. Each function in the libavcodec H.264 decoder operates on this structure. First the `decode_slice_header` function extracts the slice parameters from the H.264 stream and stores them in the `H264Context`. After this the macroblock processing starts. The macroblocks are processed one by one in scan line order. The processing of a macroblock has two steps. First, the `decode_mb_cabac` extracts the mac-

Listing 5.1: H.264 `decode_frame`

```
1 H264Context h;
2
3 decode_slice_header(h);
4
5 foreach line in frame{
6     foreach macroblock in line{
7         decode_mb_cabac(h);
8         decode_mb_internal(h);
9     }
10 }
```

roblock parameters from the stream and saves them in the `H264Context`. Second, the macroblock is decoded by applying the macroblock kernels. The kernels operate directly on the frame to avoid unnecessary copy operations. This scheme is quite straightforward and is fully coherent with the macroblock dependencies.

To be able to apply macroblock-level parallelism several changes must be made. Observe that `decode_mb_cabac` is hindering the parallelization. CABAC is a sequential operation that can only be parallelized on the frame/slice level. In this scheme every time `decode_mb_cabac` is called it extracts the parameters for one macroblock. However the macroblocks cannot be extracted in parallel since the data extracted from the stream is context-sensitive. In other words the interpretation of the extracted data is dependent on the history. Therefore the start and end points of the macroblocks are not known beforehand.

The first change, to enable macroblock-level parallelism, is decoupling the CABAC step from the decoding step. This requires the `decode_mb_cabac` to be moved outside of the loop to extract the data for each macroblock in the frame. The decoupling step needs to be performed before the macroblock processing and is necessary for strategies based on macroblock-level parallelism in general.

Second, on the Cell platform it is required that the code is split in two separate programs, one for the PPE and one for the SPEs. To implement macroblock-level parallelism the SPE program needs to perform the `decode_mb_internal` function, while the PPE handles the rest. To be able to do this on the SPE, the entire content of `decode_mb_internal` has to be ported. This mainly consists of the four macroblock kernels described in Section 2.2.

Programming the SPE is entirely different from programming the PPE. On the SPE there is no direct access to the external memory. Instead it has to operate on its private local store, which has a limited size of 256 kB. Furthermore the program image also resides in the local store space which leaves even less space for the data. Several steps can be identified in creating the SPE program.

The first step of porting the macroblock kernels to the SPE is to be able to make use of the SPE memory architecture. The libavcodec macroblock kernels operate on the entire frame. Also the motion compensation kernels can use up to 16 reference frames. Since the size of a single FHD frame is about 3 MB, it will be impossible to fit them in the local store. To solve this, the kernels need to be ported to work only the relevant

part of the frame.

The second step is to make use of the SIMD capabilities of the SPEs. To optimize in performance and possibly code size the kernels need to be vectorized. In memory limited cases, reducing the code size has the priority. FFmpeg already includes a vectorized implementation for the Altivec extensions of the powerPC. The SPEs have a very similar instruction set, which reduces the porting effort.

Finally, since the PPE and SPE program are separate programs, there is need for a communication interface/protocol. To create a suitable communication interface, several ways of inter-core synchronization were investigated in Section 3.3.3. It was observed that the combination of direct memory mapped mailbox functions and a polling PPE have the best latency characteristics when sending 32-bit messages. To make use of this in TP, the server role is assigned to the PPE. The PPE will initiate the processing of every macroblock by using the mailbox functions to message the SPEs. For RL the role of the PPE is different. The task of the PPE is to start the SPEs once each frame and wait for their completion. While it is still needed to setup an interface between the PPE and SPE, the focus shift to the more important inter-SPE interface.

Changes	Common	Specific
Decouple entropy decoding	X	
Port macroblock kernels	X	X
Define PPE-SPE interface		X

Table 5.1: Categorization of code changes.

In Table 5.1 the high level changes are summarized and categorized in Common and Specific. In the next section the implementation of the common part is discussed. In the two subsequent sections the strategy specific implementation is discussed for respectively TP and RL.

5.2 Common Changes to FFmpeg

In the previous section the code structure of FFmpeg is analyzed. The focus lied on the video decoding part of the program. This revealed that to apply macroblock-level parallelism first the entropy decoding has to be decoupled and performed beforehand on the frame level. The second step is to create a separate SPE program image for the parallel part of the strategies. In this section the common parts of implementing the TP and RL algorithms are discussed. First we start with the decoupling of the entropy decoding, followed by the general part of porting the macroblock kernels.

5.2.1 Decouple Entropy Decoding

In the original code the output of the entropy decoding is stored in the `H264Context`. The `H264Context` contains fields for the output of the CABAC step. However, these are overwritten every time a macroblock is decoded. This is not a problem for the original code since the CABAC data is directly used after the entropy decoding. It does hinder the decoupling in order to apply macroblock-level parallelism.

The decoupling of the entropy decoding builds loosely further on an implementation made by Alvarez et al. [2]. The implementation required several DMA transfers to be performed per macroblock. Transferring large structures in one DMA is much more efficient than several small ones. Furthermore, unnecessary information was transfered. In our implementation the output of the entropy decoding is grouped in two types of data structures: `H264mb` and `H264slice`. These structures are a both a subset of the fields located in `H264Context`. As the names suggest the `H264mb` contains the macroblock parameters and `H264slice` contains the slice parameters.

Each `H264mb` structure contains macroblock specific parameters of one macroblock. To save the data for an entire frame the same number of `H264mb` as the number of macroblocks in a frame are needed. This is allocated in an array of `H264mb` structures named `blocks`. To keep things coherent the array is indexed in the same order as the entropy data produces the output, which is scan line order.

The `H264slice` is not needed to save the slice parameters before they are overwritten, since new slice data only occur once a frame. The reason for using a `H264slice` is to save memory on the SPE. Alternatively, the entire `H264Context` could be copied to the local store. However, the size of this structure is at least 170kB and would not fit in the local store along with the program image.

Both `H264slice` and `H264mb` only contain the minimal needed subset of `H264Context`. These structures are needed on the SPEs. To keep the local store usage to a minimum only necessary information should be saved. After carefully making this selection a `H264slice` structure is around 12 kB in size and a `H264mb` is around 2 kB. A quick calculation shows that for a FHD frame around 15.6 MB of external memory additionally is needed. To fit the slice and a single macroblock data on the of local store 14 kB is needed. This is a big improvement over the 170kB of the H264Context.

The way the `H264slice` and `H264mb` are filled is by copying the necessary fields from the `H264Context`. By doing this at the right times, after `decode_slice_header` and each `decode_mb_cabac`, the right parameters are stored. This is done in this fashion to minimize the coding effort and to keep the adjustments as decoupled as possible. The downside is that the extra copy step introduces additional overhead. Alternatively, the output could be directly written to the corresponding `H264slice` and `H264mb` structures instead of the `H264Context`. This would increase the performance of the entropy decoding by about 10%. However, at this time we are not interested in the performance of the entropy decoding. Therefore the additional overhead was taken for granted.

5.2.2 Porting Macroblock Kernels - Generic

The generic part of porting the macroblock kernels to SPE code involves adjusting a portion of the code to handle the SPE memory hierarchy and the SIMD vectorizing. As stated before libavcodec has a vectorized implementation for the Altivec extensions. However, this applies only to the IDCT and motion compensation kernels. These two kernels can be relatively easy ported by using the SPE intrinsics, since there are a lot of similarities to the Altivec extensions. Alvarez et al. [2] provided the ported version although some small modifications were necessary. For the other two kernels, intra-prediction and deblocking filter, libavcodec does not provide a vectorized version. Instead

the regular scalar implementation is used as a base. Although vectorizing the remaining two kernels could improve performance [24][5], it is not the main goal of the research. For the same reason the vectorization is only briefly covered by two more statements. The first one is that vectorizing the motion compensation kernel was a necessary optimization in terms of code memory, while it increased performance. Second, attempts were made to implement the SIMDimized version of the deblocking filter provided by Azevedo [5]. However performance improvements were minimal while the code size actually increased with 6 kB. This is largely caused by the overhead involved in aligning the data for SIMDimized execution [4].

In the remainder of the section, the general impact of the SPE memory hierarchy on the implementation is discussed. First, the restrictions of the SPE memory hierarchy are analyzed to grasp the necessary modifications. Following this the solution to retrieving the motion data is revealed, which is the common part of the two SPE programs.

5.2.2.1 SPE memory hierarchy restrictions

As stated before the SPE does not have implicit access to the external memory. Memory requests have to be issued explicitly through a DMA unit. However, there are additional limitations imposed by the architecture. First the maximum size of the DMA transfer is 16 kB. In our case this is not a problem since our largest single DMA transfer is 12 kB for the `H264slice` structure.

Second, the maximum number of DMA accesses in flight is 16. When the DMA queue is full a DMA request stalls until a slot opens up. This could pose a problem when pre-buffering the motion data. In Section 2.2 the motion compensation kernel was discussed. This revealed that a macroblock could be divided in up to 16 sub-blocks. Each sub-block could need bi-weighted prediction. A quick calculation reveals that in the worst case 96 DMA operations per macroblock are needed. While this could pose as a performance problem, it remains to be seen how much these stalls actually occur.

Third, the DMA accesses have to be aligned to a 16-byte boundary. The micro-benches of Chapter 3.3 revealed that for optimal bandwidth characteristics the DMAs should be aligned to a 128-byte boundary. However, it is advised to only use this when the memory overhead is low. Furthermore, the size of the DMA requests has to a multiple of 16 bytes.

This last restriction requires some effort to work around. To avoid unnecessary memory copy actions in the local store all data structure should be aligned to a 16-byte boundary as much as possible. Furthermore, the inability to byte address the external memory leads to increased memory bandwidth requirements and local store requirements. The design decisions made in the TP and RL implementation are greatly impacted by this restriction. This is mostly discussed in the strategy-specific sections. In the next section we discuss retrieving the motion data, the common part for TP and RL.

5.2.2.2 DMA Access to Motion Reference Frames

As stated earlier the motion compensation kernel needs to operate on up to 16 sub-blocks. For each of the blocks a different piece of a motion reference data is required. Since a

reference frame is located in the external memory and is too large to fit in the local store entirely, only the relevant part of the frame should be brought in. The relevant part is the area pointed to by the motion vector. For the luma part of the reference block a border of two pixels above and left and three pixels below and right is needed additionally to the size of the sub-block. For the chroma components only a border of one pixel below and right is required. This translates to maximum block sizes of 21x21 and 9x9 pixels for luma and chroma respectively.

The width of the blocks are not multiples of 16. Also motion vectors point to individual pixels, which are mostly not aligned to a 16-byte boundary. This means that to bring in the data from the reference frame a certain overhead cannot be avoided. An example of the resulting DMA target of a reference block is shown in Figure 5.1.

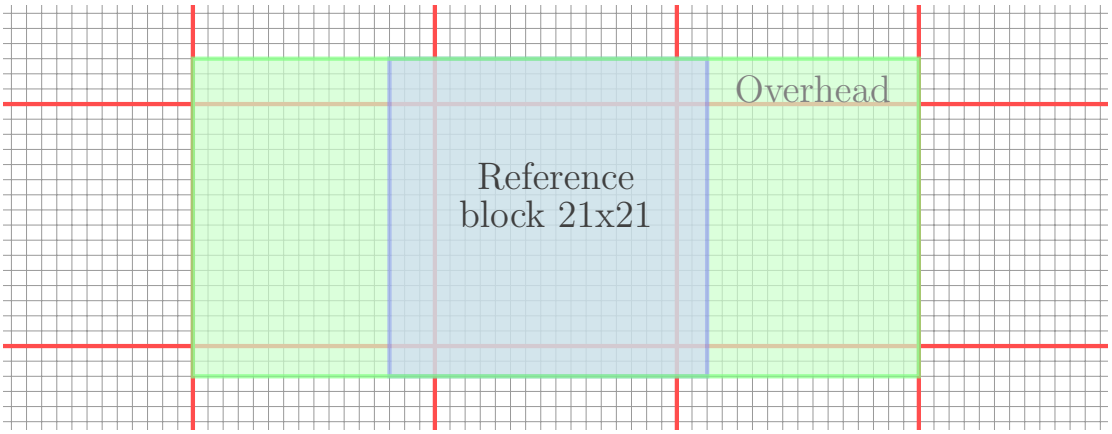


Figure 5.1: DMA overhead due to alignment restriction of a motion reference data block.

The figure shows a reference block of 21x21 pixels. Each pixel has the size of a byte. Since the width is 21 pixels the block at least crosses one 16-byte boundary. Our example shows it is able to cross two boundaries. The green area represents the overhead data due to the DMA alignment restrictions.

Another issue is that the block does not reside in continuous memory space. Each block line is part of a different scan line. Transferring this block to the local store would require 21 sequential DMA operations. Fortunately, Cell supports strided external memory access in form of the list DMA operations. A DMA list operation uses an array of global memory space addresses and its corresponding sizes. The local store target is a sequential memory space, the same as in the normal DMA operations. The maximum throughput of list operations is about five times lower compared to regular DMA operations, which is investigated in Section 3.3.2. However, it is still much more favored than using up to 21 DMA operations per block.

The libavcodec macroblock kernel functions are designed to operate on an entire frame. Since only a sub-block of the motion reference data is transfered from the frame, the kernel functions need to be modified as well. To keep the code complexity low the kernel functions for luma are modified to operate on a stride base of 48 pixels and for chroma on 32 pixels. This requires that every DMA operation for luma reference block has a width of 48. For chroma the width has to be 32. For luma the maximum reference

block size is 21x21, which means it can cross at most two 16-byte boundarys. For chroma the maximum size is 9x9 so it can only cross one boundary and hence only a width of 32 bytes is needed.

With this the SPE can perform motion compensation without having access to the entire reference frame. However there is a second issue that needs to be solved. In H.264 the motion vectors can point to somewhere outside the frame boundaries as long as the the reference block encapsulates at least one pixel of the frame. The rest of the block would then be filled by extending the edges of the actual frame date in vertical direction followed by an extension in the horizontal direction. In the original libavcodec this is handled by the `ff_emulated_edge` procedure.

FFmpeg has made an optimized by using a 16-byte border around the frame. The `ff_emulated_edge` is relatively costly procedure. With the extra border it is only required when the reference block crosses the extra border. In our SPE solution the same optimization can be used. However, the DMA transfer of the reference block needs an additional check. Figure 5.2 illustrates a case where it poses problems.

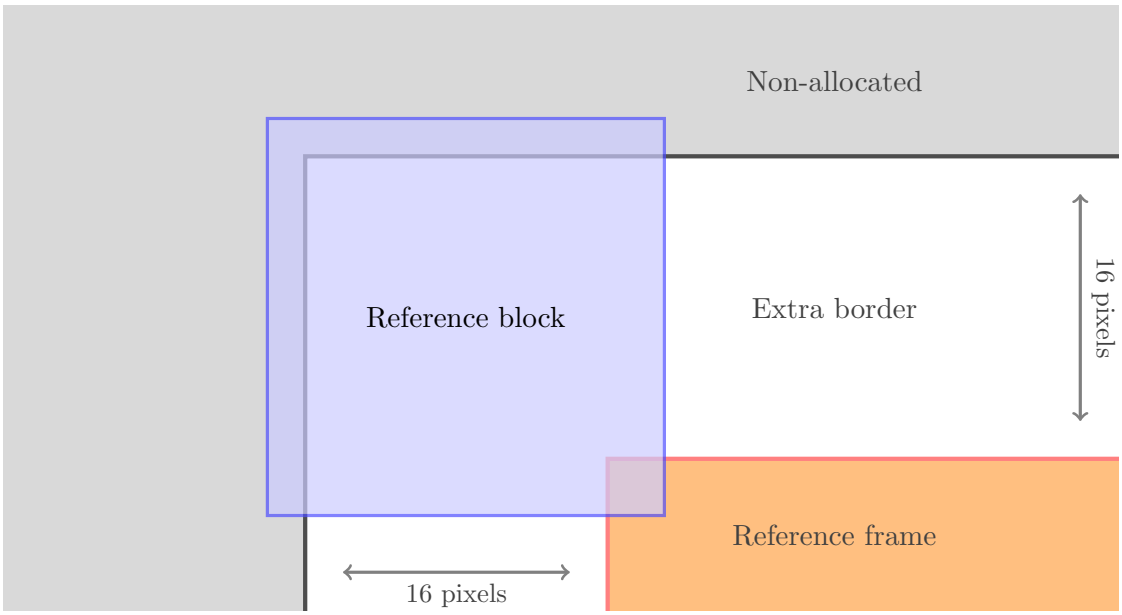


Figure 5.2: Motion vectors could point to a motion data block which encapsulates unallocated memory.

In the figure the reference block crosses the extra border. When this occurs the reference block resides partly in non-allocated frame memory. When trying to transfer this with a DMA the system might generate segmentation faults. To prevent this for these situations the starting points provided by the motion vector cannot be used. Instead the reference block must be moved back into real memory space. After this the pixels must be copied to their real position and extended the same way as in `ff_emulated_edge`. This also means an extra buffer needs to be reserved to support the extending. The copying and extending are all byte operations and cannot be vectorized due to the variable size. These cases ,however , do not occur often by using the extra border optimization on the

SPE. The average occurrence rate decreases even further when the resolution increases.

5.3 Task Pool Implementation

In the previous sections the common part of the parallelization is discussed. In this section the focus is on the TP implementation. The implementation of TP is discussed in four parts. The first part is the interface between the PPE and SPE. The second part considers the server side code that runs on the PPE. The task of the PPE is to provide synchronized access to the shared dependency table and task queue. The third part considers the implementation of the worker code on the SPE for processing the macroblocks. Due to alignment restriction the write back step required changes that impacts the parallelism. This is discusses in the final part.

Throughout the section we present simplified code fragments. The code fragments serve as a starting points for discussing the implementation decisions.

5.3.1 Interface Between PPE and SPE

The mailbox functions have the best latency characteristics for low volume (single 32-bit) PPE to SPE communication. Therefore, the interface is build on the mailbox functions. The interface has several requirements. First the interface should allow for the PPE to issue work to the SPEs. Second the interface should be able to handle the `H264mb` and `H264slice` structures. Finally, the ability to differentiate between types of workload is needed. For example, new slice parameters only occur once each frame and must be differentiated from a work unit. Also an exit signal must also be supported, which exits the SPE program.

In the chosen implementation some design rules are followed. The first one is to keep DMA transfer SPE initiated as much as possible. This increases concurrency and lowers the operations of the PPE. The second rule is to use as little as possible communication steps. Listing 5.2 presents a simplified version of the implemented SPE main loop.

In this code fragment the communication protocol is clearly visible. When the SPE image is loaded the first thing it does is to confirm the PPE that it has actually started. The SPE program then moves to a ready state. The ready state is simply a while loop in which is stays until the PPE sends the exit code. The `spu_read_in_mbox` function has the property to block until it has received mail from the PPE. Therefore no additional polling mechanism is needed. Mailbox functions can only send over 32bit messages. The best way to utilize this is to send over a `task_id`. The `task_id` can represent one of three things: the exit code, the slice task code or a macroblock ID. The macroblock IDs range from 0 to the number of macroblocks in a frame. Since the actual range depends on the resolution it is decided to use -2 for the exit code and -1 for slice code. The macroblock IDs represent the macroblock number in scan line order.

For this scheme to work two things need to be known. First is the location of `H264slice` and second the location of the `blocks` array of `H264mb`. The pointer to the `H264slice` is passed to the SPE program as an argument. The `blocks` pointer resides in the `H264slice` structure. All the SPE programs use the same `H264slice` and, therefore, have shared access to the `H264mb` array.

Listing 5.2: SPE main loop in TP.

```
1  H264Context_spu h_context;
2
3  int main(uint64_t slice_ea){
4      H264Context_spu* h= &h_context;
5      //send confirm to PPE
6      spu_write_out_mbox(1);
7
8      while (1) { //ready state
9          task_id = spu_read_in_mbox();
10         switch(task_id){
11             case -2:    //exit code
12                 return 0;
13             case -1:    //slice task id
14                 spu_dma_get(&h->slice, slice_ea);
15                 break;
16             case possible_mb_id:
17                 H264mb* nextmb= h->slice.blocks + task_id;
18                 spu_dma_get(&h->mb, nextmb, sizeof{h->mb}, ID_mb);
19                 wait_dma_id(ID_mb)
20                 hl_decode_mb_internal(h);
21                 break;
22         }
23         //send back the task id as confirmation
24         spu_write_out_mbox(task_id);
25     }
26 }
```

This results in an efficient implementation of the SPE program interface. The PPE only needs to use a single mailbox function to provide the SPE worker with a work unit. At the end of execution the work unit another one is needed to report back to the PPE.

Now let us discuss the server side interface and implementation of the TP algorithm. The code fragment of Listing 5.3 shows the simplified version of the server control loop. The first step is to fill the `H264slice` structure. This is done in `copy_context_to_slice` by copying the necessary field from `H264Context` to the `H264slice`. After this a slice signal is send to every SPE through a mailbox. Note that the status and task id is stored in the PPE program.

After this the actual control loop starts. This loop continues until all macroblocks are processed. The control loop contains an inner `for`-loop. In this loop each SPE mailbox is checked for a message. If a message is present first the status of the SPE is set to IDLE. After this the dependency table is updated if the message contained a macroblock ID and the number of macroblocks left is decremented. The second step is to check if a task and the SPE are available. If this is true a macroblock ID is popped from the task queue. This macroblock ID is send to the free SPE to start the processing and the status is set to BUSY. This continues until all the macroblocks are processed.

Using a while loop implies active waiting. Mostly this is not desired since it consumes a lot of resources. In this case we know from Section 3.3.3 this approach is faster than using mutexes or the callback function. Furthermore mailbox functions are blocking

Listing 5.3: The TP server control loop executed on the PPE.

```
1  //start all threads with the slice task id
2  copy_context_to_slice(slice_ea, h);
3  for (i=0; spe_threads; i++){
4      spe[i].status = BUSY;
5      spe[i].id = -1;
6      _spe_in_mbox_write(spe_mbox[i],task[i].id);
7  }
8  //start server control loop
9  while (mb_left){
10     for (i=0; i<spe_threads; i++){
11         if (task[i].status == BUSY){
12             //check if the spe has send the "ready' msg
13             if(spe_out_mbox_status(i)){
14                 int dataok= _spe_out_mbox_read(spe_mbox[i]);
15                 spe[i].status = IDLE;
16                 if (task[i].id>=0 && task[i].id==dataok){
17                     update_tq_and_deptable(spe[i].id);
18                     mb_left--;
19                 }
20             }
21         }
22         if (task_available() && spe[i].status == IDLE){
23             int mb_xy = get_mb_from_queue();
24             spe[i].status = BUSY;
25             spe[i].id = mb_xy;
26             _spe_in_mbox_write(spe_mbox[i], mb_xy);
27         }
28     }
29 }
```

calls. If no mail is present it will loop around until mail is received. Making use of the mailbox facilities always implies a polling procedure either way. Additionally tests show that the responsiveness does not suffer significantly when running another thread on the PPE. Also the performance degradation of the extra thread was neglectable.

However an interrupting mechanism is still preferred. This not only saves power, but makes the the code more comprehensible. The TP algorithm needs serialized access to the task queue and dependency table. The traditional interrupt system would be able to provide this. Nowadays this is only available in the embedded domain for the programmer. On the Cell, a software implementation of an interrupt is present in form of the callback function. A callback function is a function that is executed on the PPE, but called on the SPE. When a callback functions is called on the SPE the execution halts until the PPE has serviced it. In other words an interrupt. However in Section 3.3.3 it is discovered that this feature is horrendously slow and cannot be used when frequent synchronization is required.

