Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's Thesis

Realtime Optical Flow Algorithms on the Cell Processor

Pascal Gwosdek

2008-4-14

Supervisor	Prof. Dr. Joachim Weickert
Advisor	Dr. Andrés Bruhn
Reviewers	Prof. Dr. Joachim Weickert
	Prof. Dr. Reinhard Wilhelm

Affidavit

I hereby declare that this master's thesis has been written only by the undersigned and without any assistance from third parties.
Furthermore I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Saarbrücken, 2008-4-14

Declaration of Consent

Herewith I agree that this thesis will be made available through the library of the Computer Science Department. This consent explicitly includes both the printed, as well as the electronic form attached to the last page of this thesis.
I confirm that the electronic and the printed version are of identical content.

Saarbrücken, 2008-4-14

Abstract

The estimation of motion information in terms of optical flow is nowadays a common instrument for many applications in Computer Vision. Relying on the solution of a linear or nonlinear equation system, those algorithms are still comparably slow, even though realtime setups can already been established for smaller images. Modern processor layouts show promise to accelerate this procedure, or to be capable of larger problem sizes.

In this thesis, the potential of the Cell Broadband Engine built into the Sony Playstation 3 video console is explored and its applicability for the optic flow problem evaluated. After creating a setup for scientific computing on the device, different parallelization attempts distributing computational load to seven available cores are compared with each other. Hereby, specific characteristics of the Cell processor layout are addressed and the impact of specific acceleration techniques optimized for the architecture is appraised. Efficient solvers known from both sequential and other parallel architectures, namely Block SOR, Red-Black SOR and Full Multigrid, are tested for their performance on the novel platform, whereby particular attention is paid to memory management and synchronization among the different cores. It is shown that by applying hardware-specific optimizations, optical flow algorithms can be significantly accelerated compared to a traditional processor. Eventually, an interactive realtime configuration with graphical output is described and tested, and the capabilities of the Playstation 3 are compared and discussed with respect to this setting.

Acknowledgements

I would like to express my sincere thanks to my advisor Dr. Andrés Bruhn and my supervisor Prof. Dr. Joachim Weickert for their excellent mentoring and support during the development of this thesis. Their encouragement and the friendly atmosphere among colleagues in the Mathematical Image Analysis Group greatly assisted me in my work. I also like to thank Prof. Dr. Reinhard Wilhelm for consenting to review my thesis.

Special thanks also go to my parents and my sister who enabled me to perform my studies as I did and who therefore also significantly contributed to the success I have enjoyed so far.

Contents

Chapter 1

Introduction

1.1 Motivation

In Computer Vision, one of the main challenges is the automatic extraction of motion information from image sequences. Without any prior knowledge about the captured scene, one often not only wants to purely identify movements in a scene, but also likes to know in which direction an object is travelling, and at which velocity.

Since the position and own motion of the camera is usually not known as well, one often confines oneself to the relative movements of objects with respect to the camera, and describes these displacements between two subsequent frames of the sequence with a dense vector field, the *optical flow* field.

Nowadays, optical flow fields find use in a broad range of applications, like for video compression or enhancement, object tracking, driver assistance systems, autonomous navigation of robots, or video surveillance [16]. One example for such application is depicted in Figure 1.1, where the movement of several objects in a traffic sequence is appraised.

Several algorithms to estimate the optical flow have been proposed in the past. Many of them belong to the so-called *variational approaches*, which formulate the process as a minimization process of an energy functional [27]. Though these methods are among the most

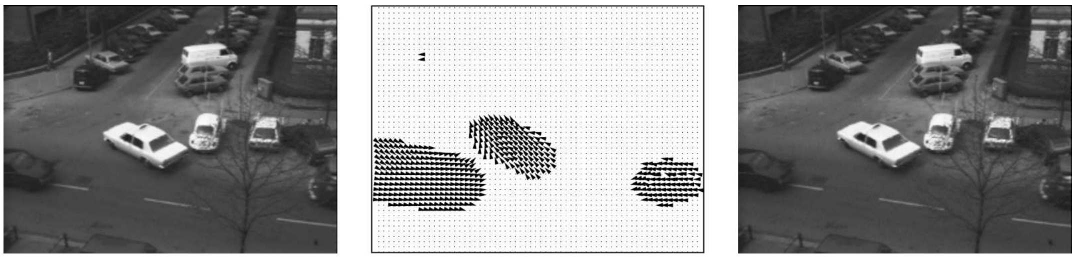

Figure 1.1: Left and **right**: Frames 10 and 11 of the well-known Hamburg Taxi sequence by Nagel. **Middle**: Estimated optical flow field, visualized with arrows pointing in direction of the motion. **Authors:** J. Weickert and C. Schnörr [67].

accurate methods available, they are also quite expensive to compute, because they require large systems of equations with millions of unknowns to be solved.