5.3.2 Updating Dependency Table and Task Queue

In the control loop of Listing 5.3 the functions `update_tq_and_deptable` and `get_mb_from_queue` are used. These functions update the dependency table and the task queue of TP. These two shared structures require synchronized access. Since only the PPE thread uses these functions this requirement is met. If the workers would directly operate on these structures without synchronization the result might become erroneous. Elements in the dependency table and task queue could be modified at the same time, which results in unpredictable behavior.

A different approach to solve this is using atomic operations issued directly from the SPE. The atomic implementation requires six atomic operations to complete on average [14]. In Section 3.3.3 the latencies of the atomic operations were investigated and it was revealed that it was two to three times lower as a mailbox. However using six operations results in a higher latency than a mailbox round-trip. The number of operations required to complete the table and task queue update are very limited. Compared to the mailbox latencies this can be neglected.

A simplified version of the dependency table and task queue functions is presented in Listing 5.4.

In Listing 5.3 some abstractions were made to simplify the understanding of the scheme. For instance `init_task_queue` is never used. In this function the macroblock id 0 is submitted to the task queue, which is necessary to start the process. Another abstraction is that the task queue is not assigned a size. In Section 4.1 it was found that the size of the task queue was defined by Equation (2.1), which represents the maximum parallelism.

The functions in Listing 5.4 implement the dependency table and circular task queue of TP. In `update_tq_and_deptable` the dependency table entry to the right and down left are decremented. If the dependency count reaches 0 the macroblock ID is submitted to the task queue. The operations on the task queue are implemented in the functions `submit_mb_to_tq` and `get_mb_from_queue`. To know in which position the macroblock IDs need to be pushed and popped two index parameters are used. The parameter `tq_next_empty_slot` stores the index of the insertion slot of the next macroblock ID. Every time a macroblock is inserted this index is incremented. The same holds true for `tq_next_mb` which stores the next macroblock ID to process. The indexes are stored in the non-modulo form to be able to check for an available task in a single compare.

In circular buffer schemes one needs to take care of circular overwrites of valid entries. In the task queue this can happen in two directions since there are two indexes. Overwriting in the first direction is already prevented by using `task_available`. This ensures that `get_mb_from_queue` returns a legal macroblock ID. Overwrites in the second direction could happen if there were more tasks available than could fit in the task queue. This is prevented by taking a queue size of the maximum macroblock parallelism defined by Equation (2.1). The valid entries of the task queue are the entries with an unprocessed macroblock ID. Since the number of free macroblocks cannot exceed the maximum macroblock parallelism, it cannot overwrite the valid entries.

Listing 5.4: Dependency table and task queue functions.

```
1  int tq_next_empty_slot;
2  int tq_next_mb;
3  int tq_size;              //equal to max parallelism
4  int task_queue[tq_size];
5  int mb_width, mb_heigth;
6
7  void submit_mb_to_tq(int mb_xy){
8      task_queue[tq_next_empty_slot++ % tq_size] = mb_xy;
9  }
10
11 void init_task_queue(){
12     tq_next_mb= 0;                //index of next mb in tq
13     tq_next_empty_slot= 0;        //index next empty slot
14     submit_mb_to_tq(0);           //submit first mb to queue
15 }
16
17 int task_available(){
18     return (tq_next_mb < tq_next_empty_slot)? 1: 0;
19 }
20
21 int get_mb_from_queue(){
22     return task_queue[tq_next_mb++ % tq_size];
23 }
24
25 void update_tq_and_deptable(int mb){
26     int total_mb = mb_width*mb_height;
27     // check if the current mb has an mb on the right
28     if ( ((mb+1) % mb_width) != 0 ){
29         int mb_right = mb+ 1;
30         int dep_count = --dep_table[mb_right];
31         if (dep_count==0)
32             submit_mb_to_tq(mb_right);
33     }
34     // check if the current mb has a down left mb
35     if ( (mb%mb_width > 0) && (mb < (total_mb- mb_width)) ) {
36         int mb_down_left = mb- 1+ mb_width;
37         int dep_count = --dep_table[mb_down_left];
38         if (dep_count==0)
39             submit_mb_to_tq(mb_down_left);
40     }
41 }
```

5.3.3 Macroblock Processing

In the previous section the main program structure is discussed with Listings 5.2 and 5.3. The two code fragments implement the interface of the SPE program to the PPE. In this section we discuss the actual macroblock processing on the SPE. In Listing 5.2 the function `hl_decode_mb_internal` is called to perform the processing and the write back step. We primarily focus on the data flow from and to the local store. The macroblock kernels are only elaborated on when needed. Due to its size, the code of

`hl_decode_mb_internal` is split in four listings. Each listing contains a part of the simplified `hl_decode_mb_internal` function. The first three listings are discussed in this section. The last listing contains the write back step, which is discussed in the next section.

5.3.3.1 Transfer H.264 intra data

Before the kernels can be applied the intra data is needed. In the first part of the `hl_decode_mb_internal`, presented in Listing 5.5, the intra data is transfered by DMA operations. To do this, the start pointer to current block in the frame is calculated using the macroblock indexes. Note that `linesize` and `uvlinesize` are equal to the horizontal resolution of luma and chroma including the extra borders. After this the intra data is transferred using `get_frame_blk`. This function is a wrapper around `mfc_getl`, the DMA list function. `Get_frame_blk` creates the DMA list and issues the transfer.

The result of the transfer is a 48x20 block of the frame around the current block for the luma component and 32x10 for the chroma components. From Section 2.2 it is known that the intra data consists of two parts: unfiltered borders and deblocking filter data. The unfiltered borders are already on the local store in the `H264mb`. The deblocking filter data needs in case of luma, 4 vertical borders of the left macroblock and 4 horizontal borders of the top macroblock frame data. This could be transferred in a list DMA transfer of 32x20.

Transferring a block of size 48x20 block is done for optimization reasons. To avoid additional copy steps after the DMA transfer the DMA buffer is also used as a working buffer. This means that the kernels operate on it. However a working buffer must be able to include all the dependency data. From Section 2.2 it is known that a piece of the top-right lower border is used in the intra-prediction. Since the kernel functions can only work on a buffer with a constant stride the line width needs to be increased to 48 for the entire block.

The DMA list still has 20 list elements. The only difference is the width of each transfer. This does not impact performance much as in Section 3.3.2 it was showed that list element sizes up to 128 bytes have roughly the same latency. Figure 5.3 illustrates the reserved luma intra data.

For chroma the same principle is used. However the implementation differs due to the difference in size of chroma blocks and different intra data requirements. From Listing 5.5 we see that the size of the chroma transfers are 32x10 pixels. A chroma block is in our case 8x8 due to the 4:2:0 sub-sampling. This means that the transfer spans over four horizontal chroma blocks. From Section 2.2 we know that the chroma components are not depending on the top right macroblock. The kernels only operate on a coverage of two chroma blocks, which equates to a width of 16 pixels. The reason to use a buffer width of 32 is again due to alignment restrictions. Because the chroma blocks have a width of 8 pixels a pair of blocks is aligned on a 16-byte boundary only half of the time. In the other half of the cases a transfer width of 32 is necessary to successfully transfer the intra data. Figure 5.4 illustrates the two possible situations of the chroma DMA transfers. In the top situation a transfer with a block width of 16 suffices by only transferring the last two blocks. However, to satisfy the kernel functions

Listing 5.5: Macroblock processing part 1 - Get intra data

```
1  DECLARE_ALIGNED_16(uint8_t, dest_y_ls[48*20]);
2  DECLARE_ALIGNED_16(uint8_t, dest_cb_ls[32*10]);
3  DECLARE_ALIGNED_16(uint8_t, dest_cr_ls[32*10]);
4
5  void hl_decode_mb_internal(H264Context_spu *h){
6      H264slice *slice = &h->slice;
7      H264mb *mb = &h->mb;
8      const int mb_x= mb->mb_x;
9      const int mb_y= mb->mb_y;
10     const int mb_type= mb->mb_type;
11     uint8_t *dst_y, *dst_cb, *dst_cr;        //ea ptrs
12     uint8_t *align_cb, *align_cr;            //aligned ea ptrs
13     //ls ptrs (abstracts the fact it is operating on a ls buffer)
14     uint8_t *dest_y, *dest_cb, *dest_cr;
15
16     int linesize = slice->linesize;
17     int uvlinesize = slice->uvlinesize;
18     int stride_y = 48;
19     int stride_c = 32;
20     int i;
21
22     dst_y  = slice->dst_y + (mb_x + mb_y * linesize) * 16;
23     dst_cb = slice->dst_cb + (mb_x + mb_y * uvlinesize) * 8;
24     dst_cr = slice->dst_cr + (mb_x + mb_y * uvlinesize) * 8;
25
26     get_frame_blk(dest_y_ls, dst_y-4*linesize-16, stride_y, 20, linesize,
           ID_get);
27     dest_y = dest_y_ls + 4*stride_y +16;
28
29     align_cb = (uint8_t*) ((unsigned) dst_cb & 0xFFFFFFF0);
30     get_frame_blk(dest_cb_ls, align_cb-2*uvlinesize-16, stride_c, 10,
           uvlinesize, ID_get);
31     dest_cb = dest_cb_ls + ((unsigned) dst_cb&0xF) + 2*stride_c + 16;
32
33     align_cr = (uint8_t*) ((unsigned) dst_cr & 0xFFFFFFF0);
34     get_frame_blk(dest_cr_ls, align_cr-2*uvlinesize-16, stride_c, 10,
           uvlinesize, ID_get);
35     dest_cr = dest_cr_ls + ((unsigned) dst_cr&0xF)+ 2*stride_c + 16;
36
37     wait_dma_id(ID_get);
```

a total width of 32 is needed.

The `dest_y`, `dest_cb` and `dest_cr` pointers store the start position of the current macroblock. For the chroma pointers the pointers are not always the same. It will depend on what the current alignment situation is.

Before we can continue, a DMA wait request is issued to wait for all the transfers with DMA identifier `ID_get`. Using the DMA unit in this manor is not optimal. Remember that with the Cell memory architecture it is possible to solve the memory stalls introduced by cache misses. However, this cannot be implemented in TP in general.

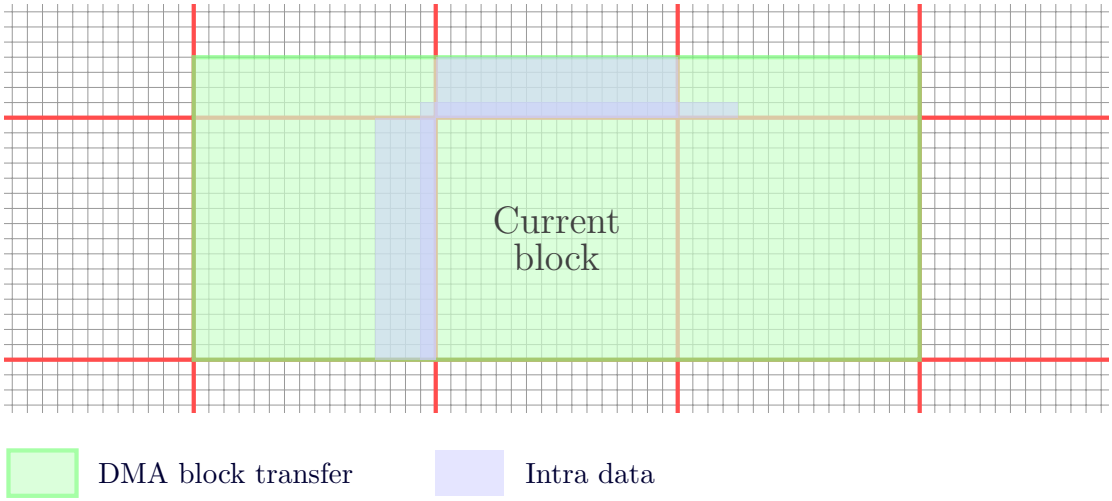


Figure 5.3: Intra data transfer of the luma component.

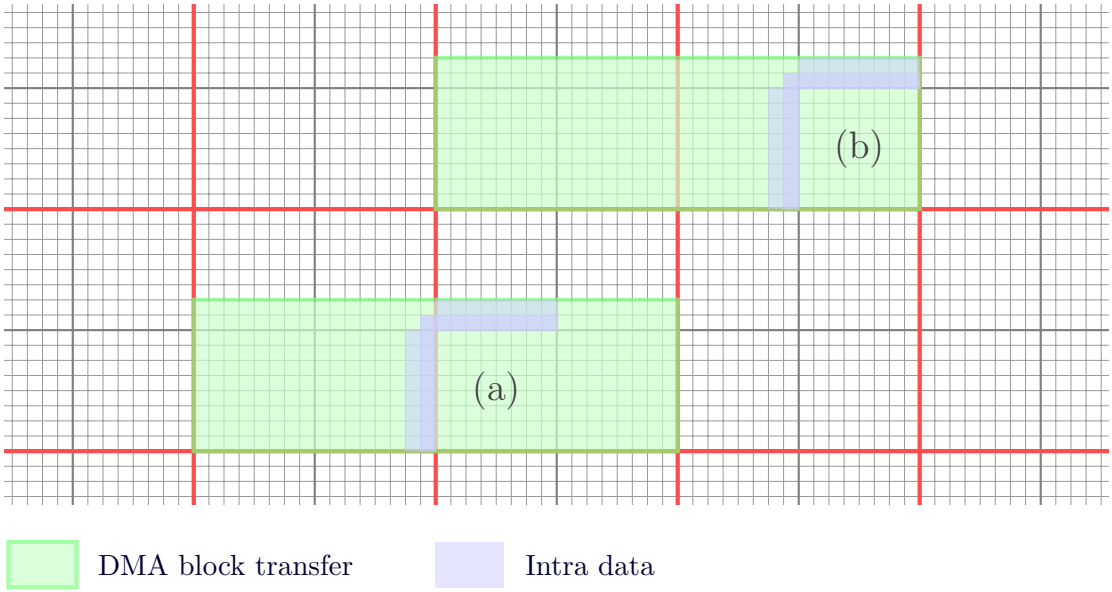


Figure 5.4: Intra data alignment issues of the chroma components. In situation (a) it is necessary to transfer 4 blocks, while for (b) this is done for consistency.

Using the DMA in this manor results in waiting for the DMA data instead of cache misses. We do ensure that the wait only occurs once, something that cannot be ensured for caches.

5.3.3.2 Macroblock processing

In the second part of `hl_decode_mb_internal` the actual processing starts. Listing 5.6 reveals that the intra-prediction or motion compensation kernel is applied depending on the macroblock type. In case of a intra-block the borders must be exchanged before

applying the prediction kernel. As mentioned before, the intra-prediction kernel works on the unfiltered borders. The borders are part of in the `H264mb` work unit.

There are two variants of the border exchange, which differentiate themselves in the last argument. In the first call an actual exchange is performed. The filtered borders in the working buffer are swapped with the unfiltered ones in `H264mb`. After the prediction step the filtered borders are put back. However the unfiltered borders are no longer needed. Instead of a swap the second `xchg_mb_border` call just replaces the borders as an optimization.

Listing 5.6: Macroblock processing part 2 - Border exchange and intra/motion prediction kernel.

```
1     if(IS_INTRA(mb_type)){
2         if(slice->deblocking_filter)
3             xchg_mb_border(h, dest_y, dest_cb, dest_cr, stride_y, stride_c,
                  1);
4
5         intra_prediction{h, dest_y, dest_cb, dest_cr, stride_y, stride_c};
6
7         if(slice->deblocking_filter)
8             xchg_mb_border(h, dest_y, dest_cb, dest_cr, stride_y, stride_c,
                  0);
9     }else {
10        hl_motion(h, dest_y, dest_cb, dest_cr, stride_y, stride_c);
11    }
```

If the current macroblock is not of the intra-type, the motion compensation kernel is applied. In Section 5.2.2 the SPE implementation for transferring motion data was discussed. In `hl_motion` this is used to transfer the motion data to the luma and chroma local store buffers. To support the maximum needed size, the luma buffers are of size 48x21 and the chroma buffers are of size 32x9. In total two luma buffers and 4 chroma buffers are needed to support the bi-weight prediction mode. For each partition of the motion compensation the buffers are reused.

Listing 5.7 shows the third part of `hl_decode_mb_internal`. In the third part the IQ, IDCT and deblocking filter kernels are applied. Only the chroma block requires inverse quantization. In the IDCT kernel the encoded variables are converted and applied to the local store buffers.

Recall that the unfiltered borders are located in `H264mb`. The code before `filter_mb` is responsible for storing the borders in unprocessed `H264mb` work units residing in the external memory. Before applying the deblocking filter kernel the macroblock borders are saved in `backup_mb_border`. After the intra-prediction step the unfiltered border fields are not used anymore. These fields are reused in `backup_mb_border` as a transfer buffer for the unfiltered borders. After this the border fields are transfered to the corresponding `H264mb` structures. This step ensures that each `H264mb` contains the unfiltered borders before it is processed. The right border is transfered to the right neighboring `H264mb` structure. The lower border is needed by the down and down-left `H264mb`. After issuing the border transfers the deblocking filter is applied.

Listing 5.7: Macroblock processing part 3 - IQ, IDCT and deblocking filter kernels.

```
1
2     IDCT_luma(h, dest_y, block_offset, stride_y);
3     IQ_chroma(h);
4     IDCT_chroma(h, dest_cb, dest_cr, block_offset, stride_c);
5
6     if(slice->deblocking_filter){
7         H264mb *mb_right, *mb_down, *mb_down_left;
8         int mb_width = slice->mb_width;
9         int mb_height = slice->mb_height;
10
11        // save unfiltered borders
12        backup_mb_border(h, dest_y, dest_cb, dest_cr, stride_y, stride_c);
13
14        // put unfiltered borders to the corresponding H264mb in external
              memory
15        if (mb_x+1 < mb_width){
16            mb_right = slice->blocks +mb_x +mb_y*mb_width +1;
17            spu_dma_put(mb->left_border, mb_right->left_border, sizeof(mb->
                  left_border), ID_put);
18        }
19        if (mb_y < mb_height -1){
20            mb_down = slice->blocks + mb_x +(mb_y+1)*mb_width;
21            spu_dma_put(mb->top_border, mb_down->top_border, sizeof(mb->
                  top_border), ID_put);
22            if(mb_x>0){
23                mb_down_left = mb_down -1;
24                spu_dma_put(mb->top_border, mb_down_left->top_border_next,
                      sizeof(mb->top_border_next), ID_put);
25            }
26        }
27        filter_mb( h, mb_x, mb_y, dest_y, dest_cb, dest_cr, stride_y,
              stride_c);
28    }
```

Note that we do not issue a `wait_dma_id`. As an optimization this is combined in a single wait statement in the write back step. This is discussed in the next section.

5.3.4 Write Back and Impact on Scalability

After the macroblock kernels are applied the working buffer contains picture data that needs to be written back to the frame. In the write back step the working buffers are written back to the frame. However due to the width of the working buffers write overlaps could occur when using multiple SPEs. The solution does not only involve the SPE code, as changes in the macroblock dependency structure are required. The latter also has its consequences on the scalability.

First, the write back step on the SPE is discussed. Second, the problem of overlapping write backs is revealed and the implemented solution is presented. Finally the impact on the scalability is analyzed in a similar fashion as in Chapter 4.

5.3.4.1 Write Back Step

In the write back step the content of the luma and chroma working buffers is transferred back to the appropriate location in the frame. From Section 2.2 it is known that the deblocking filter does not only change the contents of the current macroblock, but also the surrounding borders. In case of luma, this applies to the lower three lines of the top macroblock and the right three lines of the left macroblock. The same holds for chroma, only in this case it applies to a single adjacent line. Therefore, the entire working buffer is transferred back except the first line. Listing 5.8 shows the write back step.

Listing 5.8: Macroblock processing part 4 - Write back step TP.

```
1     put_frame_blk(dest_y_ls + stride_y, dst_y-3*linesize-16, stride_y, 19,
          linesize);
2
3     if(mb_x%2 >0){
4         reduce_stride(dest_cb_ls + stride_c, 9);
5         put_frame_blk(dest_cb_ls + stride_c, align_cb- uvlinesize, 16, 9,
              uvlinesize);
6     }else{
7         put_frame_blk(dest_cb_ls + stride_c, align_cb- uvlinesize-16,
              stride_c, 9, uvlinesize);
8     }
9
10    if(mb_x%2 >0){
11        reduce_stride(dest_cr_ls + stride_c, 9);
12        put_frame_blk(dest_cr_ls + stride_c, align_cr- uvlinesize, 16, 9,
              uvlinesize);
13    }else{
14        put_frame_blk(dest_cr_ls + stride_c, align_cr- uvlinesize-16,
              stride_c, 9, uvlinesize);
15    }
16
17    wait_dma_id(ID_put);
18 }
```

The write back is quite similar to retrieving the intra data, described in Section 5.3.3.1. In the write back step also three list DMA calls are issued and the same local store and global memory addresses are used. The difference is in the additional `reduce_stride` procedure issued before transferring the chroma blocks. This is part of the solution for solving overlaps in the write back step. As the function name suggests, the stride of the chroma blocks is reduced. The stride is reduced from 32 to 16 to have a less wide block to prevent the overlap. From Figure 5.4 it can be seen that this is only possible when the current macroblock has an odd x-coordinate, which is the case in the upper block. In the next section this problem is presented in more detail.

Before returning a `wait_dma_id` is issued to wait for all the transfers to complete. After the write back step the SPE sends a mailbox message to the PPE, which updates the dependency table and task queue. The resolved macroblock are allowed to be scheduled and the intra data produced by processing the current macroblock is required. This is the last moment possible to flush the pending DMA transfers.

5.3.4.2 Overlapping Write Back

The macroblock dependencies limit the number of concurrent macroblocks by at most one free macroblock per scan line. Furthermore the horizontal coordinates of the concurrent macroblocks have to be at least two apart. However due to the DMA restriction the write back is implemented with additional overhead in the horizontal directions. In the write back step, the content of the working buffers is written back to the appropriate locations in the frame. For luma, the block size is 48x19 and for chroma this is 32x9. Since the heights of the blocks is larger than one macroblock, the contents spans over two scan lines. This combined with the additional horizontal overhead causes overlapping frame writes when the concurrent macroblocks are to close to each other.

Since the macroblocks have variable execution times, this results in unpredictable behavior and erroneous data. Figure 5.5 illustrates the overlap for the luma component. The data values of the overlapping areas depend on the retire order of the concurrent macroblocks (CMBs).

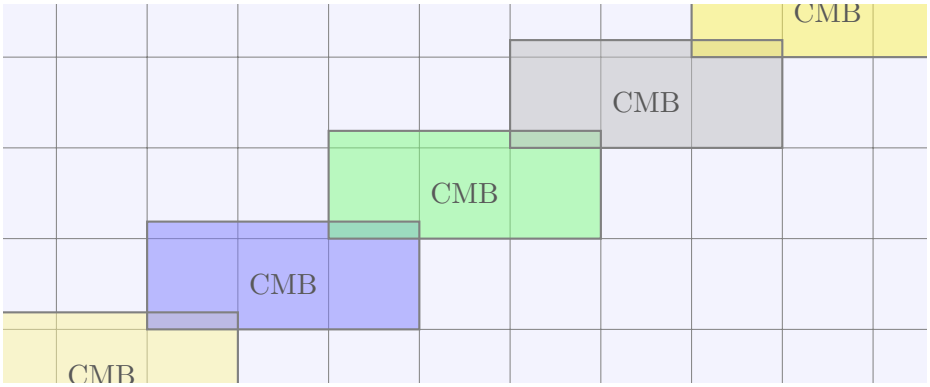


Figure 5.5: Overlaps in the write back step of the luma component by concurrent SPEs.

The overlap occurs for both luma and chroma components, albeit for different reasons. In case of luma the overwrites occur because we do not want to reduce the stride of the local store buffer. In the write back step only the right and current macroblock needs to be written back. This not desired as this would introduce additional copy steps. However, it is possible to perform to resolve the overlap.

For chroma blocks we cannot reduce the stride. Chroma blocks have a size of 8x8 pixels. Due to DMA alignment and size restrictions, half of the time it is required to write back a block of 32x9. When the current macroblock is aligned to a 16-byte address the left adjacent macroblock is not aligned and vice versa. In Listing 5.8 the `reduce_stride` is used to reduce the stride of the chroma blocks for uneven numbered blocks. Because this is only possible for odd blocks, overlaps still occur for even blocks as shown in Figure 5.6.

The simplest and best solution is to increase the minimal horizontal spacing of the CMBs. In Figure 5.6 the overlap ranges over two macroblocks. Increasing the spacing from two to four would resolve the problem. However, due to the stride reduction overlaps only occur when the CMB are on an even horizontal position. Increasing the spacing by only one also trigger an alternating effect on the CMBs. Increasing the spacing from two

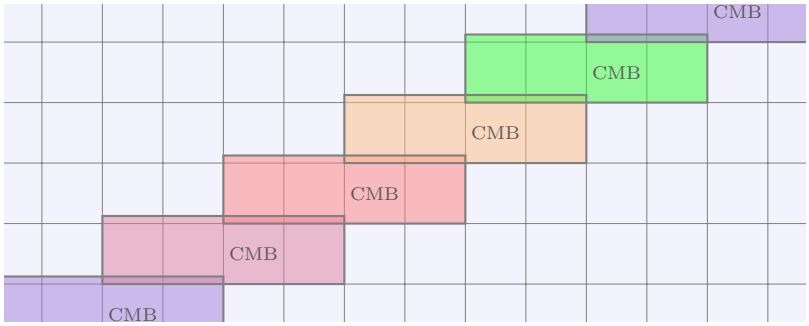


Figure 5.6: Overlaps in the write back step of the chroma components.

to three, therefore, resolves the overlapping. This is shown graphically in Figure 5.7.

Increasing the macroblock spacing effects both the luma and chroma components. The luma component technically does not require the additional spacing. However, the forced spacing does allow the stride size of the the luma component to remain the same. In Listing 5.8, no additional reduce stride procedure is performed for luma.

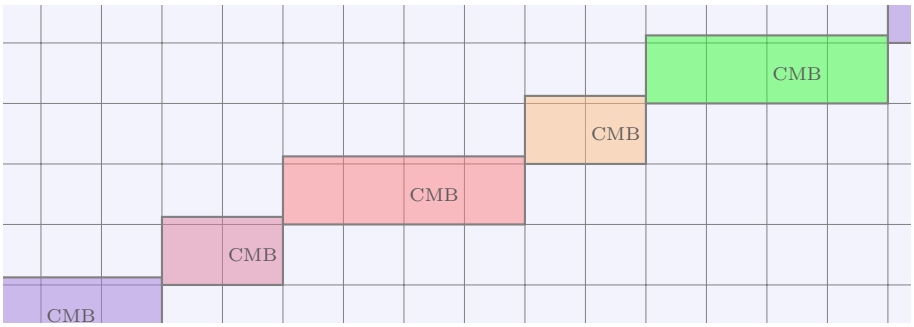


Figure 5.7: Resolving overlap by increasing the macroblock spacing.

Implementing the increased macroblock spacing requires a revised dependency structure. When updating the dependency table we can simply update the down-left-2 instead of the down-left macroblock. This ensures that a spacing of three macroblocks is maintained. Figure 5.8 shows the revised dependency structure and corresponding dependency table.

The drawback of the revised dependency structure is that it reduces the parallelism. The additional spacing reduces the average number CMBs as only one can occur on each line at same time. In the next section the impact of the revised dependency structure on the scalability is analyzed in detail.

5.3.4.3 Impact on Scalability

The solution to the overlapping write backs, discussed in the previous section, reduces the parallelism by increasing the macroblock spacing. In Equation (2.1) the maximum macroblock-level parallelism is defined. The equation states that both the frame width and height can limit the maximum parallelism. In case of FHD and all other 16:9

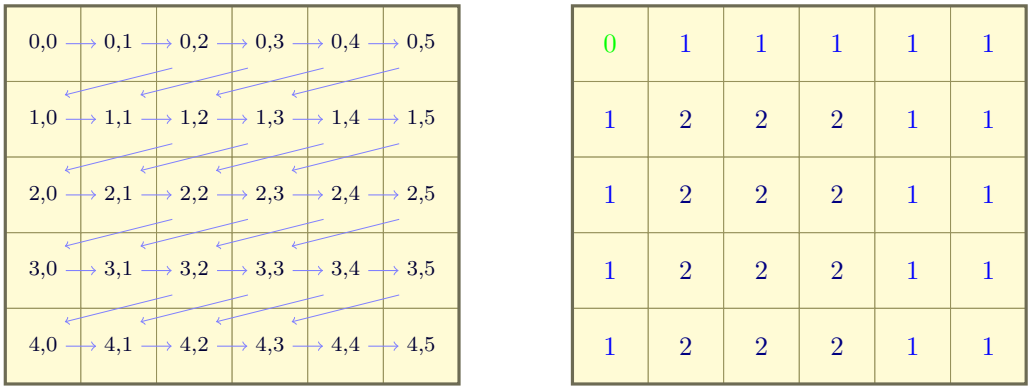


Figure 5.8: Revised dependency structure and corresponding dependency table to implement the additional spacing.

resolutions the limit is in the frame width component.

The equation is set up to abide the macroblock dependencies. In our case, however, this is changed with the increased spacing. Equation (5.1) is a revised version of Equation (2.1), which now takes a spacing variable into account.

$$ParMB_{revised} = min(\lceil N_{MB,hor}/Space_{hor} \rceil, N_{MB,ver}) \tag{5.1}$$

With the regular dependency structure the horizontal spacing is two and Equation 5.1 equals Equation 2.1. In our case it is three which results a maximum parallelism of 40 for FHD, down from 60 originally. The additional horizontal spacing reduces the parallelism by 1/3. This reduction clearly impacts the scalability and efficiency of TP.

In Chapter 4 the efficiency losses on the algorithm level were investigated. The efficiency loss incurred by the reduced parallelism is platform specific. Hence, if the DMA unit could work with an 8-byte alignment this would not be a problem. The platform specific effects, e.g. synchronization overhead, were not taken into account due to the complexity of accurate simulation. However, the solution for overlapping write backs required a revision of the TP algorithm. The effects can be simulated with some small modifications in the simulator.

In Figure 5.9 the results for the simulation of the reduced parallelism are plotted together with the TP results of Figure 4.17. The plot uses FHD sequences with variable macroblock execution times and QHD with Gaussian random variables with a standard deviation of 5 and a mean of 20.

As the plot shows, the additional spacing definitely impacts the scalability. The maximum speedup is lower and reached with lower number of processing elements. However, if we focus on the scalability up to 16 processing elements the impact is minimal. The speedup for FHD at 16 processing elements is 14.3x for a spacing of 3 compared to 14.8x with the regular spacing.

The relative low decrease in speedup is quite surprising since the maximum parallelism has been reduced from 60 to 40, a reduction of 1/3. However the maximum speedup is only around 20% lower. To understand this, first take a look at Figure 5.10. The plot shows the macroblock ramping for a FHD frame with unlimited number of pro-

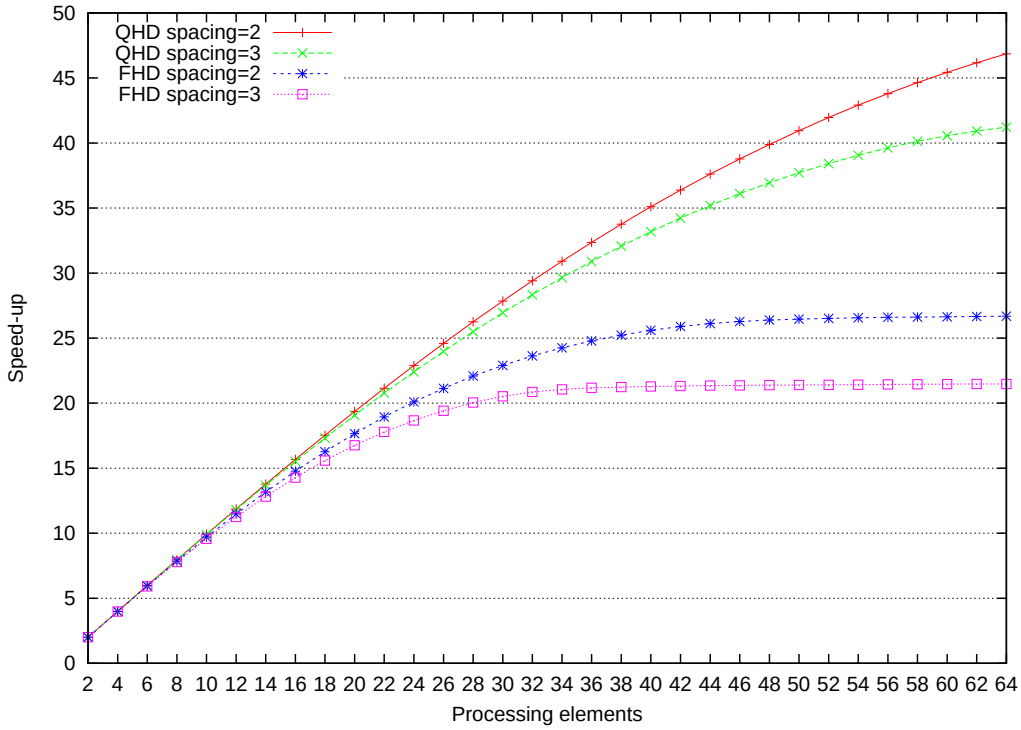


Figure 5.9: Impact on the scalability due to the additional spacing.

cessing elements. Constant macroblock execution times are assumed. It can be clearly seen that both the maximum number of free macroblocks and the ramping slope has decreased by 1/3. However the time necessary to complete the frame is 318 compared to 251 for regular spacing. This coincides with the reduced speedup as 21% less time is required to perform the regular TP decoding, with unlimited number of processing elements.

The difference of the two plot lines is in the horizontal part between the ramping. For larger spacings this is much longer. This means that the maximum parallelism is sustained longer and in turn results in a lower scalability drop. The number of time units the maximum parallelism is maintained is proportional to difference of the vertical macroblock resolution and the maximum parallelism. The factorial difference of how long the maximum parallelism is maintained for the two spacings is $\frac{68-40}{68-60} = 3.5$. Note that even with the reduced parallelism TP still has better efficiency and scaling compared to RL.

In the next section we discuss the implementation of RL. The main differences in the implementation lie in the concurrent communication and computation model, the local store memory management, and the distributed control mechanisms. These implementation differences originate from the data flow characteristics of the RL algorithm.

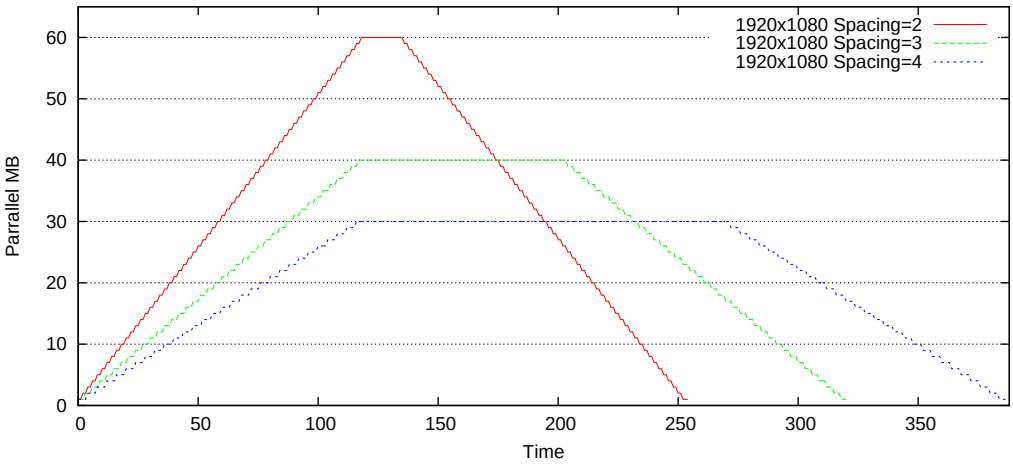


Figure 5.10: Macroblock ramping with reduced parallelism with constant macroblock execution times.

5.4 Ring-Line Implementation

In the previous sections the implementation regarding the common and TP parts have been discussed. The focus has been on working with the memory hierarchy of the Cell processor. In this section we continue this with the RL implementation. A distinct difference, however, is that the Cell memory hierarchy is no longer seen as a burden as the RL algorithm maps efficiently on the Cell architecture.

The RL implementations is discussed in several parts. First, the inter-core interface is discussed. On the Cell it is required to create separate programs for the PPE and SPE. To communicate between them an interface needs to be defined. Due to the distributed control mechanism in RL, also a SPE to SPE interface is required. In the second part the local store buffers are discussed. This part concerns the arrangement of the working buffers and the RL buffers, discussed in Section 4.2.3.3. In the third part the actual macroblock processing is discussed. The macroblock processing itself is quite straight forward. The interesting part is in the surrounding steps which provide the data. Compared to TP this has been simplified and is more efficient. Following this is the write back step. Like the data transfers for the macroblock processing, this is more efficient in RL. However, the solution also impacts the scalability. The impact on the scalability is analyzed and discussed in detail. The final part concerns the pre-buffering step. Pre-buffering is the most important step in terms of performance improvement compared to TP. The goal of the pre-buffering is to hide the latencies of the motion data and work unit transfers. In the motion compensation kernel the communication had to be decoupled from the computation This required a relatively large revision in the motion compensation code.

Throughout the section simplified code fragments are presented. The code fragments serve as the starting points for discussing the implementation decisions.

5.4.1 Inter-Core Interface and Distributed Control

5.4.1.1 Interface Between PPE and SPE

In RL the interface between the PPE and SPE is quite simple. The PPE does not need to act as a server as was the case for TP. The PPE only needs to start the SPEs once each frame. The simplified code fragment of this process is shown in Listing 5.9. The entropy decoding is done beforehand and its result is assumed to be in the `blocks` array.

First, the slice informations is copied to a `H264slice` structure pointed to by `slice_ea`. Then the slice code is send to each SPE. For convenience the slice code is the same as in TP. After this the PPE simply waits for all SPEs to finish processing their macroblock lines and message back. After the SPEs have signaled back, the frame is completely processed.

Listing 5.9: PPE program interface of RL.

```
1 //start all threads with the slice task id
2 copy_context_to_slice(slice_ea, h);
3 for (i=0; spe_threads; i++){
4     spe[i].id = -1;
5     _spe_in_mbox_write(spe_mbox[i],task[i].id);
6 }
7
8 for (i=0; i<spe_threads; i++){
9     _spe_out_mbox_read(spe_control_area[i]);
10 }
```

The simplified SPE main code is listed in Listing 5.10. At the start of the code, before it messages back to the PPE, a DMA transfer is initiated to bring in a `H264spe` structure. In TP the main argument contains the `slice_ea` and no other information is needed. For RL several other parameters are also required for successful execution. These parameters are `spe_id`, `spe_total` and the `tgt_spe` pointer. The `H264spe` structure packs these parameters together with the `slice_ea`. The `spe_id` and `spe_total` are used to find out the which lines the SPE needs to process. The `tgt_spe` pointer is the global memory pointer of the target SPE local store space. It is used as an memory address offset to transfer control and intra data directly to the target local store. Then, a confirmation is sent to the PPE and the SPE enters the "ready" state.

In the "ready" state, the SPE expects one of two signal codes via the mailbox. In case the stop signal is received, it simply returns and exit. When it receives a slice signal the processing of the frame/slice starts. With this the PPE to SPE interface is defined. Compared to TP only a single mailbox message is required each frame.

The first step in decoding the macroblocks is to issue a DMA get operation for the `H264slice` structure. After `slice` is filled and some initialization functions it enters a while loop in which the actual macroblock processing is performed. The functions in the while loop take care of the distributed control, data pre-buffering, macroblock processing and write back. In the next section the SPE to SPE interface and the distributed control are discussed.

Listing 5.10: The SPE main loop of the RL implementation.

```
1  H264Context_spu h_context;
2  int main(uint64_t argp)
3  {
4      H264Context_spu* h = &h_context;
5      H264spe* spe = &h->spe;
6      H264slice* slice = &h->slice;
7      H264spe *spe_params = (H264spe *) argp;
8
9      spu_dma_get(spe, spe_params, sizeof(h->spe), ID_spe);
10     wait_dma_id(ID_spe);
11
12     //send confirm to ppu
13     spu_write_out_mbox(1);
14     while(1){
15         task_id = (int) spu_read_in_mbox();
16         if (task_id==-2) {
17             return 0;
18         }
19         else if (task_id==-1){
20             spu_dma_get(slice, spe->slice_params, sizeof(h->slice),
                   ID_slice);
21             wait_dma_id(ID_slice);
22             int stride_y = (slice->mb_width%2)? (slice->mb_width+1)*16:
                   slice->mb_width*16;
23             int stride_c = stride_y>>1;
24
25             init_block_offset(stride_y, stride_c);
26             init_mb_buffer(h);
27             while((h->mb=(H264mb *)get_next_mb(h))){
28                 while(!dep_resolved(h));
29                 hl_decode_mb_internal(h, stride_y, stride_c);
30                 update_tgt_spe_dep(h, 0);
31             }
32         }
33         spu_write_out_mbox(task_id);
34     }
35 }
```

5.4.1.2 SPE Distributed Control

One of the main differences between the TP and RL implementation is the distributed control. The control task in TP is assigned to the PPE and has a centralized behavior. In RL each SPE is part of the controller. In the algorithm each RL processor is only dependent on its neighbors. The processors are connected in a ring structure. This provides a scalable synchronization structure as every processor has a different target.

The functions `dep_resolved` and `update_tgt_spe_dep` in Listing 5.10 handle the distributed control. These functions respectively stall the execution when the macroblock dependencies are not met and send the control data. The function bodies are presented in Listing 5.11.

Listing 5.11: Functions implementing the distributed control mechanism.

```
1  DECLARE_ALIGNED_16(spe_pos, dma_temp); //dma temp for sending
2  DECLARE_ALIGNED_16(volatile spe_pos, src_spe); //written by SPE_ID -1
3
4  static inline int dep_resolved(H264Context_spu *h){
5      H264slice *slice = &h->slice;
6      int spe_id = h->spe.spe_id;
7      int mb_proc_dep = src_spe.mb_proc;
8      if (spe_id==0)
9          return (h->mb_proc < mb_proc_dep-1 +slice->mb_width)? 1:0;
10     else
11         return (h->mb_proc < mb_proc_dep-1)? 1:0;
12 }
13
14 static void update_tgt_spe_dep(H264Context_spu *h){
15     H264mb *mb = h->mb;
16     H264slice *slice = &h->slice;
17     H264spe *spe = &h->spe;
18     int mb_x = mb->mb_x;
19
20     if (mb_x%2==0 && mb_x!=0) || mb_x==slice->mb_width-1){
21         spe_pos* dma_spe = &dma_temp;
22         spe_pos* tgt_spe = spe->tgt_spe + (unsigned) &src_spe; //located in
               target spe local store
23         dma_spe->mb_proc = h->mb_proc;
24         spu_dma_barrier_put(dma_spe, tgt_spe, sizeof(dma_temp), ID_put);
25     }
26     h->mb_proc++;
27 }
```

The function `dep_resolved` checks if the dependencies are resolved. The function is repeatedly called in a while loop until the dependency resolves. The algorithm for determining if the dependency is resolved is quite small and contains only a few basic operations. It check if the source SPE has processed two macroblocks more than the current SPE. If this is the case the next macroblock in the line can be processed. In RL there is an implicit rule that states that the SPEs cannot gain on each other. This check satisfies this rule for all SPEs except the first one. To close the ring structure SPE 0 is dependent on the last SPE. The last SPE can never have processed more SPEs than the first. This is solved by offsetting the number of processed macroblocks with one line for SPE 0.

As you can see the `dep_resolved` "polling" fashion. When a thread has to wait in traditional programs it often goes in a sleep state and is waken up by a signal. This synchronization construct is relatively slow and not useful in fine-grained parallelism. The polling mechanism implemented here has a very fast response, since `dep_resolved` contains only a few basic operations. Furthermore, polling on a SPE does not consume shared resources. The local store and the processing engine both reside in the SPE.

The only parameter required for the synchronization is the number of macroblocks the current SPE has processed. This allows for a very simple communication. In

`update_tgt_spe_dep` this parameter is packed in a 16-byte `spe_pos` structure and send via a barrier DMA. The use of a barrier DMA is required as it synchronizes the DMA queue. When a barrier command is issued, the previous issued DMAs must retire before the barrier command is processed. This ensures that the intra data arrives before the synchronization data. The number of macroblocks processed is incremented afterwards.

An `if`-condition is present around the barrier DMA operation. The condition specifies that the target SPE should only be updated for every even block and the last macroblock in the line. This has to do with implementation issues regarding the write back and write intra data step issued at the end of `hl_decode_mb_internal`. More on this is revealed in Section 5.4.4.

5.4.2 Local Store Buffers

The actual macroblock processing is done in `hl_decode_mb_internal`. The internal structure is quite different from TP, despite using the same function name. While the kernels themselves are mostly the same, the memory management needed an overhaul in order to support the advantages of the data flow behavior of the RL algorithm. As is the case for TP, the Cell memory hierarchy requires additional effort for the programmer. The RL algorithm, however, can make efficient use of this as it maps well on the memory hierarchy of the Cell architecture. In this section we focus on the data management of the local store buffers with exception of the motion data and work unit buffers. These buffers are part of the pre-buffering scheme, which is explained later in the chapter.

As explained in Section 4.2 the RL algorithm needs a buffer to store pre-sent intra data. The larger the buffers the better the performance until it reaches the maximum buffer size specified by Equation (4.4). An optimum in the buffer size is found in Equation (4.6), which delivers the same performance with only two times the minimal buffer size. While in theory this is an attractive trade-off between memory requirements and performance, additional parameters are to be considered in the practical implementation.

Before continuing explaining the choices regarding the buffers and buffer sizes, we should take a look at the implemented buffers in Listing 5.12. Each of the buffers allocated in the listing is discussed in the next sections.

5.4.2.1 Combined Ring-Line and Working Buffer

The line buffers act as both the working buffers and RL buffers. The requirements of the RL buffer are that it is able to fit the intra data of the source SPE. The requirements of the working buffer are to be able to fit the intra data and the current macroblock, to serve as DMA targets and sources, and to have the macroblock kernels operate on them. By combining these buffers the memory organization becomes more efficient in terms of performance.