By numerical improvements, these algorithms could already be accelerated by two orders of magnitude, and for smaller image sizes, even realtime performance have been achieved on standard PCs [14]. For various time-critical applications however, these results are still not sufficient.

Modern parallel processor architectures promise a remedy to this problem, since they offer a higher performance by distributing the algorithm to several cores. Particularly, the Cell Broadband Engine built into the Sony Playstation 3 video console, one of the cheapest parallel computers on the market, has already been used for many scientific purposes: Current application fields include for instance matrix multiplications [18], the implementation of diffusion processes and the Fast Fourier Transform [71], an entire raytracer implementation [7], or partial differential equation based video compression [44].

1.2 Goals

This thesis is meant to evaluate the applicability of the Cell Broadband Engine for the optic flow problem, and should in particular test whether it is eligible to accelerate common algorithms significantly.

For this purpose, a Sony Playstation 3 video console shall be prepared for scientific use, and a development platform allowing to create, debug and execute new programs should be set up. By a closer study of the Cell Broadband Engine's characteristics, a general impression of the underlying architecture should be gained and ways and means shall be considered to use these peculiarities for advanced optimizations.

Within this framework, several efficient algorithms with prospect of realtime performance shall be suitably parallelized for the application on the Cell processor. By optimizing them for this architecture, their runtime performance is to be be enhanced best possible, compared to a sequential implementation on an Intel Pentium 4 processor. Particular attention should thereby additionally be paid to the performance with respect to different image sizes, especially towards larger frame dimensions.

Since the experiments are going to be performed on a Playstation 3, this special platform shall as well be attentively regarded, and related advantages or disadvantages should be carved out accordingly.

1.3 Contents

The structure of this thesis is as follows. In Chapter 2, an introduction to the Playstation 3 video console is going to be given, and its hardware specifications are described. Proximately, the Cell Broadband Engine Processor is in detail presented in Chapter 3, whereby special attention is paid to the cores used for high performant computations. In particular, their specific characteristics, as well as their technical means of interaction and synchronization are going to be explored. Chapter 4 is finally going to conclude this rather hardware-oriented episode with information about how to enable the Playstation 3 for scientific computing. This covers both the installation of a multi-purpose operating system and compiler toolchain, as well as a discussion about the opportunities this setting offers with respect to development and debugging of new programs.

Subsequently, a short introduction into the optical flow problem is given in Chapter 5, and the CLG optical flow model by Bruhn *et al.*, which is going to be used throughout this thesis, is being explained [17].

Chapter 6 introduces the so-called *Successive Over-Relaxation* method as a simple solver for equation systems arising in context of the optical flow computation. Beginning with a theoretical introduction and an appraisal of the performance of existing implementations, two parallel implementations are going to be presented aiming at an acceleration of the computation process. In a last experiment, the second algorithm is implemented in an alternative concrete realization to reduce constant expenses. The effectiveness of all parallelization attempts is thereby immediately evaluated in terms of performance benchmarks.

In Chapter 7, a Multigrid solver is used, which is known for a high performance and fast convergence on sequential architectures. Like for the previous method, the theory behind this idea is briefly formulated, and reference benchmarks are issued on a sequential implementation. For the optimized version, a new parallelization concept is being presented and a framework is introduced that allows both high performance and simplicity in development and maintenance. Again, this newly devised version is being evaluated with respect to runtime performance, and is compared to the results achieved on traditional hardware.

Eventually, the developed algorithms are tested in an interactive realtime setting, including webcam input, processing of the yielded frames and output of the results in an appealing manner. Chapter 8 hence evaluates the general qualification of the Playstation 3 for such purpose, and shows advantages and disadvantages of this configuration.

Finally, the prior units are concluded in Chapter 9, and a recapitulatory review about the applicability of both the Playstation 3 and in particular the Cell Broadband Engine is given, before the thesis is going to be completed with proposals for advanced research topics based on the presented results.

1.4 Typographic Conventions

Throughout this thesis, there are references to several programs involved in the process of creating programs for the Cell architecture, of analysing the program flow, or of improving the performance. In order to highlight these terms and to visually separate them from the surrounding paragraphs, they are set in typewriter font.

Hereby, bold face denotes parts which can literally be typed into the console such as **`programs`** and **`--options`**. Meta symbols like placeholders are set in normal weight and in angle brackets, as it is for `<foobar>`. The system prompt is denoted by a single **`$`** sign, while internal prompts might differ from program to program. For both types, though, requests for interaction are symbolized by a rectangular cursor: ▮.

Chapter 2

Sony Playstation 3

2.1 History

On December 3, 1994, Sony Computer Entertainment Inc. released their Playstation video console to the Japanese market [68]. Being a newcomer on the video gaming market, Sony succeeded to displace Nintendo Corp. from their almost certain first place they still claimed with their Super Nintendo Entertainment System, and should dominate the market for quite some years. From today's perspective, two main factors can be made responsible for this change: On the one hand, Nintendo did not spend much effort to press ahead with the development of their own console, the N64, since they did not see their own position to be endangered at all. On the other, Sony very early disclosed many internal design documents towards third party game developers, such that the Playstation could quickly resort to a large assortment of available video games.