The line buffers are statically allocated to contain 20 lines of a FHD image. This results in the luma buffer `dest_y_ls` of 120*16*20=38400 bytes and the two chroma buffers `dest_cb_ls` and `dest_cr_ls` of 120*8*10=9600 bytes. The choices made regarding the buffer allocations are to extract maximal performance from the SPEs. In Chapter 6 a more balanced approach is discussed, which makes a trade off between buffer size and additional copy steps.

Listing 5.12: Local store allocation of the combined working and RL buffers, border buffers and write back buffers.

```
1 //mb line buffers - statically allocated for up to 1920 width video
2 DECLARE_ALIGNED_16(uint8_t, dest_y_ls[120*16*20]);
3 DECLARE_ALIGNED_16(uint8_t, dest_cb_ls[120*8*10]);
4 DECLARE_ALIGNED_16(uint8_t, dest_cr_ls[120*8*10]);
5
6 //border buffers
7 DECLARE_ALIGNED_16(uint8_t, top_border_ls[120][16 + 2*8]);
8 DECLARE_ALIGNED_16(uint8_t, left_border_ls[17+2*9 + 13]); //last 13 is
      padding
9
10 //write back buffers
11 DECLARE_ALIGNED_16(uint8_t, dma_y_ls[32*20]);
12 DECLARE_ALIGNED_16(uint8_t, dma_cb_ls[16*10]);
13 DECLARE_ALIGNED_16(uint8_t, dma_cr_ls[16*10]);
```

With a height of 10 and 20 lines the line buffers are the same as the working buffers used in TP in this regard. The height allocation allows the source SPE to directly write the intra dependency data into the working buffer. An additional copy step would be required in case the working and RL buffers were separated.

The width of the buffer is chosen with the optimal RL buffer size of Equation 4.6 in mind. Equation 4.6 specifies that the optimal buffer size is equal to two times the frame width in terms of macroblocks divided by the number of ring processors. This means that with more processors, less buffer slots per processor are needed. The implementation, however, needs to support a range of processing elements for the sake of investigating the scalability. The required buffer size is largest when there is only one processor decoding a FHD sequence. In this case the number of buffer slots would equate to 2*120/1=240. However 240 buffer slots does not fit in the local store. Therefore, only two or more SPEs are supported in the RL implementation. Two SPEs requires a buffer with 120 slots, which fits in the local store.

Since the buffers are statically allocated, they are used with every resolution video and SPE configuration. When using a lower resolution only part of the line buffer is used. This does not impact performance. However, the same cannot be said for using less available buffers with increasing number of SPEs. According to Equation (4.6), using more processors results in a lower buffer size requirements while maintaining equal performance. In practice, however, it still is desired to use a line buffer for every SPE configuration. This has two distinct reasons, both concerning the performance.

First, using a smaller buffer requires extra communication with the source SPE. When using a buffer size lower than the maximum buffer size specified by Equation (4.4), it is necessary to inform the source SPE which buffer slots are free. While it is unlikely to have a buffer stall with the optimal buffer size, it is still possible that this occurs. Therefore, it is required to communicate with the source SPE to prevent overwriting buffers with valid data at all time. Communicating with the source SPE is not needed if the maximum buffer size is used. In this case buffer stalls are fully resolved.

The second reason is to prevent a copy step when the buffers run full. Recall that the line buffers are acting both as the working buffers and RL buffers. As working buffers the intra data should always be placed relatively to the current macroblock, as shown in Figure 5.3. As RL buffers the buffer slots are used circular. These two requirements conflict at the circular loop back as illustrated in Figure 5.11.

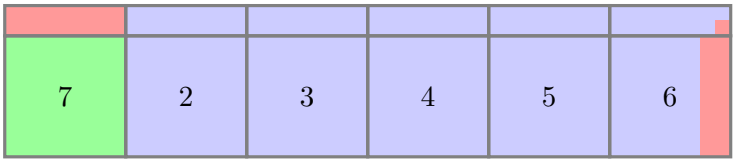


Figure 5.11: Buffer conflict at the circular loop back.

In the figure we see that the left intra data is at the end of the buffer while the top intra is at the beginning. When using this as a working buffer it would not be possible to decode macroblock 7. A possible solution is to put another buffer slot at the front. This slot should not be used by the source SPE to write dependency data. Before decoding macroblock 7, in now the second slot, macroblock 6 should be copied to the first buffer slot. Smaller buffers require this additional action more often than larger buffers.

However, the loop back conflict can be completely avoided by using an entire line as a buffer. For the first macroblock in the line no left intra data is needed. By taking the buffer the same size as an line, the circular loop back and the line start will coincide. In this case no additional actions are necessary.

In Chapter 6 the memory usage of both TP and RL is investigated more in depth. In the same chapter we also comment on a possible adjustments to support higher than FHD resolutions. This is not possible by scaling the current implementation due to local store size limitations.

5.4.2.2 Border Buffers and Data locality

Next to the line buffers in Listing 5.12, there are two border buffers. The border buffers contain the unfiltered borders send by the source SPE. The same as for the line buffers, the `top_border_ls` contains enough slots for a line of a FHD image. The `left_border_ls` contains the left border of one macroblock.

The left border buffer can only contain the unfiltered right border of one macroblock. Because the SPEs process an entire line the right border is used by the same SPE. The SPE can keep the unfiltered borders locally since it is only used while processing the next macroblock. This structure is actually quite similar to the approach used in the original libavcodec in which the macroblocks are also decoded in scan line order.

In TP the unfiltered borders resides in the `H264mb` work units. With TP, intra data would always be transfered via the external memory through the frame buffer and the `H264mb` work units. The predictability of the RL algorithm allows us to keep all the intra data on-chip. In the RL structure only the next SPE requires the intra data as it processed the next line. Due to the unpredictable dynamic assigning, this was not possible in TP.

5.4.2.3 Write Back Buffer

The write back buffers are used as the source buffer for writing the picture data back to the frame, residing in the external memory. The buffers are needed to sequentialize the picture data from the line buffer. The DMA unit is not capable of strided local store accesses. Since the processed data resides in a line buffer it is not possible to send an entire block with a DMA list operation directly. An alternative is to use a separate DMA operation for each block line. However this would decrease performance even further due to DMA setup overhead.

The size of the DMA transfer buffers is chosen to contain two macroblocks in width. This automatically means that two macroblocks are written back at once. It is necessary due to the alignment restrictions of the Cell platform. The write back procedure is explained further in Section 5.4.4.

5.4.3 Macroblock Processing

As explained in the previous section, the RL macroblock kernels operate on a line buffer. By combining the RL and working buffers less local copy steps are required. This section shows what consequences this has on the macroblock processing. The implementation of the RL `hl_decode_mb_internal` used the TP version as a starting point.

The first part of `hl_decode_mb_internal` is shown in Listing 5.13. The first thing to notice is that the stride size is no longer fixed. Now it is equal to the line width since our working buffer is an entire line. Second, there is no longer a DMA step at the start to bring in the intra data. What remains is calculating the start point of the current macroblock in the line buffer. Before decoding the macroblock the top intra data is already present due to pre-sending by the source SPE. The left intra data is inherently present due to decoding of the previous macroblock.

The function calls to the intra and motion prediction kernel remain the same. For both kernels internal modification were needed to support the variable stride sizes. Furthermore, the `xchg_mb_border` function now has to operate on the new border buffers instead of the ones located in `H264mb`. In addition to the variable stride size, the code in `hl_motion` needed a big overhaul to support pre-buffering of the motion reference data. This, however, is abstracted from `hl_decode_mb_internal`. The overhaul in `hl_motion` is part of the pre-buffering and is discussed in Section 5.4.5.

The second part of `hl_decode_mb_internal` is shown in Listing 5.14. The same as for the previous kernels the IDCT, IQ and deblocking filter now need to operate on a line buffer, which has a stride size depending on the resolution. Adding the support for variable stride sizes has little to no impact on the performance.

The main difference in Listing 5.14 is in storing the unfiltered borders. The new border buffers are used in `backup_mb_border`, similar to `xchg_mb_border`. The `H264mb` fields for the borders are not needed in RL as the intra data is kept on-chip. Remember that in TP a DMA was issued before `filter_mb` which executes the deblocking filter kernel. This is no longer present since it is moved to the write back phase. This is actually more natural as it is part of the intra data, which is now send together. The border DMA was issued as soon as possible in the TP version for optimization reasons.

Listing 5.13: Macroblock processing part 1 - Prediction and border exchange.

```
1
2  void hl_decode_mb_internal(H264Context_spu *h, int stride_y, int stride_c){
3      H264slice *slice = &h->slice;
4      H264mb *mb = h->mb;
5      const int mb_x= mb->mb_x;
6      const int mb_y= mb->mb_y;
7      const int mb_type= mb->mb_type;
8
9      uint8_t *dest_y, *dest_cb, *dest_cr;     //ls ptrs (abstracts the fact
           it is operating on a ls buffer)
10     int i;
11
12     dest_y = dest_y_ls + mb_x*16 + 4* stride_y;
13     dest_cb = dest_cb_ls + mb_x*8 + 2* stride_c;
14     dest_cr = dest_cr_ls + mb_x*8 + 2* stride_c;
15
16     if(IS_INTRA(mb_type)){
17         if(slice->deblocking_filter)
18             xchg_mb_border(h, dest_y, dest_cb, dest_cr, stride_y, stride_c,
                   1);
19
20         intra_prediction{h, dest_y, dest_cb, dest_cr, stride_y, stride_c};
21
22         if(slice->deblocking_filter)
23             xchg_mb_border(h, dest_y, dest_cb, dest_cr, stride_y, stride_c,
                   0);
24
25     }else {
26         hl_motion(h, dest_y, dest_cb, dest_cr, stride_y, stride_c);
27     }
```

Another important difference is that only the lower unfiltered borders are transfered to the target SPE. Recall that in TP both the lower and right borders are transfered to the corresponding `H264mb` structures in the external memory. This was necessary because it is not known which SPE processes which macroblock. However in RL the SPEs process an entire line. The right borders are only used when processing the right macroblock, which is processed on the same SPE. The lower borders of the three components are packed in a single DMA transfer and sent to the top border buffer of the target SPE.

The predictability of the RL algorithm is exploited to simplify transferring the intra data, while also making it more efficient. Less transfers are needed and more data is kept on-chip. Furthermore, the data transfers are non-blocking. The same holds for the transfers in the write back step discussed in the next section.

5.4.4 Write Back and Impact on Scalability

After applying the kernels the picture data has to be written back to the frame. Recall that in TP this was quite inefficient. For every decoded macroblock a size of two to four blocks were written back. Also three transfers of the unfiltered borders to the external

Listing 5.14: Macroblock processing part 2 - IDCT, IQ and deblocking filter.

```
1
2     IDCT_luma(h, dest_y, block_offset, stride_y);
3     IQ_chroma(h);
4     IDCT_chroma(h, dest_cb, dest_cr, block_offset, stride_c);
5
6     if(slice->deblocking_filter){
7         int mb_height = slice->mb_height;
8
9         // save unfiltered borders
10        backup_mb_border(h, dest_y, dest_cb, dest_cr, stride_y, stride_c);
11
12        /* border DMA moved to write back*/
13
14        //deblocking filter
15        filter_mb(h, mb_x, mb_y, dest_y, dest_cb, dest_cr, stride_y,
              stride_c);
16    }
```

memory were required per macroblock. It is already revealed that the latter is no longer needed. Data locality and pre-sending the intra data have replaced this completely.

The RL algorithm also allows us to optimize the necessary write back operations. This is actually as much as an optimization as it is an obligation. With the RL algorithm the write back is delayed until two blocks can be written back at once. The chroma blocks have a width of 8 pixels and cannot be written back by itself due to the DMA alignment restrictions. In TP up to four blocks in width were written back.

During the write back step also the intra data is pre-sent to the the target SPE. For the same reason as delaying writing the picture data to the frame, this is also delayed. However, delaying the intra date pre-send step effectively causes a larger spacing of the concurrent macroblocks. The larger spacing impacts the scalability negatively. In this section both the write back step and its impact on the scalability are discussed in detail.

5.4.4.1 Write Back

In the write back step of the RL implementation the picture data and the intra data is transferred to the external memory and target SPE respectively. The code fragment in Listing 5.15 takes care of the write back to frame and inter-SPE transfers. In the listing the actual DMA transfers are issued in `dma_pic_data`. This is always preceded by a `wait_dma_id` and a `prepare_buffer` call. The `wait_dma_id` is necessary to ensure that the previous write back has completed to prevent valid data overwrites in the write back buffers. The `prepare_buffer` function fills the write back buffers with the to be transferred data. The `wait_dma_id` call normally does not block as the time between write back steps mostly equates the decoding time of two macroblocks.

From the `if`-conditions we can see that the write back step should only start if the current macroblock has a non-zero even x-coordinate or is the last one in the line. If both apply the write back is issued twice. Furthermore, each condition has an `offset`.

Listing 5.15: Macroblock processing part 3 - Write back step RL.

```
1     int offset;
2     //send output data to tgt spe and picture in main mem
3     if (mb_x%2==0 && mb_x!=0){
4         offset = 2;
5         wait_dma_id(ID_put);
6         prepare_buffer(dest_y, dest_cb, dest_cr, stride_y, stride_c, offset
              );
7         dma_pic_data(h, mb_x-offset, mb_y, stride_y, stride_c);
8
9         if(mb_x == slice->mb_width-1){
10            offset=0;
11            wait_dma_id(ID_put);
12            prepare_buffer(dest_y, dest_cb, dest_cr, stride_y, stride_c,
                  offset);
13            dma_pic_data(h, mb_x-offset, mb_y, stride_y, stride_c);
14        }
15    }else if(mb_x == slice->mb_width-1){
16        offset = 1;
17        wait_dma_id(ID_put);
18        prepare_buffer(dest_y, dest_cb, dest_cr, stride_y, stride_c, offset
              );
19        dma_pic_data(h, mb_x-offset, mb_y, stride_y, stride_c);
20    }
21 }
```

The `offset` indicates which blocks should be transfered. The write back buffers have a width of two macroblocks. This means that always the width of two blocks is transferred once. An `offset` of two means that the previous two blocks are transfered. An `offset` of one means the previous and the current, and zero means the current and the next. For now let us focus on the most common condition, which is when `offset` equals two.

In the write back step two adjacent blocks are paired due to the DMA alignment restriction. For luma the alignment is not a problem, since it is always aligned to a 16-byte boundary. However, chroma has a width of 8 pixels and is only aligned half of time. If we would have transfered the paired blocks separately, it would result in writing the exact same area twice directly after each other. Combining them prevent the redundancy.

The reason to use the two previous blocks is that the operations on the blocks have not finished before the current macroblock has been decoded. Recall that the deblocking filter modifies not only the current macroblock, but also the left and top adjacent borders. In TP the right adjacent block had to be written back as well for this reason. Again, because in RL an entire line is decoded on the same SPE, the write back can be delayed until all processing has been performed. Doing so prevents any redundant transfers in the horizontal direction. Figure 5.12 illustrates the write back timings for a short line.

At the end of the line the blocks are written back directly after it is processed. For the last macroblock in the line there is no next macroblock that performs the deblocking filter. For an even line size (last MB has an odd x-coordinate) an `offset` of one is

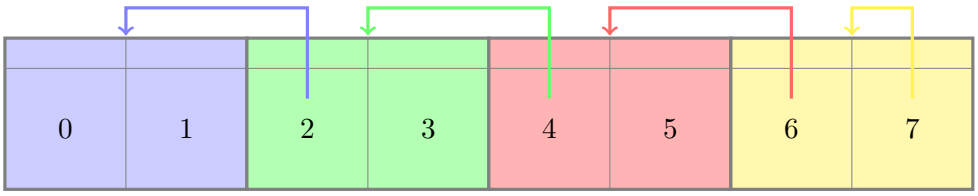


Figure 5.12: Delayed and combined write back of previous two macroblocks to avoid redundancy.

needed. For uneven line sizes the last block has no adjacent pairs and an offset of zero is used. The `offset` also applies for the unfiltered border and intra data transfer to the target SPE. By applying this scheme no picture data is written more than once in the horizontal direction.

In the vertical direction, however, frame overwrites still occur, since the top four lines are written back as well. In TP this caused concurrent write overlaps and the blocks were spaced out further to prevent unpredictable and erroneous results. While this solved the overlapping, it did not reduce the number of transfers. In RL we can keep the data on-chip until all processing has been performed. The lower four(two for chroma) lines are sent to the target SPE since they are needed as intra data and the data in these lines can still change. The solution is to let the target SPE write the lower four lines of the current MB back to the frame after it has processed them. Only the top 16 (8 for chroma) lines of the write back buffers are transferred to back to the frame, while the lower four(two for chroma) are pre-sent to the target SPE. Figure 5.13 shows that this cannot be applied for the top and bottom line. In practice the top macroblock line can be treated the same since there is an extra border around the frame. With the bottom line of frame the lower lines are not needed as intra data and the write back buffer is fully written to the frame. All the actual DMA operations are issued in `dma_pic_data`, which consist of both the write back to frame and pre-send of the intra data for the luma and chroma components.

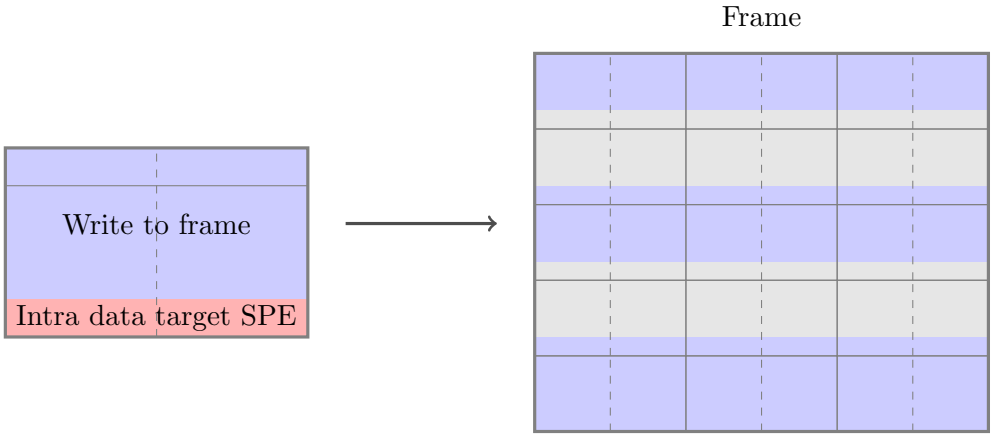


Figure 5.13: Write back targets of data in the DMA transfer buffers. The upper 16 lines go to the frame, while the lower 4 are sent to the target SPE as intra data.

Back in Listing 5.11 the function `update_tgt_spe_dep` updates the `mb_proc` counter on the target SPE. This can only be increased after the intra data is present. Therefore, the same `if`-conditions surround the barrier DMA. The counter is only incremented once in two blocks on the target SPE to synchronize with the intra data.

However delaying the pre-send of the intra data will reduce parallelism. Recall that additional spacing reduces the parallelism for TP. By delaying pre-sending the intra data the spacing is effectively increased. Looking at Figure 5.12 reveals that the spacing between macroblocks is either three or four macroblocks. In the next section the impact of the delayed intra data pre-send on the scalability is analyzed.

5.4.4.2 Impact on Scalability

The combining and delaying of the write back results in a larger horizontal spacing of the concurrent macroblocks. Since the implementation affects the algorithm, the effects can be simulated. For this purpose the earlier used simulator of Section 4.2.3 is updated to exactly simulate this write back behavior. To investigate the effects of the additional spacing, the regular and practical RL variants are compared. For both variants variable MB execution times and the maximum buffer size are used. The results of the simulations are plotted in Figure 5.14.

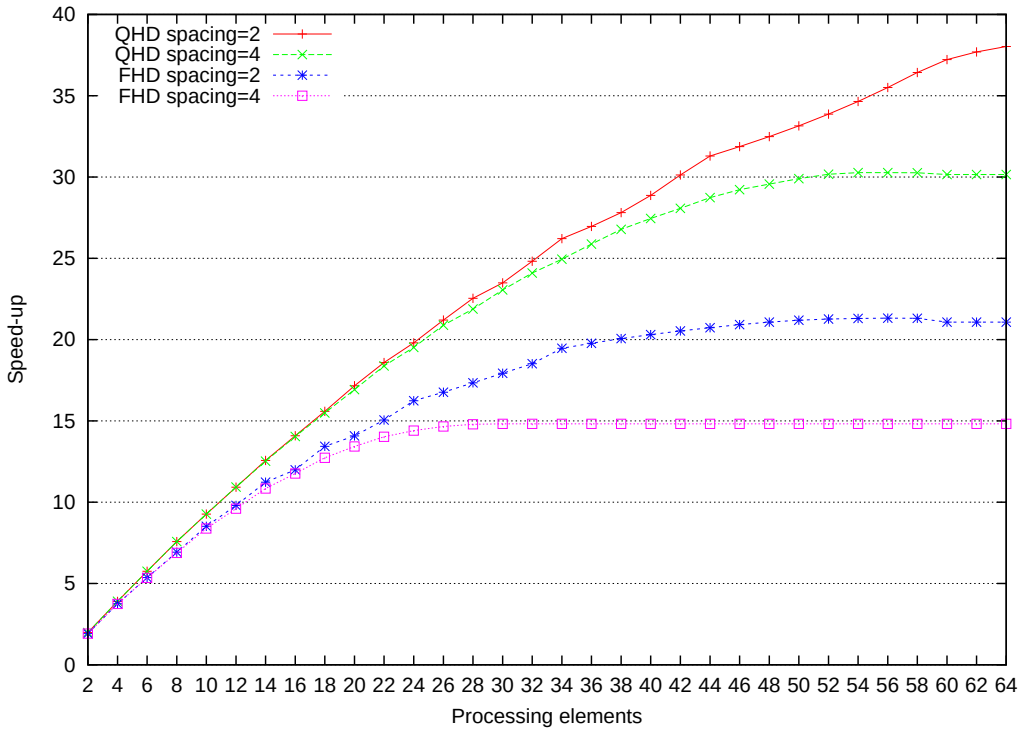


Figure 5.14: Impact of increased spacing on the Ring-Line scalability.

From the figure we see that the scalability suffers considerable with higher number of processing elements. Compared to the reduced parallelism results of TP, in Figure 5.9,

we see that the impact on RL is larger. This is not surprising as the parallelism with a spacing of four decreases to half the original, according to Equation 5.1. The effect on 16 processing elements and less remains small for both FHD and QHD. The attained speedup for FHD using 16 SPEs is 11.7x, down from 12x. It can be seen that the scalability remains close to the regular spacing up to a certain point. After this point the scalability is quickly saturated.

5.4.5 Pre-Buffering

The final part of the RL implementation concerns arguably its most important task. In Figure 4.11 it was shown that it is possible to concurrently do the communication and computation in RL. The write back and intra data pre-send step already are concurrent as they are implemented with non-blocking DMA calls. To have full concurrent communication the work units and the motion data must be pre-buffered.

Decoupling predictable communication from computation allows for very efficient use of the Cell architecture. It is generally believed that future architectures will incorporate a Cell-like memory hierarchy. However, most programmers cannot properly handle the responsibility of managing the data flow. Often this is an even larger burden than thread-level parallelization. Furthermore, the application is not always eligible for exploiting concurrent computation and communication. However, in a lot of situations it is possible and when this is the case the performance advantages can be considerable. The memory latencies are hidden and the effects off the "Memory Wall" are avoided. The implementation of H.264 could, therefore, serve as an example for programmers as well as a reference for future programming models.

The pre-buffering implementation is discussed in several parts. First, the actual pre-buffering strategy is discussed as it is shown in Listing 5.10. The functions `init_mb_buffer` and `get_next_mb` called in main loop abstract the pre-buffering process. Following, the communication decoupling of the motion compensation kernel is discussed. Finally, the motion data buffer allocation is revealed.

5.4.5.1 Pre-buffering mechanism

The function calls of `init_mb_buffer` and `get_next_mb` in Listing 5.10 abstracts pre-buffering process. It is good programming practice to keep things ordered and divided as much as possible. The body of the two functions are mostly the same. For `init_mb_buffer` the difference is that it must handle the irregularities of starting the pre-buffering. We first focus on `get_next_mb` and comment on `init_mb_buffer` in a later stage. Now take a look at `get_next_mb` in Listing 5.16.

In the `get_next_mb` function several steps are taken. At the start of the function an `if`-condition checks if there is still work to do. When all macroblocks have been processed a zero is returned and the main loop in Listing 5.10 exits. If this is not the case, up to two DMA transfer tasks are performed. The first one is to pre-buffer a `H264mb` working unit. The second is to pre-buffer the motion data if applicable.

In normal situations one level of pre-buffering suffices in form of double buffering. While one of the buffers is used in the processing, the other is filled by the pre-buffering and vice versa. However, in this case pre-buffering the motion data is dependent on its

Listing 5.16: Implementation and abstraction of the pre-buffering step.

```
#define TAG_OFFSET_MB ID_buf1
#define TAG_OFFSET_MC ID_mc_buf1

static void *get_next_mb(H264Context_spu *h){
    H264slice *slice = &h->slice;
    H264spe *spe = &h->spe;
    H264mb *mb_buf = h->mb_buf; //H264mb mb_buf[3]
    H264mc *mc_buf = h->mc_buf; //H264mc mc_buf[2]
    H264mb *next_mb, *next_dma_mb;

    if (h->mb_proc >= h->mb_total)
        return (void *) 0;

    if (h->mb_proc < h->mb_total-2){
        next_dma_mb = slice->blocks + h->mb_id;
        spu_dma_get(&mb_buf[h->mb_dma], next_dma_mb, sizeof(H264mb), h->
            mb_dma + TAG_OFFSET_MB);
        h->mb_dma = (h->mb_dma+1)%3;
        h->mb_id++;
        if (h->mb_id%slice->mb_width ==0)
            h->mb_id+=(spe->spe_total-1)*slice->mb_width;
    }

    h->mc = &mc_buf[h->mc_idx];
    wait_dma_id(h->mc_idx + TAG_OFFSET_MC);
    h->mc_idx = (h->mc_idx+1)%2;
    if (h->mb_proc < h->mb_total-1){
        wait_dma_id(h->mb_mc + TAG_OFFSET_MB);
        H264mb *mb = &mb_buf[h->mb_mc];
        H264mc *mc = &mc_buf[h->mc_idx];
        if(!IS_INTRA(mb->mb_type)){
            calc_mc_params(mb, mc);
            fill_ref_buf(h, mb, mc);
        }
    }
    h->mb_mc = (h->mb_mc+1)%3;

    next_mb = &mb_buf[h->mb_dec];
    h->mb_dec = (h->mb_dec+1)%3;

    return next_mb;
}
```

work unit. If we directly try to pre-buffer the motion data, the work unit would becomes a blocking transfer. This is solved by increasing the pre-buffering level of the work units to triple buffering. The motion data remains double buffered.

Figure 5.15 illustrates the scheduling of the first couple of macroblocks. The first two steps differ from the others, because there is no processing done and is taken care of in `init_mb_buffer`. Each of the DMA steps, in the non-rounded rectangles, are non-blocking. They are checked on completion before initiating their equivalent step in the

next slot. From the figure can be derived that triple buffering is used for the work units. In the same slot as macroblock *x* is processed, the work unit of *x+2* is pre-buffered. Similar for the motion data it can be derived that double buffering is used. In the same slot as macroblock *x* is processed, the motion data of *x+1* is pre-buffered. At slot four there is an empty slot to indicate that not all macroblocks need motion compensation.

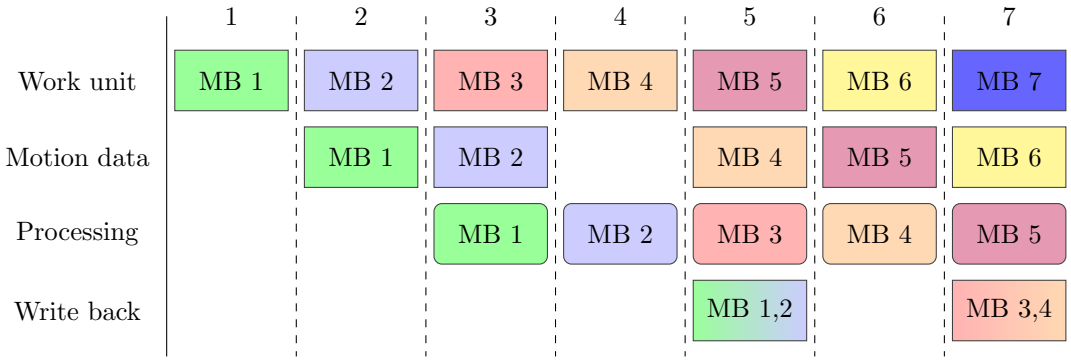


Figure 5.15: Initializing buffers and decode of first macroblocks. The DMA steps in the non-rounded rectangles are non-blocking.

If we move back to the code, several `if`-conditions surrounding the buffering are observed. The second condition prevents pre-buffering a work unit in the the last two iterations, because of the triple buffering. Also at the end of a macroblock line it is necessary to continue pre-buffering the macroblock data of the next line. To calculate the next line the SPE ID and the total number of SPEs is used. This information resides in `H264spe`.

Before proceeding with the actual transfer calls for the motion data, a wait-statement is issued to ensure that the previous issued motion data has arrived. It is required to check if the previously issued transfer has completed before issuing an overwrite and before using it. In this case the latter applies. Before the next motion data can be pre-buffered we also need to wait on its depending work unit. Since triple buffering is used the waiting time is mostly zero.

Note that several pointers have been updated. With the pointer updates two things are accomplished. First, the pre-buffering mechanism is abstracted from the processing. The processing step can simply use the data structures addressed by the pointers. Second, additional local store copy steps are avoided.

With these steps the pre-buffering mechanism is revealed. This can be applied to any predictable algorithm. It is important to remember that pre-buffering mechanism should be abstracted from the actual processing. Also the necessary levels of pre-buffering should be investigated. Finally, it is required to check the completion of the previous transfer before using the data and reusing the buffer slot.

However, to successfully perform the pre-buffering often another step is needed. In more complex cases it is necessary to force code changes for explicitly decoupling the communication and the computation. This applies to the motion compensation. From Listing 5.16 it can be seen that two functions are used to pre-buffer the motion data,

`calc_mc_params` and `fill_ref_buf`. These functions decouple the communication from the computation of the motion compensation kernel. In the next section the body of these two functions is discussed.

5.4.5.2 Decoupling the Motion Compensation

To pre-buffer the motion data it is required to decouple the communication from the computation in the motion compensation. This process is similar to the decoupling of the entropy decoding from the macroblock processing, discussed in Section 5.2.1. In this case also a new structure is introduced in form of `H264mc`. The `H264mc` structure functions as a communication structure between the pre-buffering and the processing, in the same way as the `H264mb` is used.

Looking back at Listing 5.16 shows that two functions are used for pre-buffering the motion data. Both functions operate on the `H264mc` structure. First, `calc_mc_params` is called to calculate the motion compensation parameters for identifying the necessary transfers. Among others this consist of the partitioning and the motion vectors. This information is stored in the `H264mc` structure. Second, the `fill_ref_buf` procedure issues the actual DMA transfers by using the `H264mc` structure. Furthermore, the local store pointers of each individual motion block are stored in the `H264mc`. The `H264mc` is used for decoding the next macroblock. Since the motion data is double buffered this is also the case for `H264mc`.

Showing the body of the two functions is too large for its purposes. Instead the contents of the `H264mc` and the actual double buffered motion data buffer are shown in Listing 5.17. This provides a good base for the detailed analysis of the communication decoupling.

The `H264mc` structure contains two fields. The first field is an array of 16 `H264mc_part` structures. For the motion compensation kernel the macroblock can be split up in up to 16 quadrants. Each `H264mc_part` contains the parameters associated to a quadrant. The possible quadrant configurations were discussed in Section 2.2. The second field stores the number of partitions the macroblock consists of. The contents of each `H264mc_part` is filled during the `calc_mc_params` call. These parameters are both needed by `fill_ref_buf` to issue the transfers and the modified `hl_motion` to perform the motion compensation.

The last field of H264mc_part, `ref`, is filled by `fill_ref_buf`. In `fill_ref_buf` the actual DMA transfers are issued for each quadrant by using the parameters in `H264mc_part`. For example, the `x_offset` and `y_offset` fields are the two parts of the motion vector. The `ref_data` structure contains the local store pointers of the motion data of its corresponding quadrant. The local store start pointers of each component of the motion data is stored in the pointer array `data`. Two of these `ref_data` structures are needed, to support bi-weighted prediction. Several of the others fields of `H264mc_part`, for instance `weight`, `list0` and `list1`, are used as switches for the possible motion compensation options of H.264.

The rest of the fields in `ref_data` are only filled in case edge emulation is required. In Figure 5.2 an example situation was presented when this is the case. These fields are needed in two-fold, one set for luma and one for chroma. Most of the time edge extension

Listing 5.17: Motion pre-buffering structures and motion data buffer.

```
typedef struct ref_data{
    uint8_t *data[3];
    int emu;
    int start_ysrc[2];
    int start_ydst[2];
    int end_ysrc[2];
    int end_ydst[2];
    int start_xsrc[2];
    int start_xdst[2];
    int end_xsrc[2];
    int end_xdst[2];
}ref_data;

typedef struct H264mc_part{
    int n, chroma_height;
    int x_offset;
    int y_offset;
    int itp;
    int weight;
    int list0, list1;
    int use_weight;
    ref_data ref[2];
}H264mc_part;

typedef struct H264mc{
    H264mc_part mc_part[16];
    int npart;
}H264mc;

//actual double buffer for motion data
DECLARE_ALIGNED_16(uint8_t, mc_ref[2][2*(16*(4+5)*48 + 2*16*(2+1)*32)]);
```

is not required and these fields are not used.

After the operations the `H264mc` structure contains all the parameters needed in the motion compensation. The pointers to the motion data of all partitions is contained in the structure. When the motion compensation requires motion data it can simply look up the local store pointer, instead of requesting it via a DMA transfer. However, a step still remains as the allocation of the motion data buffers on the local store has not yet been revealed. In the next section the `mc_ref` buffer where the actual motion data resides is discussed.

5.4.5.3 Motion Data Buffer Allocation

For the motion compensation kernel a variable number of buffers are needed to store each component of the motion data partitions. Furthermore, the required size of the buffers is variable. Different buffer sizes are required often. Therefore, dynamic memory allocation would have a negative impact on the performance. Instead a custom simplified memory allocation scheme is implemented. The implementation is discussed in two part.

First, we investigate the worst case memory size requirements, followed by the memory allocation procedure.

The size of the buffer is chosen to fit the worst case amount of reference data. This is done as efficiently as possible without having an negative impact on the performance and code complexity. The `mc_ref` buffer of Listing 5.17 is allocated to contain the double buffered motion data.The listing reveals that the size of a single buffer equals `[2*(16*(4+5)*48 + 2*16*(2+1)*32)]` pixels/bytes.

From Section 2.2 we know that for luma the motion data block has the size of the partition with an additional border of two pixels. Therefore, the width and the height of the block increases a total of four pixels. Due to interpolation this has to be increased by another pixel for both luma and chroma resulting in the extra 5 and 1 respectively. The worst case situation in terms of memory is when the macroblock is tiled in 16 4x4 partitions. This explains the factor 16 and, a height of 4 and 2 for respectively for luma and chroma.

The width of the blocks is 48 for luma and 32 for chroma. In Section 5.2.2.2 it was explained that transferring the motion data requires additional overhead. Finally the entire buffer size needs to be doubled to support bi-weighted prediction. With this buffer size the `mc_ref` buffer always has enough space. Furthermore, the kernels can directly operate on the buffers, since sufficient memory is allocated to support the fixed strides of 48 and 32.

The pointers in the `data` array point to locations in the `mc_ref` buffer. While there is sufficient memory in total, the allocation of the individual blocks still is left unexplained. To keep the memory allocation efficient and ordered a custom algorithm is used to handle the memory distribution. The solution is to implement a simplified version of `malloc`. The implementation requires a pointer, which is referred to as `memptr`. At the start of every pre-buffer step one of the motion buffers is "freed". This is done by setting the `memptr` to the start of the to be freed motion buffer. Every time a motion data buffer is requested, the current `memptr` is returned and incremented with the requested size. Figure 5.16 shows an example content of one of the buffer slots of the `mc_ref` buffer. In this simple scheme only one increment is required each time a motion buffer is needed.

5.5 Conclusions

In Chapter 4 two parallel H.264 decoding strategies were discussed. Both TP and RL exploit macroblock-level parallelism. In this chapter the implementation of both strategies on the Cell platform is discussed. For the implementation the open source FFmpeg code [12] is used as a base. The core of FFmpeg is the libavcodec library. The libavcodec contains the actual audio and video codecs, including a H.264 decoder. It was discovered that the interface to libavcodec poses problems for parallelizing the entropy decoding. Therefore, no attempts were made to implement 3D-Wave and multi-frame RL.

For both parallel implementations the entropy decoding had to be decoupled from the macroblock processing. For this the `H264mb` and `H264slice` structures are used. A `H264mb` structure holds the output of the entropy decoding for one macroblock. The number of `H264mb` structures needed is equal to the number of macroblocks in the frame. The `H264slice` holds the slice parameters of which only one is required. These structures

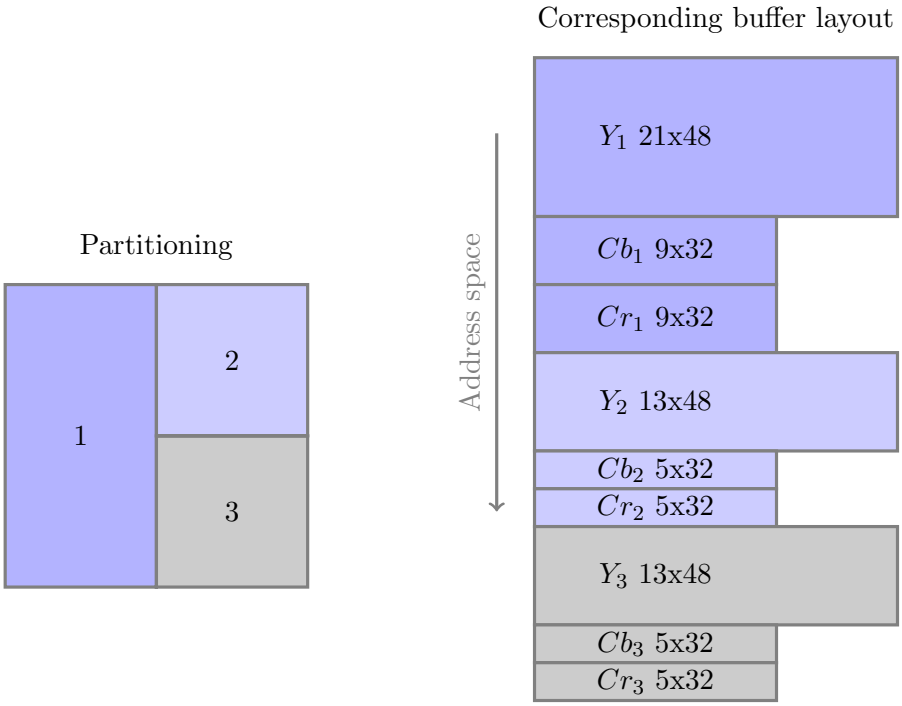


Figure 5.16: Motion data organization of the individual partitions in the `mc_ref` buffer.

are a subset of `H264Context` and only hold the necessary field to vastly reduce the memory requirements.

The role of the PPE is different for TP and RL. In TP the PPE has the task of the server, which provides synchronized access to the shared structures. The server assigns the work units(MB) to the SPEs and also updates the shared structures after a SPE has processed a work unit. For the messaging the fast mailbox facilities are used. In RL the PPE only needs to start the SPEs for each slice. In RL there is are no shared structures as it uses a distributed control mechanism. This is implemented using barrier DMA transfers to the target SPE. The only required information to satisfy the line dependencies is the number of macroblocks it has processed in total.

The main challenge of programming the Cell platform is making use of the memory hierarchy. The SPEs do not feature the memory abstraction of traditional cache-based processors. Instead, they can only operate on data residing in their local stores. Communication with the external memory is explicit via the DMA unit. Since the local store size is limited, the required data for the macroblock kernels is carefully studied. It is revealed that the intra prediction and deblocking filter required intra data from the current frame, while the motion compensation required motion data from up to 16 reference frames as. The relevant part of the frames, however, is small and could be transferred instead of the entire frame. The macroblock kernels required some adjustments to work on a smaller parts of the frame.

In TP the DMA transfers are sequential to the computation. The intra data is first brought in by a transferring a 48x20 frame block surrounding the current macroblock.

The block size is chosen so that it contains the intra data, while abiding to the DMA restrictions. This data is transferred to the working buffer on which the macroblock kernels are applied. For the motion compensation every partition requires different motion data pointed to by the motion vectors. This is brought in before processing each partition. After the kernels are applied the working buffer is written back to the frame. Due to DMA alignments and size restriction and unpredictable macroblock assigning more data has to be written back than necessary. This causes possible overlapping frame writes for concurrently processed macroblocks. The solution is to increase the horizontal macroblock spacing from two to three and reduce the stride of uneven chroma blocks from 32 to 16. For each macroblock a frame block of 48x19 is written back for luma. For chroma this alternates between 32x9 and 16x9 for macroblocks with even and uneven x-coordinates.

However, the increased macroblock spacing decreases the macroblock parallelism. While the maximum parallelism has decreased by 1/3, the impact on the scalability is small. Simulation results indicate the maximum speedup decreased only 21% for FHD. This can explained by the fact that the average parallelism has only decreased by 21%. Decreases in the maximum parallelism increases the duration this lower maximum can be maintained. For 16 processing elements the speedup only dropped from 15x to 14.5x.

In RL the DMA transfer latency is hidden behind the computation. This is possible due to the predictable nature of RL. The intra data is pre-sent to the target SPE and, therefore, also kept on-chip. This conserves a lot of external memory bandwidth. A line buffer is used to act both as the target for the pre-sent intra data and as working buffers. Using a line buffer has advantages as no additional local store copy steps are required. The motion data is pre-buffered to the local store in a double buffered fashion. For this the communication had to be decoupled from the computation in the motion compensation kernel. In the pre-buffering scheme the entire motion data of the next macroblock is transferred to the local store, while processing the current.

In the write back step a pair of two adjacent macroblocks is send back to the frame. The intra data is also pre-sent in this step. The combined write back is a result of the DMA restriction and a memory bandwidth optimization. Due the alignment and size restrictions the 8x8 chroma components cannot be written by themselves. In RL the write back can be delayed after the adjacent macroblock has been decoded, since the entire line is decoded on the same SPE. However, the write back is delayed another slot, since the processing of the previous macroblock has not completed until the deblocking filter of the current macroblock is applied. For the same reason only the top 16 lines of the transfer buffer is written back to the frame, while the lower 4 are sent as intra data to the target SPE. In RL, the write back only writes each pixel of the output frame once. In TP this is written three to six times.

However, delaying the write back has the same effect as increasing the spacing to four in TP. The simulation results presented a drop of 30% in maximum speedup for FHD sequences, which is in line with average parallelism drop. For 16 processing elements the effects were also minimal. For FHD the speedup dropped from 12x to 11.7x. However after this point the scalability was quickly saturated.

In the next chapter a static resource analysis of the two implementation is provided. This includes among other the memory size and bandwidth requirements. The TP

implementation requires less local store memory, while RL is more efficient in terms of memory bandwidth. Also a more local store efficient buffer strategy is presented, which allows the RL to scale to Super HiVision resolutions with the current SPE local store size. In Chapter 7 the experimental results are presented. These results are analyzed and compared to the theoretical values of the reduces parallelism presented in this chapter.

6 H.264 Resource Analysis

In Chapter 5 the Cell processor implementation of the Task Pool (TP) and Ring-Line (RL) parallel strategies have been revealed in detail. Before presenting the performance results in Chapter 7, the resource requirements of the parallel strategies are analyzed. This allows for better interpretation of the performance results.

We start with analyzing the HDvideobench sequences, which are used as benchmarks in Chapter 7. The time spent in the entropy decoding and the macroblock kernels are measured. For the timings the sequential version of the algorithm is used. Similar work has been done by Alvarez et al. [3], but on different target platform. This is followed by investigating the additional memory usage of the two parallel implementations. More specifically, the size of the additional structures for decoupling the entropy decoding and the local store memory footprint is investigated. Finally, an analysis is given of the memory bandwidth requirements for both parallel implementation. In the previous chapter it was noted that the bandwidth requirement for Ring-Line were lower than for the TP implementation. This is investigated in detail, since it provides the insight in what manner the memory bandwidth is the bottleneck of the performance.

6.1 Kernel Profiling

In this section the time spent in the entropy decoding and macroblock kernels is investigated to provide insight on what the ratio is between the entropy decoding and macroblock processing. While we have limited ourselves in investigating the macroblock processing, it is still desirable to see what part this is of the entire application. The timings of the macroblock processing also provide a good view of the size of the work units. The ratio between the size of the work unit and the synchronization overhead determine the scalability. Furthermore, the macroblock timings are also needed to determine the effectiveness of hiding the memory latency in RL.

The results are obtained by running the sequential FFmpeg executable on the test platform described in Section 3.2, which implies that only the PPE is used. The executable was built with the FFmpeg optimizations and Altivec support enabled. The test sequences are the HD and FHD versions of the BlueSky, Pedestrian, RiverBed and RushHour videos from the HDVideoBench. Table 6.1 lists the timing results for the FHD sequences. The HD sequences provided almost identical results and are therefore not presented.

The table shows the average time spend in the specific kernels per macroblock. The average total macroblock time is between 10 and 20 us. The sequences BlueSky, Pedestrian and RushHour show similar results. The time spent in the entropy decoding (CABAC) is around 25%. BlueSky does a bit more motion compensation, while Pedestrian is a little more intra-prediction orientated. RushHour is in between.

Kernels	FHD Bluesky		FHD Pedestrian		FHD RiverBed		FHD RushHour	
	μs	%	μs	%	μs	%	μs	%
Cabac	3.05	23.00	3.02	27.69	7.40	39.95	3.12	26.32
Intra	0.06	0.46	0.27	2.44	1.17	6.30	0.15	1.27
Motion	6.33	47.82	3.61	33.08	1.56	8.43	4.43	37.41
IQ + IDCT	0.41	3.10	0.48	4.45	1.82	9.84	0.39	3.32
Deblock	2.82	21.27	2.80	25.68	5.34	28.82	3.11	26.28
Other	0.57	4.34	0.73	6.66	1.23	6.65	0.64	5.41
Total	13.24	100.0	10.91	100.0	18.52	100.0	11.85	100.0

Table 6.1: Average macroblock kernel times of the FHD HDVideoBench sequences.

RiverBed, however shows a completely different profile. All the kernels except for the motion compensation need a lot more time. Riverbed is a highly predictive sequence and relies heavily on the intra-prediction. In Section 6.3 the exact ratio of intra and motion prediction is investigated for determining the memory bandwidth. While intra-prediction requires less work than motion compensation, the time spend in the entropy decoding increases.

Regular H.264 streams are more like the other three sequences. The intra prediction, inverse quantization, and IDCT only take a fraction of the total time. The "other" kernel contains overhead and the border exchange, and is also only a small part. The ratio of the entropy decoding to macroblock processing is about 1 to 3. If we assume the SPEs are as fast as the PPE, multiple entropy decoders are necessary to balance the system.

6.2 Memory Requirements

In this section the memory usage requirements for the parallel implementations are investigated. The differences with the original sequential algorithm are determined. This includes the extra structures for the decoupled entropy decoding in the external memory and the local store usage.

6.2.1 External Memory Requirements

Decoupling the entropy decoding requires to store all its results in `H264mb` structures for each macroblock in a frame. Each `H264mb` structure has a size of 1.9-2 kB. Additionally, each slice has a `H264slice` structure of 12 kB. However this can be neglected as it is only needed once for each slice. For single-sliced frames, a total of 15.5 MB is needed for a single FHD frame to store the entropy data in work units. For QHD this goes up to 62.2 MB and for Super Hi-Vision resolutions this goes up even further to 249 MB.

This memory size poses a limit on the number of frames in flight. Each frame in flight requires a separate matrix of `H264mb` structures to store the entropy data. The memory size requirements could limit the parallelization of the entropy decoding. The higher the resolution the higher the need for having multiple frames in flight in order

to speed up the entropy decoding. The memory requirements increase quadratically for higher resolutions. Therefore, parallelizing the entropy decoding for future H.264 standards still remains an open topic. At the same time this also sets a practical limit to 3D-Wave parallelism [18], which requires tens of frames in flight to fully exploit the available parallelism.

To reduce memory requirements for parallelizing the entropy decoding for future high resolution H.264 standards, it might be wise to allow slicing for only the entropy decoding part. For example by slicing the QHD frames in 4 rectangles allows it to be parallelized with 4 processing cores, while only requiring the memory needed for 1 frame. In contrast to regular slices, this will not require a additional deblocking step after the macroblock processing. The sacrifice in compression rate by having additional slice headers can be neglected.

6.2.2 Local Store Requirements

In this section the local store memory requirements is investigated for both the TP and the RL implementation, discussed in Chapter 5. Each SPE has a local store with a size of 256 kB. The local store is shared between the SPE program image and local variables. First, the program image is depicted, followed by the local variables. Table 6.2 shows the total size and partitioning of the SPE images of both parallel implementations. The code is compiled with the `-O2` optimization level, which strikes a balance between performance and image size. The same setting is used for obtaining the performance results in Chapter 7.

Object file	Task Pool		Ring-Line	
	bytes	%	bytes	%
dsputil_cell.o	64986	56.1	64336	54.3
h264_decode_mb_spu.o	8192	7.1	7680	6.5
h264_filter_spu.o	5384	4.6	4816	4.1
h264_idct_spu.o	1880	1.6	1880	1.6
h264_intra_spu.o	28440	24.6	28312	23.9
h264_main_spu.o	384	0.3	2896	2.4
h264_mc_spu.o	6376	5.5	8376	7.1
ff_h264_spu	115824	100.0%	118496	100.0%

Table 6.2: Local store usage - SPE program image size of TP and RL.

The two SPE program images are about the same size. The RL image is a little larger due to the extra code for implementing the pre-buffering of the motion data and the distributed control. The program images are quite large as they occupy almost half of the local store. The object files `dsputil_cell.o` and `h264_intra_spu.o` are mostly responsible for this. The object file `dsputil_cell.o` consists mainly of functions used in the motion compensation kernel, while `h264_intra_spu` contains the intra prediction functions.

The SPE programs are about equal in size. This does not, however, apply to the size of the local variables. RL requires more local store space for implementing the pre-sending and pre-buffering schemes discussed in Section 5.4.2 and 5.4.5. With the implemented RL buffer scheme it is not possible to scale to resolutions beyond FHD. The main goal of the implementation is to extract as much performance as possible, while abiding the standard. In Chapter 5, it was promised that we would introduce a more balanced approach in this chapter for scaling beyond FHD. Table 6.3 depicts the local variable sizes for TP and RL as well as the to be elaborated balanced RL approach.

Local store	Task Pool	Ring-Line	Ring-Line balanced
Working buffer	1600	57600	11520 +1600
Edge data	-	3888	3888
Motion data			
-(pre)buffer	1584	39936	30720
-biweight	1280	1280	1280
-edge extend	15840	15840	15840
Write-back	320	960	960
DMA list elements	504	4160	4160
H264Context_spu			
-H264mb	2000	5712	5712
-H264slice	12624	12624	12624
-H264mc	-	6280	6280
-other	688	1048	1048
Other H.264 related	3004	3460	3460
Total	25188	138532	84836

Table 6.3: Size in bytes of the data structures of Task Pool, Ring-Line, and the balanced Ring-Line implementation.

The size difference between TP and RL is quite high for the local data structures. Adding the SPE program image to the local data variables shows that the RL implementation uses almost all of the available local store. The stack is also allocated in the local store.

In contrast, the local store usage of TP is quite economical. Furthermore, the size of the TP structures stays constant for larger resolutions. For RL, however, the working buffer and the edge data need to scale with the width of the video sequence. In Section 5.4.2 it was revealed that to optimize performance, the RL and working buffers were

combined. Furthermore, buffer slots equal to the number of macroblocks in an entire line are allocated in the local store. Due to local store size limititations, only resolutions up to FHD can be supported. For QHD the working and edge buffers need to be doubled in size. This would require an additional 57600+3888=61488 bytes of local store space, which simply is not available.

When considering the balanced RL approach, the memory footprint is a lot smaller. Around 53 kB less local store space is needed to enable the same support for FHD as in the implemented RL. However, the reduced memory requirement comes with a performance cost. In the balanced RL approach, the working buffers and the RL buffer are separated. In Section 5.4.2 it was discussed that when combining the working and RL buffers, additional copy steps are avoided. By separating the two buffers they are reintroduced. The balanced buffer strategy can be best explained by looking at Listing 6.1.

Listing 6.1: Local store buffers of the balanced Ring-Line approach.

```
1 //Ring-Line buffer - statically allocated for up to 1920 width video
2 DECLARE_ALIGNED_16(uint8_t, rl_y_ls[120*16*4]);
3 DECLARE_ALIGNED_16(uint8_t, rl_cb_ls[120*8*2]);
4 DECLARE_ALIGNED_16(uint8_t, rl_cr_ls[120*8*2]);
5
6 //Working buffer
7 DECLARE_ALIGNED_16(uint8_t, dest_y_ls[48*20]);
8 DECLARE_ALIGNED_16(uint8_t, dest_cb_ls[32*10]);
9 DECLARE_ALIGNED_16(uint8_t, dest_cr_ls[32*10]);
```

The balanced approach still maintains the idea of allocating buffer slots for an entire line. Because of this no backwards communication to the source SPE is necessary. When comparing the buffers to the ones in Listing 5.12, we see that the balanced RL buffers are 5 times smaller than the implemented working buffers. In this buffering scheme only the intra data is put in the RL buffer, therefore only 4 and 2 lines are needed for luma and chroma respectively. The intra data has to be copied from the RL buffer to the working buffer before processing. Also in the working buffer data must be shifted by one block before processing the next macroblock. The kernels are then applied to the working buffer in the same fashion as in TP. Based on the number of copy operations, it is expected that performance loss incurred by the additional copy steps is within 5%.

In the balanced approach scaling to QHD only requires an additional 11520+3888=15408 bytes. With this approach it is therefore possible to even scale to Super HiVison (8k x 4k) resolutions with current SPEs. Scaling further with equal local store space is not possible without applying the RL buffer relations discussed in Section 4.2.3.3.

From Table 6.3 it is also seen that the motion data pre-buffer reduced in size from 39936 to 30720. This can be accomplished by modifying the motion compensation functions to enable processing on luma blocks with a stride size of 32. The luma buffer has a static width of 48 pixels to contain the widest motion data block of 21x21. However, when the highest amount of motion data is required the luma blocks are all 9x9. Since this can only cross one 16-byte alignment border, only a motion data block with a width of 32 bytes is required. This has not been implemented to keep down the coding effort.

When properly implemented only a few additional conditions need to be checked, and it is expected that the performance loss is negligible.

6.3 Bandwidth Requirements of Macroblock Decoding

In Chapter 5 the Cell implementations of TP and RL were discussed. It was concluded that the TP implementation required more external memory operations and bandwidth than RL due to two reasons. First, RL keeps the intra data on-chip by using local store to local store transfers. The second is that the write back step only occurs once for every two blocks, where with TP it has to be performed at the end of every macroblock. The analysis of the memory bandwidth requirements is neccesary to analyze the performance results presented in Chapter 7 in order to asses in what matter the memory bandwidth forms the bottleneck.

To investigate the memory bandwidth requirement, the video sequences must first be profiled. The bandwidth requirements depend on the frequency and mode motion compensation is required. More specifically, the bandwidth requirements depend on the motion compensation partitioning of the macroblock. The four FHD video sequences have been analyzed in this regard and the results are presented in Table 6.4.

Partitioning	List elements	BlueSky	Pedestrian	RiverBed	RushHour
Intra	-	569	1770	6769	1092
16x16	39	2400	3093	749	3522
16x8	23	451	110	457	263
8x16	39	201	67	119	128
8x8	23	898	277	626	371
8x4	15	70	1	4	3
4x8	23	49	1	1	2
4x4	15	68	0	0	1
Bi-weight					
16x16	78	3803	3033	89	3076
16x8	46	0	3	20	6
8x16	78	202	72	127	133
8x8	46	2862	267	128	435
8x4	30	0	0	0	0
4x8	46	0	0	0	0
4x4	30	0	0	0	0
Total list elements		579705	386789	82500	427626

Table 6.4: Motion compensation profile of the FHD HDVideoBench sequences. The table list the average occurrences per frame.

The table shows the average occurences of the different motion compensation parti-

tions per FHD frame. The BlueSky sequence requires the most motion compensation, while RiverBed primarily depends on intra prediction. The other two sequences are in between but tend more towards BlueSky.

Since the BlueSky sequence has the highest memory bandwidth requirements, its profile is used for further analysis. The profile should be seen as optimistic. The fact that B-frames uses more motion compensation than I/P-frames is neglected. Furthermore, the BlueSky numbers are an average over 100 frames.

The bandwidth requirements also have a static part. For TP the work unit and intra data need to be brought in for each macroblock. The unfiltered borders and resulting picture block need to be written back. These memory operations are static for each sequence as its size and occurence rate is the same for each macroblock.

Table 6.5 shows the total number of DMA transfers and the relative load on the memory bandwidth for the TP implementation. For the list elements this is determined by using the results of Table 3.2, which lists the number of list elements per second. Since all the transfers for TP have a width of either 32 of 48 byte the average is used, which equates to 116 M list elements per second.

The work unit and unfiltered borders are sequential DMA transfers. It was shown that the characteristics for this type of transfer is very different from the list DMA. This needs to be compensated by reducing the its weigth accordingly. For the sequential DMA the base is set to the results obtained in Section 3.3.1. For both cases the results obtained from the experiments with 16 SPEs are used.

List DMA	List elements	Occur/frame	Total elements	Rel. BW load
Get intra data	40	8160	326400	0.28%
Write back	37	8160	301920	0.26%
Motion data	(See Table 6.4)		579705	0.50%
Sequential DMA	DMA size	Occur/frame	Total size	Rel. BW load
Put unfiltered borders	32	24480	765 kB	0.07%
Get work unit	2000	8160	15.6 MB	0.07
Relative bandwidth load per frame per second				1.18%
Maximum frames per second				84.7 fps

Table 6.5: Memory subsystem requirements for FHD BlueSky using the Task Pool implementation.

In the table the steps involving list DMA and sequential DMA are separated. The total elements presented the number of list elements transferred per frame. The table shows that most of the bandwidth is used for list DMA operations. The highest frame rate before the bandwidth becomes the bottleneck is 84.7 fps. While this is sufficient for any FHD sequence, it quickly becomes the bottleneck when moving to higher resolutions or frame rates. For example, QHD has four times the number of macroblocks and, therefore, requires four times as much bandwidth. Using TP for QHD sequences would

result in a maximum of 21.2 fps, which is not sufficient for smooth playback. As stated earlier, the used profile can be considered optimistic for BlueSky as unbalance on the memory subsystem load is not considered. In addition the ramping of TP adds to this. Furthermore, the results of Table 6.5 only consider the macroblock processing part of H.264. The entropy decoding also requires memory bandwidth. In our implementation of TP and RL the entropy decoding is done before the macroblock processing. Therefore, it does not cause additional contention on the memory subsystem. However in a real application, the entropy decoding will run concurrent with the macroblock processing. In this case additional memory bandwidth is required.

The RL implementation requires less memory bandwidth. By keeping the intra data on-chip and having a combined write back step, the number of DMA list transfers decreased significantly. In Chapter 3.2 it was revealed that the on-chip bandwidth was a factor 10x higher than the memory bandwidth. Therefore, the on-chip bandwidth is not a bottleneck. The bandwidth requirements for RL are investigated and presented in Table 6.6. For calculating the relative bandwidth load the results from Table3.2 were used. As base for the write back step, the average of the results of the list element width 16 and 32 are used. In RL the write back step only uses these two widths. The average equals 132.5 M list elements per second. For the motion data the same base is used as in TP, which equals 116 M list element per second.

List DMA	List elements	Occur/frame	Total elements	Rel. BW load
Put intra data	8	8160	65280	-
Write back	32	4080	130560	0.10%
Motion data	(See Table 6.4)		579705	0.50%
Sequential DMA	DMA size	Occur/frame	Total size	Rel. BW load
Put unfiltered borders	32	8160	255 kB	-
Get work unit	1904	8160	15.5 MB	0.07
Relative bandwidth load per frame				0.67%
Maximum frames per second				149.2 fps

Table 6.6: Memory subsystem requirements for FHD BlueSky using the Ring-Line implementation.

With RL the intra data and unfiltered borders are not transfered via the external memory and have no contribution. Also the write back requires far less memory bandwidth than is the case with TP. The total requirements of the memory subsystem per frame is 0.67%, which is 43% lower compared to the bandwidth requirements of TP. With less motion compensation oriented sequences, the relative difference is even higher. The reduced load on the memory subsystem results in a 76% increase of the maximum frames per second. This is considerable since the bandwidth is sufficient to support QHD sequences with RL. Again these numbers should be regarded as optimistic figures for the BlueSky sequence.

6.4 Conclusions

In this chapter several investigations have been performed. First, the relative computational requirements of the entropy decoding, and macroblock processing have been analyzed for the HDVideoBench sequences. This revealed that in most cases the entropy decoding, motion compensation and deblocking filter require the most computation. The ratio of the entropy decoding to the macroblock processing time is found to be around 1 to 3.

Second, the memory usage of both the external memory and the local store are investigated. This revealed that for each frame in flight, 15.5, 62.2, and 249 MB are required for respectively FHD, QHD and Super Hi-Vision sequences. The memory requirements limits the amount of parallelism that can be exploited by the 3D-Wave.

Third, the memory bandwidth requirements are investigated. This revealed that the more efficient RL implementation could scale up to 149.2 fps before you hit the bandwidth wall. For TP this limit is reached at 84.7 fps. These numbers correspond to the FHD BlueSky sequence. Other sequences require less motion compensation and have a higher limit.

In the next chapter we will investigate the actual performance of the parallel implementations. The results obtained in this chapter are used in the analysis of the performance results. This is especially the case for the analysis regarding the memory bandwidth requirements, performed in Section 6.3. In Chapter 8 the resource analysis is used in projecting the hardware requirements of future applications of parallel H.264 decoders.

Experimental Results and Analysis

7

In Chapter 4 the theory behind two parallel H.264 macroblock decoding strategies were discussed in detail. The first one was the Task Pool (TP) strategy, which builds on a centralized worker-server model. The second one was the novel Ring-Line (RL) strategy, which builds on a data flow like principle. The theoretical analysis, which assumes perfect conditions, revealed that the TP strategy has better scalability. Following in Chapter 5 the two strategies are implemented. Here it is revealed that RL maps better on the Cell architecture than TP, since RL allows for concurrent communication and computation. Therefore, a better performance per core is expected. In Chapter 6 it was revealed that RL has 1.75x to 3x lower memory bandwidth requirements compared to TP.

In this chapter the actual performance of the TP and RL implementations is investigated. First, we present the experimental results of the two implementations using the HDVideoBench sequences. Thereafter, the obtained performance is analyzed. The differences between the theoretically expected and the actual performance is explained through further analysis. The analysis allows us to project the performance and scalability on future many-cores. Finally, also the performance and efficiency of the Cell processor using the parallel H.264 strategies is compared with modern state-of-the-art x86-processors from Intel and AMD.

7.1 Experimental Results

In this section the two H.264 parallel implementations are benchmarked. The test platform is the Cell Blade which consists of two Cell processor. A total of 16 SPEs is usable. The FHD sequences of the HDVideoBench are used as input. For more details, see Section 3.2.

The FFmpeg binaries for TP and RL are build on our development platform, which is a PS3 running Yellow Dog Linux 6.2. The FFmpeg build is configured with code optimizations enabled and makes use of the PowerPC Altivec extensions. The optimization flag for the code running on the PPE is -O3, while the SPE code is compiled with -O2 for a good balance between speed and size.

In the benchmarks only the performance of macroblock decoding is measured. In both implementations this is separated from the entropy decoding, which is performed before the macroblock decoding. The performance metric used is the average frames per second for decoding the 100-frame sequences.

The time spent in the macroblock processing is measured once a frame. This is realized by placing the timer readouts around the equivalent of the simplified code fragment of Listing 5.3 for TP and Listing 5.9 for RL. The PPE hardware timer is used for the measurement. The timer has a resolution of 14.8 MHz on the Cell Blades. The command used to run the FFmpeg program is:

`./ffmpeg -threads` *p* `-i` *input*`.h264 -y` *dropoutput.yuv*
where *p* ranges from 2 to 17 indicating the number of SPEs+1, *input* is the input sequence and *dropoutput.yuv* is a symbolic link to `/dev/null`.

The average performance in frames per second using the FHD HDVideoBench sequences are presented in Figure 7.1.

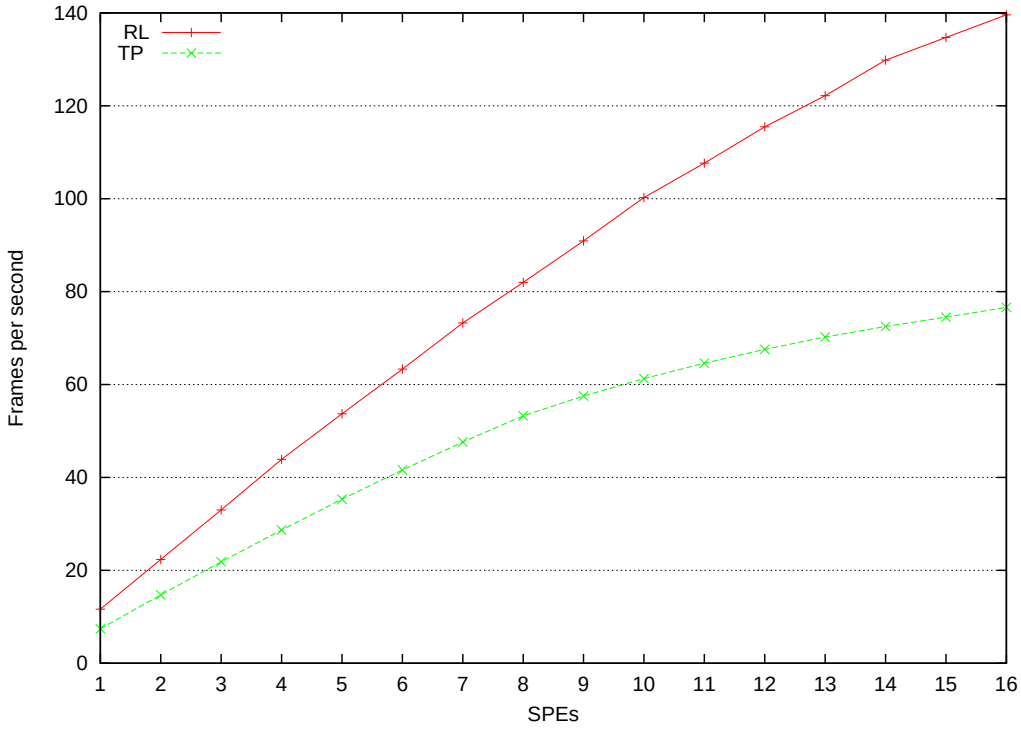


Figure 7.1: Average performance in frames per second of the HDVideoBench FHD BlueSky, Pedestrian and RiverBed sequences.

The results show that both the TP and RL implementation show scalability. Using more SPEs increases the frames per seconds over the entire range. The performance of RL is between 1.6x to 2x higher than the performance of TP. For TP the maximum attainable fps is 160 and 82 for HD and FHD respectively, while RL scales up to 300 and 157 fps.

The RL implementation running on 16 SPEs has sufficient processing performance to perform the macroblock processing of QHD sequences in real-time. Since QHD sequences have 4 times as many macroblocks per frame, the projected performance is 31 to 40 fps. The TP implementation is not capable of this. In terms of performance the RL implementation is clearly superior.

The performance on a single SPE, which is shown in Figure 7.2, is also much higher for RL. The difference is mainly caused by having concurrent communication and computation. The figure also shows that the single SPE performance of RL is close to the performance of the PPE running the original sequential FFmpeg. In a single case (BlueSky FHD), it matches the attained frames per second. This is quite good consid-

ering that in terms of chip area the PPE is about twice as large as a SPE.

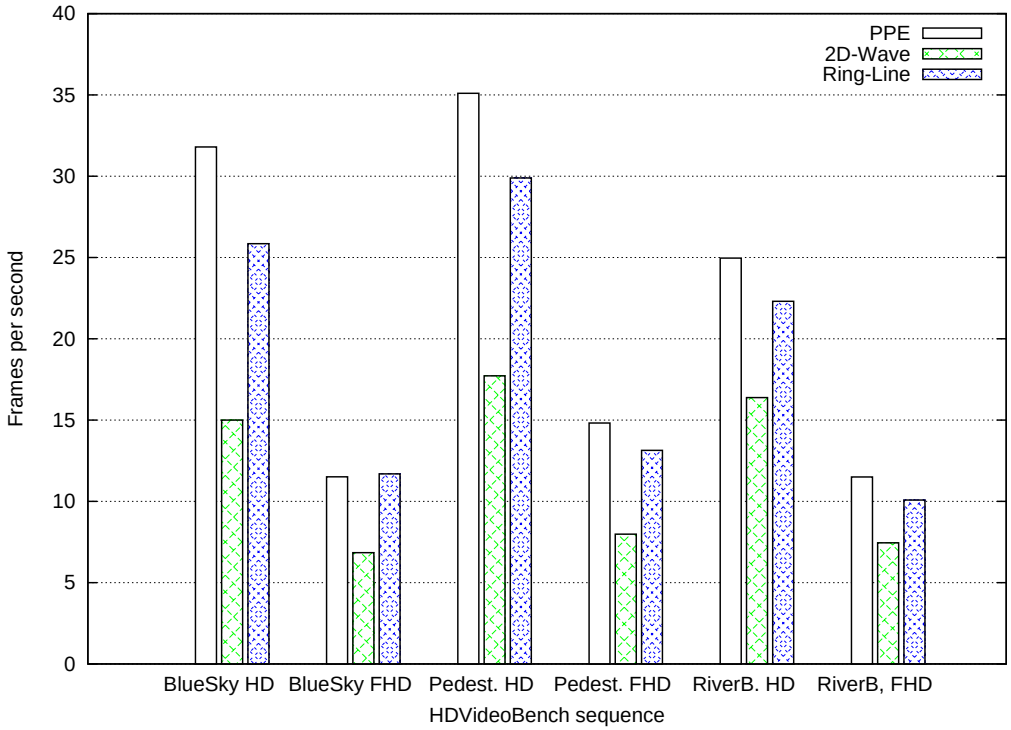


Figure 7.2: Single core PPE sequential, Task Pool and Ring-Line frames per second.

For the HD sequences the scalability seems to be almost at its limits for both implementations with 16 SPEs. For the FHD sequences the scaling is also leveling off, but has not reached its maximum yet. It should also be noted that the two implementation have a different preference for the video sequences. Ordered in terms of performance from high to low for RL they are Pedestrian, BlueSky and RiverBed. TP performs worst with BlueSky and approximately equally with Pedestrian and Riverbed.

The experimental results show that the actual scalability of TP is worse compared to RL. This could be seen as a surprise as in Chapter 4 the theoretical scalability of TP is shown to be superior. In the next section we investigate the deviation of the obtained results with the theoretical expectancy.

7.2 Performance Analysis

In the previous section the experimental results were presented and several observations were made. The most important was the fact that scalability of TP is lower than RL. While also the performance of RL is higher than TP, it can be explained by the fact that RL maps better on the Cell architecture. Therefore, the performance per SPE is higher and in turn translates to better overall performance. Explaining the attained scalability, however, requires further investigation.

In Section 4 the theoretical analysis of TP shows a scalability of 15 with 16 processing elements, for the FHD BlueSky sequence. From the experimental results only a scalability of approximately 10 is observed. The TP implementation discussed in Section 5.3 revealed that the additional spacing incurred a scalability loss. However, the impact is also investigated in Section 5.3.4.3. The scalability for the BlueSky sequence only dropped slightly to 14.5. In this section we mainly focus on the differences (and the lack of difference) in practical and theoretical scalability of both implementation. In the course of this investigation the other observations are also explained. These include the sequence preference, the attained frames per second and, the performance differences with a single SPE and PPE.

First, the practical and theoretical scalability is compared to have a detailed view of the impact over the full range of 1 to 16. This is performed for both TP and RL. The theoretical scalability is defined as the scalability with perfect platform conditions, e.g. zero communication delay and infinite bandwidth. Following, to find the cause of the additional inefficiency, the code is profiled to uncover the bottlenecks when scaling up the SPEs. After the bottleneck are identified these are discussed further and finally the several conclusions about the performance and scalability are drawn.

7.2.1 Practical vs. Theoretical Scalability

In this section, as the first step in tracing back the scalability loss, the exact differences of the practical and theoretical scalability are investigated. In Chapter 5 it was revealed that the Cell implementation of the TP and RL strategy reduced the theoretical parallelism. For the sake of determining the platform-specific efficiency losses, the simulator has to mimic the exact scheduling behavior of the implemented solutions. Figure 7.3 shows the differences between the actual scalability and the expected scalability obtained via the simulator. In Figure 7.4 the actual scalability is normalized to the expected scalability.

The figure shows interesting results. First, the normalized efficiencies of RL are actually higher than 100%. This implies that the scalability of the implemented RL is higher than theoretically possible, which should not be possible. The difference originates from the method of obtaining the theoretical limits. The input of the simulator are the macroblock execution times obtained by timing the them on the TP implementations using a single SPE. From the results can be derived that the executions times are less variable in the RL implementation, resulting in less efficiency loss due to dependency stalls. The fact the normalized efficiency drops a little when approaching 16 SPEs, implies that the scaling is actually is a little lower than theoretically possible at the end. Still the scalability is impressive as it manages to stay approximately the same as the theoretical limits.

The normalized efficiency of TP is also very interesting. The results show that TP does not manage to scale the same as theoretically expected. In the theoretical results the platform-specific effects are not taken into consideration. The difference between the theoretical and practical results are caused by these effects. A good example of such an effect is increasing synchronization overhead.

When looking at the TP plot lines, two stages can be identified. In the first stage, up

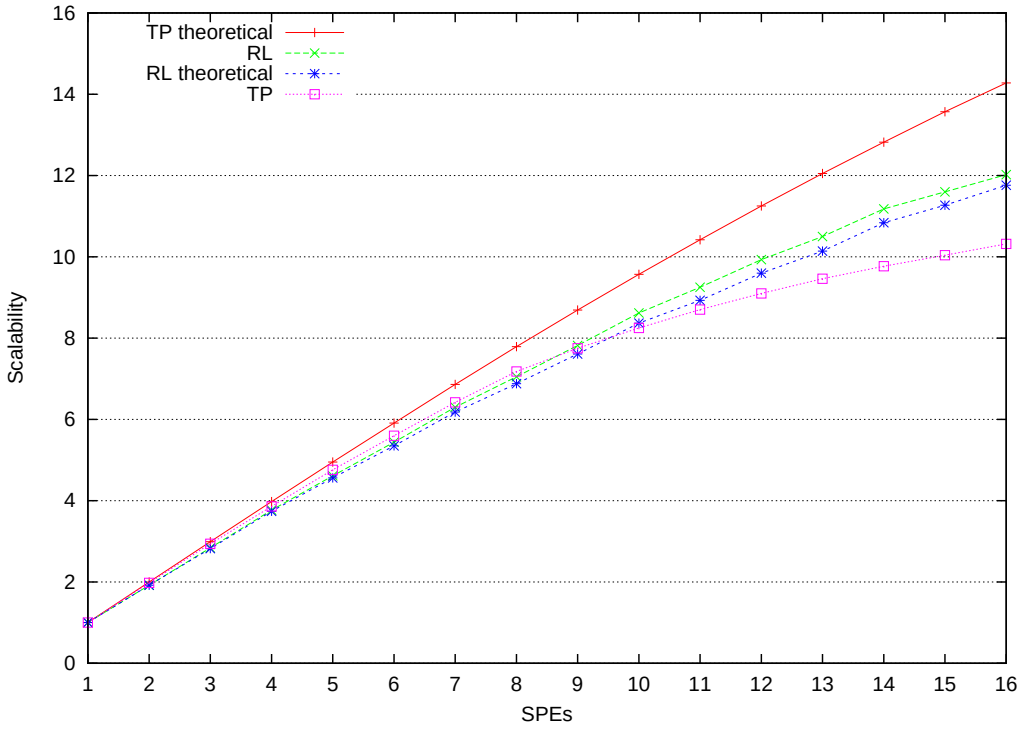


Figure 7.3: Average scalability of the TP and RL implementation compared to the theoretical scalability results obtained using the simulator.

to 8 SPEs, the rate of decline is smaller than in the second stage ranging from 8 to 16. In both stages the rate of decline with respect to the normalized efficiency, is approximately constant. The slope change is caused by the additional latency incurred by using off-chip SPEs. As shown in Figure 3.3 the average latency of a mailbox round-trip increases from 0.41 us to 0.86, when moving from 8 to 16 SPEs.

The initial decline is caused by two platform-specific effects, both contention related. First, there is a additional contention at the synchronized access via the PPE. With increasing number of SPEs the average synchronization latency increases and therefore also the average macroblock execution time.

The second effect is contention on the external memory access. With additional SPEs the average time spend waiting for a DMA operation to complete is higher when several SPEs are trying to do the same.

With RL these problems are avoided with the distributed control, SPE-SPE communication, and the concurrent communication and computation.

With this the causes of the scalability differences are identified. However, it is unknown what the relative contributions are of each effect. It could be the case that the contention on accessing the shared structures is mostly the cause and the memory contention is insignificant, and vice versa. Therefore, the exact contributions of the platform-specific effects on the scalability loss are investigated.

To investigate the contributions of the effects, the profiling divides the execution

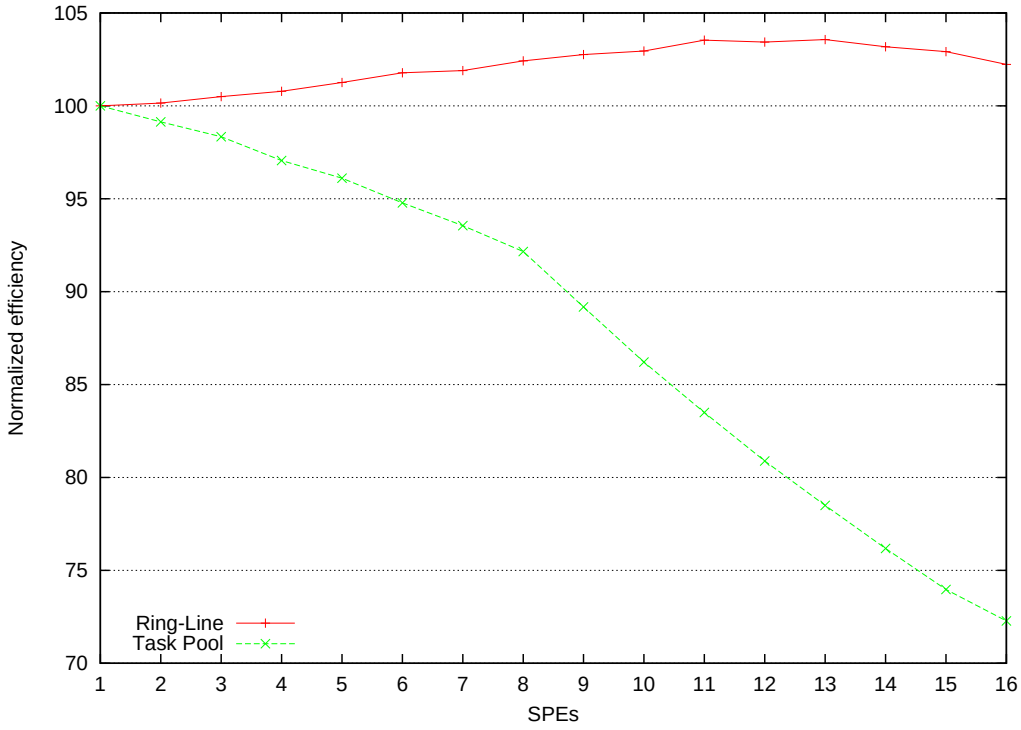


Figure 7.4: Normalized efficiency of practical to theoretical scalability of FHD sequences.

times in four parts. These parts are the actual macroblock processing, DMA steps, synchronization latency and dependency stalls, and the ramping stalls. The profiling results are shown in Figure 7.5 and 7.6 for TP and RL, respectively. The FHD BlueSky sequence is used in both cases. It was not possible to separate the dependency stalls and synchronization latency in the profiling. However, the synchronization latency can be deduced by using the RL results of this part. Since RL follows the theoretical scalability, the synchronization latency contributions is has to be constant for all number of SPEs. In the results for 1 SPE it is seen that the contribution of this step is zero and, therefore, it can be assumed that the increase with more SPEs in this part is entirely due to dependency stalls.

The figures show interesting results and mostly confirm the expected behavior of both implementations. In both cases the macroblock processing part is constant and about the same. The relative time spent in ramping stalls is also within the expectation described in Chapter 4. The other two steps are clearly the cause of the observed performance differences.

The first to notice is the difference in the DMA step contribution. One of the advantages of RL is that the memory latencies can be hidden behind the computation. This clearly has its contribution to its performance advantage. For TP the DMA step is relatively large. Furthermore it also increases steadily when scaling the number of SPEs. With more than one SPE, contention on the external memory access occurs.

The dependency stalls and synchronization latency is much higher for TP. This is

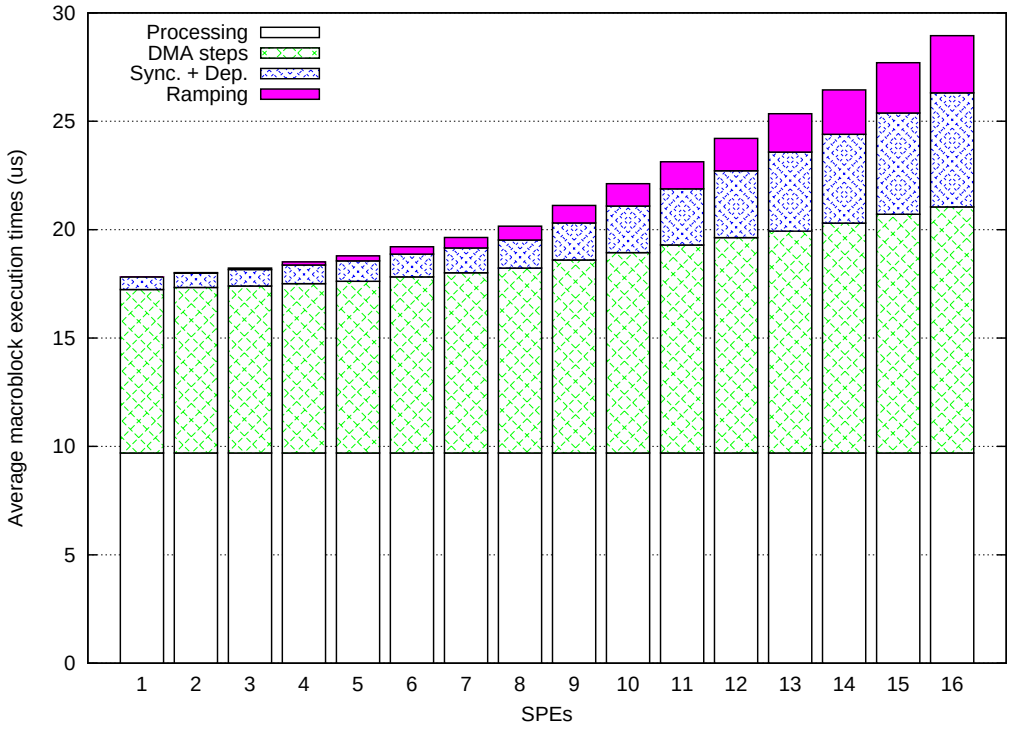


Figure 7.5: Breakdown of the average MB execution time for the Task Pool implementation using the BlueSky sequence.

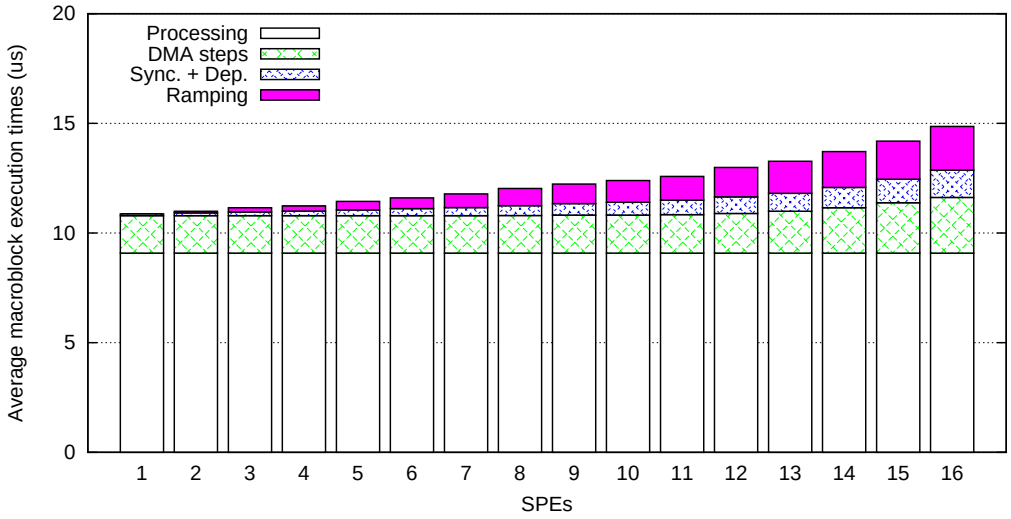


Figure 7.6: Breakdown of the average MB execution time for the Ring-Line implementation using the BlueSky sequence.

especially the case for more than 8 SPEs. As previously noted, this is partly caused by the off-chip latency. However, the increase in the dependency stall and synchronization latency implies that contention plays a significant role in the scalability loss.

In the next two sections the memory access contention and synchronized access contention is discussed in detail. This is followed by a conclusion on the scalability of the two parallel implementation.

7.2.2 Memory Access Contention

The DMA step consists of local store moves, creating DMA lists, issuing the DMA operation and waiting for their completion. With increasing SPEs the waiting for completion time increases, while the rest is constant. From the RL profile it can be seen that the DMA step is constant until around 12 SPEs, after which it increases a little. From this it can be derived that the constant part is around 1.7 us as generally RL does not have to wait for DMA completion.

When we look at the DMA step contribution of TP, it can be seen that it is much larger. In TP the time for the DMA step starts at 7.5 μs and goes up to 11.3 μs. As in RL this step also consists of a constant and variable part. In Section 6.3 it was shown that the number of memory operations of TP is around 1.6x higher than RL. Therefore, the static part of TP is approximated to 2.7 μs. This means that the dynamic part of the DMA step ranges from 4.8 to 8.6 μs. The time waiting for DMA completion has increased almost two-fold due to the memory access contention.

This, however, is not surprising when considering that in the same section it was found that 84.9 fps was approximated as the maximum frames per seconds. After this the memory bandwidth becomes a hard bottleneck. From Figure 7.1 it is seen that at 16 SPEs, the BlueSky performance is at 68.8 fps. This means that already 81% of the memory bandwidth is used. Furthermore, 84.9 fps is considered an optimistic limit.

Running into the memory bandwidth bottleneck also explains the fact that the DMA step increases for the RL implementation. After 12 SPEs the DMA step increases relatively aggressive. At 16 SPEs the DMA step has increased from 1.7 to 2.5 us. From the fact that this increases, we can derive that the memory latency cannot be completely hidden behind the computation. This can be viewed as surprising as RL uses less memory bandwidth and the average time that the DMAs have to complete is 9-10 us, which is the time of the macroblock processing.

In Section 6.3 it was shown that the maximum performance for the RL implementation is 149.2 fps. In Figure 7.1 it is shown that a performance of 136 fps is attained at 16 SPEs. This means that 91% of the optimistically approximated bandwidth is used. Considering this it could actually be seen as surprising that the DMA step has not increased more.

When we look at the profiling results of the other sequences in Table 7.1 and 7.2, the prior deduction can be confirmed. In Section 6.3 it is shown that RiverBed is much less demanding in terms of memory bandwidth. This is also less strongly the case for Pedestrian. Calculating the maximum frames per second for Pedestrian and RiverBed in the same fashion as for BlueSky, results in 98.5 and 132.8 fps respectively for TP. For RL this is 196.2 and 357 fps.

At 16 SPEs, the TP bandwidth usage is at 80% for Pedestrian and 62% for RiverBed. The bandwidth usage of the three sequences correlates well with the duration increase of the DMA step. At 16 SPES, the RL bandwidth usage is at 80% for Pedestrian and only at 35% for RiverBed. As expected the DMA step of RiverBed remains equal for all SPE configurations. For Pedestrian only a slight increase is seen at the end from 1.31 to 1.42 us. When comparing this to BlueSky, a similar increase can be noted at 13 SPEs, at which the performance is 121.4 fps. At this point the bandwidth usage is at a very similar 81%.

	Task Pool			Ring-Line		
SPEs	1	8	16	1	8	16
Processing	8.76	8.76	8.76	8.31	8.31	8.31
DMA steps	5.83	6.69	8.94	1.30	1.31	1.42
Dep + Sync	0.58	1.29	5.30	0.08	0.33	0.87
Ramping	0.00	0.57	2.08	0.00	0.65	1.59
Total(us)	15.17	17.31	25.08	9.69	10.60	12.19

Table 7.1: Profiling results of FHD Pedestrian.

	Task Pool			Ring-Line		
SPEs	1	8	16	1	8	16
Processing	11.96	11.96	11.96	11.81	11.81	11.81
DMA steps	3.61	4.00	5.15	0.76	0.76	0.77
Dep + Sync	0.58	1.32	5.18	0.10	0.31	0.58
Ramping	0.00	0.39	1.62	0.00	0.90	2.39
Total(us)	16.15	17.67	23.91	12.67	13.78	15.55

Table 7.2: Profiling results of FHD Riverbed.

7.2.3 Synchronized Access Contention

In the previous section the effects of the memory access contention were discussed. In this section we continue with the other cause of the TP scalability loss, namely the synchronized access contention. From Figure 7.5 it is seen that for TP the increase in dependency and synchronization stalls is as much contributing to the efficiency loss as the memory access contention. RL does not suffer from this and only shows increase due to dependency stalls.

In TP more SPEs causes the PPE to distribute work at a higher rate. Even if the PPE is able to handle the work distribution throughput wise the scalability still suffers due to contention effects. This is much like the memory access contention discussed in the previous section. Compared to RL, which features a distributed control, the increase in synchronization stalls is quite significant.

The distributed control of RL is only part of the difference. More importantly is that in RL the synchronization is one-way and non-blocking. It simply sends an incremented value to the target SPE. The contribution of the synchronization step in RL can be considered close to zero. For TP the synchronization is two-way and blocking. The SPE send a message to the PPE and has to wait for a response before continuing.

To make a calculated guess at the synchronization delays of the TP implementation, this is equaled to the difference between the dependency and synchronization stall of the TP and RL step. As stated earlier, separate measuring the dependency stalls and synchronization latencies is not possible due to practical reasons. The results are shown in Table 7.3.

SPEs	1	2	4	6	8	9	10	12	14	16
BlueSky	0.49	0.54	0.65	0.73	0.84	1.19	1.57	2.33	3.17	4.01
Pedestrian	0.48	0.55	0.69	0.82	0.97	1.32	1.73	2.57	3.44	4.43
RiverBed	0.48	0.55	0.69	0.84	1.01	1.43	1.86	2.72	3.68	4.61

Table 7.3: Approximated average synchronized access latency per macroblock of the Task Pool implementation.

The synchronization latencies are about the same for the three sequences. The results for Pedestrian and BlueSky are a bit higher. This is expected as the performance of these sequences is higher. With higher fps, the synchronized access request rate is also higher and in turn causes more contention. We also clearly see the a rapid increase for more than 8 SPEs. This is mostly caused by the fact that part of the SPEs is located off-chip. It is expected that if all the SPEs are located on a single chip the access latencies would have been around 2 us with 16 SPEs.

However, 2 us is still quite large as it is already at 20% of the processing time. This is simply too much for efficient scaling. In the current implementation of TP running on the Cell, the performance is for a great part bottlenecked by the memory subsystem. If this would be taken out of the equation and more SPEs were to be added, the synchronized accesses would quickly become the bottleneck. Also increasing the IPC of the current SPEs results in lower scalability. When the average macroblock time decreases, the average synchronized access request rate increases. This in turn will cause more contention and higher average access latencies.

In Section 3.3.3 the synchronization mechanisms were investigated and the fast mailbox was chosen because of its low latency characteristics. It can be concluded that the currently available synchronization mechanisms simply do not meet the requirements to provide a sufficiently scalable TP. Adding parallelism with 3D-Wave would not increase scalability as it is already bottlenecked by the centralized design.

7.3 Cell Efficiency Comparison

In the previous sections the performance of the two parallel implementations have been revealed. The RL implementation has proven to be the faster and more scalable solution.

	(Dual) Cell processor			X86 processor		
	SFRL 8	SFRL 16	MFRL 16	Pentium4	Core2 Duo	PhenomII x4
BlueSky	81.8	136.1	160.6	33.9	80.6	81.2
Pedestrian	92.9	156.9	186.1	36.8	83.1	85.8
RiverBed	71.3	125.8	150.7	23.5	51.2	61.2
RushHour	87.0	149.1	176.8	32.7	74.1	76.9
Average	83.2	142.0	168.6	31.7	72.3	76.3
Transistors	234M	468M	468M	125M	410M	758M
Fps/MTrans.	0.36	0.30	0.36	0.25	0.18	0.10
MB time(us)	11.8	13.8	11.6	3.9	1.7	1.6

Table 7.4: Performance in frames per second of macroblock processing using FHD sequences.

In this section the RL performance results of the Cell processor are compared to the performance results of several x86 consumer processors. For the comparison a Pentium 4 3.4 Ghz Prescott, Core 2 Duo E8400 and a Phenom II x4 940 are used. The exact specifications of the test platform can be found in Section 3.2.

As is the case for the Cell processor, the tests only measure the macroblock decoding times. For the three processors the original version of FFmpeg, prior to the parallelization changes, is used. Since the x86 platforms do not offer access to a hardware counter, the `clock_gettime` facility is used. This timer is not as accurate and low latency as the PPE and SPE timers. To compensate for this the average `clock_gettime` latency is measured and subtracted from the final results for every time it is used.

The FHD sequences of the HDVideoBench are used in the test. The sequences are encoded to contain only one slice per frame. Since FFmpeg only has support for slice parallelism, the processors cannot benefit from there multi-threading capabilities. For the sake of a clear comparison, the presented results of the processors are normalized to 3.2 GHz. Since the processors all run close to 3.2 GHz the inaccuracy is minimal. Table 7.4 shows the performance results.

In the table the results for the (regular) single-frame RL (SFRL) and multi-frame RL (MFRL) are listed with 8 and 16 SPEs. Since the MFRL is not implemented, the results are extrapolated from the SFRL results. This is done using the scalability analysis of Chapter 4. The scalability difference of SFRL and MFRL is simply factored in.

The table shows that the Cell processor using the RL algorithm is clearly the most efficient solution. The comparison is not completely fair, since the Core2 Duo and PhenomII x4 are multi-core processors. However, parallelizing the macroblock processing is a similar way as TP or RL for the x86 homogeneous CMPs is projected to provide little speed-up. The average macroblock decoding time per core is below 2 us. Synchronization has to be very fast, even with the limited core count of 2-4, to gain improvements. It is expected that even with a properly parallelized solution the CMPs will not be as efficient

as the Cell RL implementation.

The x86 processors performance does scale well when the IPC improves. This can be clearly seen from moving from a Pentium4 to a Core2 Duo or PhenomII processor. The average macroblock time has been decreased by a factor 2.4. This is not surprising as the sequences are decoded sequentially. However, the improvements in IPC are slowing down and power limitations have put a halt to frequency scaling. It is expected that the practical limit will be reached at around 1 us per macroblock. Decoding a QHD sequence is still possible, but moving further will not be feasible.

In terms of absolute performance the projected performance of MFRL with 16 SPEs is impressive. Furthermore, RL is expected to both scale with IPC per core improvements as well as the number of cores, given that the memory subsystem capabilities scales accordingly. Judging from the x86 performance results, the Cell processor still has a lot of room for improvements in terms of IPC. The average macroblock execution time per core is a factor 7-8 times higher compared to the PhenomII.

On the other hand moving to higher resolution allows for higher number of processing elements, while maintaining the same efficiency. For MFRL it is expected that for Super HiVision resolutions, which has 4 times the width of FHD, 64 SPEs can be used while maintaining an efficiency ratio of 90%. In Chapter 8 the requirements for future applications of the RL algorithm is discussed further.

7.4 Conclusions

In this chapter the performance results of TP and RL have been presented and analyzed. The performance results of RL are impressive. With 16 SPEs, the obtained frames per second for macroblock processing is between 125 and 160 fps for FHD sequences. For TP the performance is lower at 68 to 82 fps depending on the used sequence. The performance difference is between 1.6x and 2x and originate from a better scalability and base performance, both in favor of RL.

The scalability of RL with 16 SPEs is observed to be around 12x. This coincides with the theoretical expectations. It has been observed that the Cell implementation of RL follows the theoretical scalability over the entire range of 1 to 16. On the other hand, TP only showed a scalability of 10x with 16 SPES. This is much lower than the theoretically expected 14.3x.

The difference in theoretical and actual scalability of TP is caused by the influence of platform-specific effects. These effects are the memory access contention and synchronized access contention. Both these effects have an approximately equal negative influencing on the scalability. They both contribute about 4-5 us in the average macroblock decoding time at 16 SPEs compared to 1 SPE, in the FHD BlueSky sequence.

The RL implementation is virtually unaffected by the platform-specific effects. This follows from three features of RL. First, the non-blocking distributed control has been mapped on the Cell to send synchronization messages to the neighboring SPE. This synchronization is non-blocking as it does not expect a response. Second, the concurrent communication and computation hides the memory latencies. Therefore, the memory access contention only starts to have a visible influence on the performance, when the available memory bandwidth has been saturated for more than 80%. After this point the

memory latencies can no longer be completely hidden behind the computation. Finally, using explicit SPE-SPE communication conserves a lot of memory bandwidth compared to TP. Therefore, the performance is able to scale further in terms of performance before becoming memory bandwidth limited.

In contrast, the TP implementation suffers from both the memory and synchronized access contention. The centralized worker-server scheme requires synchronized access to the shared structures. Increasing the number of SPEs in turn increases the contention at the PPE, which acts as the server. The synchronized access latency is still relatively low up to 8 SPEs t at 1 us. After this point, the latency increases rapidly to around 5 us. The additional latency of the off-chip SPEs causes the accelerated increase. The memory access contention is also affecting the scalability of TP. Since the memory latencies are not hidden behind the computation, the effects of the contention are directly visible in the execution time.

It can be concluded that the TP implementation is not very scalable in practical application. The scalability will improves if both the memory bandwidth and synchronization latency characteristics improve. However, from a performance and efficiency point of view it is better to opt for the RL solution, instead of trying to improve on the TP. Due to its algorithmic characteristics it is able to leverage both improvements in IPC per core and in the core quantity. In both cases TP would quickly be bottlenecked by the synchronization contention.

When comparing the Cell processor using the RL implementation to modern state-of-the-art x86 processors using a the original FFmpeg decoder, it can be concluded that the Cell processor is superior in both absolute performance and performance/transistor. Furthermore, it is shown that the SPEs have a lot of room for improvement in terms of IPC. The average macroblock execution times are 7-8 times higher compared to the Core2 and PhenomII processors.

In the next chapter several case studies of future applications of RL are discussed. The case studies concern possible future iterations of the H.264 standard.

8 Future applications

8.1 Quad HD and Super HiVision

As the name suggest Quad HD stands for a resolution of 3840x2160, which is four times FHD. Super HiVision goes another step further and represents a resolution of 7680x4320, which has 16 times the pixel count of FHD. With the unending urge for better visual experiences these resolutions are likely to be standardized in future H.264 iterations. More compute capabilities are required to decode these type of sequences. With diminishing IPC and frequency improvements, single-threaded performance will not fulfill this need. The scalable Ring-Line (RL) approach, however, can make use of future many-cores to deliver the necessary performance.

To investigate the hardware requirements for Quad HD and SuperHiVision, the obtained results in Chapter 7 are used as a base. More specifically, the performance of the projected MFRL with 16 SPEs is 168 fps for FHD. To be a bit conservative we use 150 fps as the base performance. The projected efficiency of the MFRL at this setting is 90%. The required frame rate of todays HD video sequence is between 24 to 30 fps. This is enough for smooth playback most of the time. However, if fast motion sequences a higher frame rate still brings a better experience. Therefore, in addition the frame rate requirements in our analysis is set to 60 fps.

Quad HD video has two times the horizontal width of FHD. In Section 4.2.3 it is derived that increasing the width of the video allows us to proportionally increase the number of SPEs, while maintaining equal efficiency. Therefore, 32 SPEs could be used with an efficiency of 90%. With 32 SPEs the performance would be 75 QHD fps. This is more than the required 60 fps and even 25 SPEs would be enough.

With Super HiVision the horizontal resolution doubles again and 64 SPEs can be used with an efficiency of 90%. The projected performance at this resolution is 37.5 fps, which does not meet the 60 fps requirement. While adding more SPEs is a possible solution, it decreases the efficiency. It might be more efficient, area and/or power wise, to increase the IPC of the SPEs instead. In Section 7.3 it was discovered that there is still a lot of headroom to improve in this area. Therefore, it is suggested to increase the IPC with a factor 60/37.5=1.6 compared to the current SPEs.

The entropy decoding should be performed on a high performance ILP centric core like the AMD PhenomII x4 cores. The single-threaded CABAC performance on this processor is measured to be around 160 FHD fps. The number of entropy decoding instances should be kept as low as possible to conserve memory. Each entropy decoding instance requires two work unit matrices to operate in a double buffered fashion. For QHD $\lceil 1.5 \rceil$ of these cores are needed, which results in 4x62.2=248.8 MB of external memory. For Super HiVision this increases to 6 cores and 12x249=3 GB. As you can see the memory requirements rise up quickly as supporting higher resolutions requires more

cores and more memory per core.

Using less powerful cores like the Cell PPE or SPE for the entropy decoding, requires a factor 6-8 more memory. Clearly this is not desired. A standard revision in regards of the entropy decoding might solve the problem. By using slices only on the entropy decoding, inversely proportional memory is required. Since only the entropy decoding is sliced the macroblock decoding part stays the same. The compression rate suffers a little due to the extra slice headers, however this is negligible for low number of slices.

8.2 Stereoscopic 3D and free viewpoint video

Stereoscopic 3D video has been a hot topic for several years. In the envisioned application of stereoscopic 3D, the screen images are quickly alternated for each eye in a synchronized pace with the blackening of the shutter glasses. Our eyes receive frames from two different viewpoints of the video, thus creating a 3D effect. Free viewpoint video allows the user to freely navigate in real world visual scenes, as known from virtual worlds in computer graphics.

Up to now there has not been a real breakthrough to the consumer market, however, at the moment there is a lot of movement in the 3D-TV direction. It is expected that a real breakthrough is imminent in 2010. With current high refresh rate LCD television, ranging up to 240 Hz, 3D functionality can simply be added by upgrading the external player [28]. No new televisions are necessary, which lowers the threshold for adoption.

Several proposal have been applied for extending the H.264 standard to enable multiple view paths [25][8]. The proposals have in common that they suggest inter-path frame dependencies to increase compression rate, while maintaining the picture quality. In other words, no frame parallelism can be exploited. The desired solution is to process the frames sequentially at a much higher frame rate. The RL approach has great potential in this regard.

For our case study we focus on stereoscopic 3D video. We assume 60 frames per second per view path totaling in 120 frames per second to satisfy both eyes. Extending to free point video simply translates to adding more view paths. The considered resolutions are FHD, QHD, and Super HiVision, discussed in the previous section.

To decode 3D FHD at 120 fps even the current single-frame RL implementation on 16 SPEs suffices for the macroblock processing. In Section 7.1 it is shown that the performance is between 125 to 160 fps. As explained in the previous section, using fast ILP centric cores for the entropy decoding is important in terms of memory usage requirements. The CABAC performance of the PhenomII core is measured to be 160 fps. Since only one instance of the CABAC decoder is needed, two work unit buffers are used, with a size of 2x15.5=31 MB.

Going further to QHD requires more compute capabilities. Since QHD is essentially four times FHD, the computational requirements also quadruple. For the entropy decoding 3 PhenomII cores are needed. The size of the work unit buffers is in this case 6x62.2=248.8 MB. For the macroblock processing more cores are required. In QHD the horizontal resolution is doubled which enable us to use 32 SPEs, while maintaining 90% efficiency. The projected performance of 32 SPEs is around 75 QHD fps. Therefore, using 32 SPEs does not deliver sufficient performance. To maintain the scaling efficiency

it is suggested to increase the IPC with a factor 120/75=1.6 to meet the performance requirements.

Finally, moving to stereoscopic 3D in Super HiVision resolutions requires immense compute capabilities. First, for the CABAC decoding another factor 4 in entropy decoding cores is required, totaling in 12 cores. The memory requirements are in this case 24x249 MB=6 GB. For the macroblock decoding the number of SPEs can be doubled again to 64, which results in a performance of 37.5 Super HiVision fps. As the performance is now clearly not up to the required level adding more SPEs cannot solve the problem on its own. A factor 120/37.5=3.2 in performance is lacking. This can be solved by increasing the IPC of the cores with a factor 3.2 compared to the current SPEs. To put this into perspective this is at the same level as the Pentium 4 Prescott core, which still has a factor 2-2.5 lower IPC compared to the PhenomII core.

Table 8.1 summarizes the requirements. The memory bandwidth requirements should be interpreted as a factorial performance improvement over the current Cell memory subsystem.

	CABAC cores	Memory size	SPEs	SPE IPC	Memory bandwidth
Full HD	0.75	31 MB	16	1x	1x
Quad HD	3	248.8 MB	32	1.6x	4x
Super HiVision	12	6 GB	64	3.2	16x

Table 8.1: Hardware requirements for stereoscopic 3D-TV at 120 fps using the MFRL decoding strategy. CABAC cores are assumed PhenomII cores at 3.2 GHz, capable of processing 160fps at FHD

For free view point video at the QHD and Super HiVison resolutions a standard revision is suggested. Both the memory requirements and the inter-path frame dependency will hinder further parallelization.

8.3 Embedded and Accelerator Integration

Devices like set-top boxes, bluray players, mobile phones, and televisions integrate H.264 decoding capabilities. As a trade-off to between power consumption, time-to-market and production cost a embedded solution is used. The embedded SoCs consist of a semi-custom processor to do the task at hand. Often these processors are designed to meet a certain performance specification. The RL algorithm could be used as the base for these chips to perform the H.264 decoding as it has proven to be both efficient and scalable.

A possible design for a embedded chip would be to use a further specialized SPE core for the macroblock processing. Work has been done in this direction by Meenderinck [19]. A speed up of 2x was achieved by adding 12 instructions to the SPE ISA. For the entropy decoding one might opt for one or more CABAC ASIC to conserve memory requirements. A small control processor is needed to perform the synchronization.

The RL approach could also motivate the integration of SPE like cores as accelerators. With transistor budget doubling every 18-24 months, filling the chip with useful components becomes increasingly difficult. Adding more cores does not always help as programs can only be parallelized to a certain extend. Also due to power restriction we might end up with a processor that can only have part of the executions cores active.

9 Conclusion

In this thesis we have analyzed, implemented, and compared two parallel H.264 decoding strategies on the Cell architecture. The implementation and comparison of the two strategies has great value for gaining insight on programming the Cell architecture and future many-cores in general. Both the Task Pool (TP) and the novel Ring-Line (RL) approach exploit macroblock-level parallelism. Investigating the parallelization of H.264 is also important for solving the compute requirements of future iterations and applications of the H.264 standard.

The first step in our investigation was to provide a theoretical scalability analysis of the TP and novel Ring-Line approach. The TP algorithm builds on a worker-server model, in which the macroblocks act as the work units. By using the dependency table and task queue structures, the state of the macroblocks is updated according to the macroblock dependencies. When the dependencies of a macroblock resolves it is pushed in to the task queue, where it waits for an available worker. The theoretical speedup found through simulation a FHD sequence is found to be 14.8x with 16 processing elements. The platform-specific effect like synchronization overhead are not considered in the simulation.

In the novel RL approach each processing element processed an entire scan line of macroblocks. When this is finished, it continues with the next available line until the entire frame is decoded. No shared structures are required as it is build on a distributed control mechanism. The processing elements are mapped in a ring network and only require to synchronize with their neighbor(s). The RL simulation results show a speedup of 12x with FHD and 16 processing elements. The dynamic macroblock scheduling of TP is less affected by the dependency stalls, resulting from variable execution times. In exchange the RL algorithm is predictable, as it can be exactly predicted which processing element processed which macroblocks. This allows us, among others, to hide the communication latencies behind the computation.

For both TP and RL the DMA alignment restriction causes parallelism loss. For TP, the overlapping writes to the frame would cause faulty images. The solution is to increase the minimal horizontal spacing of the concurrent macroblocks by one. This caused the parallelism to drop from 60 to 40 with FHD sequences. The effect on the scalability remains small with 16 SPEs as the speedup dropped from 14.8x to 14.3x. In RL this is solved by combining and delaying the write back step of two adjacent macroblocks, which also conserves memory bandwidth. The speedup drop of 12x to 11.7x was marginal at 16 SPEs.

In RL, by keeping the intra data on-chip and the more efficient write back, the memory bandwidth requirements are between 1.75x to 3x lower. It is investigated that for TP using the FHD BlueSky sequence a maximum of 84.7 fps is attainable, before saturating the memory bandwidth. For RL this is 149.2 fps. In less motion compensation

demanding sequences as RiverBed the difference is even higher with 132.8 vs 357 fps.

The experimental results showed that RL is between 1.6x and 2x times faster than TP. With the FHD Bluesky sequence using 16 SPEs a performance of 68.8 and 136.1 fps is attained for TP and RL respectively. The performance level of RL is impressive as it is close to the memory bandwidth limits. Even more impressive is that the scalability follows the theoretical expectancy almost perfectly. In contrast, the performance and especially the scalability of TP is poor. Most of the performance difference originate from the concurrent communication and computation of RL. However, a theoretical speedup of 14.3x is expected, but only 10.3x is observed. The practical scalability has dropped below the (theoretically) less scalable RL. This is caused by platform-specific effects, namely the memory access contention and synchronized access contention.

While TP exhibits excellent scalability in theory, it can be concluded that TP is not a very scalable solution in practice. The scalability deficiencies will be even more visible by using more and/or faster cores. In contrast, the practical results of RL show that it is virtually unaffected by platform effects. This property indicates a truly scalable implementation. It is expected that RL scales effortless to higher number of cores, when using larger resolutions.

Since RL is only applicable to the macroblock decoding part of H.264, the parallelization of the entropy decoding remains an open topic for future work. It has been shown that parallelizing the entropy decoding on the frame level requires large amount of memory, increasing quadratically with the resolution. A related future work is implementing the multi-frame RL. In this thesis we have restricted ourselves to single-frame RL, due to implementation difficulties. The projected performance gain of multi-frame RL is 20% for FHD sequences using 16 SPEs, increasing the efficiency to 90%.

From the comparison of TP and RL a more general conclusion of programming the Cell and local store architectures can be drawn. Extracting maximum performance out off the Cell requires a level of predictability in the algorithm to exploit concurrent communication and computation. Furthermore, predictability also allows for a distributed control mechanism. When exploiting fine-grained parallelism the latter is a requirement to scale to many-core processors. To extract performance out of future heterogeneous many-cores parallelism with a distributed data flow is required. Future parallel programming languages and models should stimulate the programmer to specify the parallelism in a data flow manner.(We are currently working on a parallel programming model based on function block-level parallelism. Implementing this requires large innovation in the programming language, compilers, runtime system, operating system, and the architecture.)

Bibliography

[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.H. Meenderinck, B.H.H. Juurlink, and M. Valero, *Scalability of macroblock-level parallelism for h.264 decoding*, Proc. Int. Conf. on Parallel and Distributed Systems, 2009.

[3] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, *A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC*, Proc. IEEE Int. Workload Characterization Symposium, 2005, pp. 24–33.

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

[5] A. Azevedo, C.H. Meenderinck, B.H.H. Juurlink, M. Alvarez, and A. Ramirez, *Analysis of Video Filtering on the Cell Processor*, Proceedings of International Symposium on Circuits and Systems (ISCAS), May 2008.

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

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

[8] C. Bilen, A. Aksay, and G.B. Akar, *A multi-view video codec based on H.264*, Proceedings of the IEEE International Conference on Image Processing (ICIP?6), 2006, pp. 541–544.

[9] CBEA JSRE Series, *Spe runtime management library 2.3*, 2008.

[10] T. Chen, R. Raghavan, JN Dale, and E. Iwata, *Cell Broadband Engine Architecture and its First Implementation: a Performance View*, IBM Journal of Research and Development **51** (2007), no. 5.

[11] Y.K. Chen, X. Tian, S. Ge, and M. Girkar, *Towards efficient multi-level threading of h.264 encoder on intel hyper-threading architectures*, Proc. Int. Parallel and Distributed Processing Symposium, vol. 18, 2004.

[12] *The FFmpeg Libavcodec*, `http://ffmpeg.org`.

[13] M. Flierl and B. Girod, *Generalized B pictures and the draft H.264/AVC video-compression standard*, IEEE Transactions on Circuits and Systems for Video Technology **13** (July 2003), no. 7, 587–597.

[14] M. Greenwald and D. Cheriton, *The synergy between non-blocking synchronization and operating system structure*, ACM SIGOPS Operating Systems Review **30** (1996), no. si, 123–136.

[15] A. Gulati and G. Campbell, *Efficient mapping of the H.264 encoding algorithm onto multiprocessor DSPs*, Proc. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 5683, 2005.

[16] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez, *Performance analysis of cell broadband engine for high memory bandwidth applications*, May 2007, pp. 210–219.

[17] M. Kistler, M. Perrone, and F. Petrini, *Cell multiprocessor communication network: Built for speed*, IEEE micro **26** (2006), no. 3, 10.

[18] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez, *Parallel Scalability of Video Decoders*, Journal of Signal Processing Systems **57** (2009), no. 2.

[19] C. Meenderinck and B. Juurlink, *Specialization of the Cell SPE for Media Applications*, Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors-Volume 00, IEEE Computer Society, 2009, pp. 46–52.

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

[21] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al., *The Design and Implementation of a First-Generation CELL Processor*, Proc. IEEE Int. Solid-State Circuits Conference (ISSCC), 2005.

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

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

[24] H. Shojania, S. Sudharsanan, and Chan Wai-Yip, *Performance improvement of the h.264/avc deblocking filter using simd instructions*, Proc. IEEE Int. Symp. on Circuits and Systems ISCAS, May 2006.

[25] A. Smolic, H. Kimata, and A. Vetro, *Development of MPEG standards for 3D and free viewpoint video*, Three-Dimensional TV, Video, and Display IV **6016**.

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

[27] 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** (2003), no. 7, 560–576.

[28] A. Woods, T. Rourke, and K. Yuen, *The Compatibility of Consumer Displays with Time-Sequential Stereoscopic 3D Visualisation*, Proceedings of the K-IDS Three-Dimensional Display Workshop, vol. 2006, Citeseer, 2006, p. 21.

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

[30] 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).

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