From this time on, a strong competition between Sony, Nintendo, and Sega started to flare up, and was often dubbed the *console war* by the media. To the end of 1998, Sony released its sixth generation video console, the Playstation 2, to the market [70]. Even though Microsoft started to challenge the established companies with its Xbox being published to the same time, and Nintendo contributed its Game Cube into the race, the Playstation 2 could still claim the first place in the sales statistics, and even became the most often sold video console ever. Besides the substantial hardware configuration and a large fan community, the downwards compatibility towards original Playstation games can be spot as one main reason for this success.

With the next generation of video consoles being in prospect, the managers of Sony Computer Entertainment were indeed aware of the fact that there is a lot at stake, and the success or failure of the next product to be published might bring a final decision in the competition. Nintendo did still not miss the connection and was in a good position to claim its former position as a market leader back, and Microsoft was in addition expected to improve its first attempts on this market [69].

During the past generations of video consoles two success factors turned out to be most effective ones, namely a great variety of games, and computational power. While for the first

Figure 2.1: The Sony Playstation 3 video console. **Source**: `http://www.golem.de`.

point, the existing number of old games and in particular a good collaboration with third party games developers could only be of benefit, the second criterion was a lot harder to achieve, especially since very little was known about the efforts the competitors will invest.

Due to these trade secrets, the dimensioning for the new hardware for the Playstation 3, officially marketed as PLAYSTATION® 3, was kept rather optimistic, hoping to be finally ahead of the others. The CEOs of Sony Computer Entertainment Inc. hence encouraged performances of the new processor to hit the magnitude of a factor of 1000 with respect to the current Playstation 2 chip [40].

However, this idea was soon discarded due to technical and financial limits in the construction of hardware, and in particular due to problems with the power consumption and required heat dissipation. Instead, a factor of 100 has been said to be achievable, and the research division of IBM was commissioned to aid with general support from a technical viewpoint, but also to be finally involved in the actual design of the chip. In parallel, experts from Nvidia Corp. have been called in to contribute to the layout of the graphics chipset.

2.2 Hardware

Though the main processor differs significantly from those built into standard PC architectures, the general design of the Playstation 3 can be considered closer to such layout than to former video consoles. External interfaces have explicitly been held compliant to common standards, like the four USB 2.0 front ports the controllers can be connected to for initial registration, or for charging. These ports can furthermore be used to connect USB mice or keyboards, which are officially supported by the firmware. In the game menu, they then replace functions elsewise assigned to the controller, and ease text input in the built-in browser window or other input forms. Components like the 2.5” 60 GB harddisk with SATA connectors are widely available in PC stores as well [50]. For online gaming and web based distribution of older or smaller video games, the Playstation 3 has a Gigabit LAN connector, by which it can be connected to the internet, as well as a wireless LAN chipset and a bluetooth transceiver.

The basis of the system is represented by the Cell Broadband Engine processor, the entirely novel design developed by IBM and Sony. Since it plays the central role for this thesis, it is explained in more detail in course of Chapter 3. The main memory is with 256 MB rather small, in particular in context of the rather high data throughput the Cell processor can achieve. However, since it is connected to the Cell processor using eight data channels with bus clocks of 400 MHz, a maximal bandwidth of 25.6 GB/s is achieved. Memory operations can thus be performed quite cheaply, which is especially inevitable in the context of up to eight cores potentially issuing requests at the same time.

For graphics processing, Nvidia likewise developed a customized redesign for the special application in the Playstation 3. Despite of being based on the G-71 chipset, little is known about the actual specifications of this so-called *RSX Reality Synthesizer*. However, it is claimed to be clocked with 550 MHz, to work upon 256 MB of dedicated graphics memory, and to be capable of 136 shader operations per clock cycle [50]. This should it settle in about the region of the GeForce 7800 when it comes to performance, while its memory bandwidth is rather comparable to the GeForce 7600 GT. Since Sony Computer Entertainment, Inc. is a promoting member in the Khronos group speaking up for open standards, they most probably also use OpenGL ES, the embedded system variant of OpenGL, to drive this chip [41]. To connect external monitors, both composite and HDMI output is available, and the video standards 480 i/p, 576 i/p, 720 i/p and 1080 i/p are supported [54]. For the 1080 i and p formats, the maximal resolution of 1920×1080 pixels is achieved.

A novelty in the video console sector is the included double speed slot-in Blue-ray drive, which is also capable to play $8\times$ DVD and $24\times$ CD. It represents the main input device for video games, and is extended by the internal hard disk, which can be filled with games from online sources, as well as by a multi card reader. Furthermore however, the Blue-ray drive can also play videos distributed on any of the supported media types, and a software media player included in the firmware supports several container formats and video compression standards [50]. This makes the Playstation 3 also a cheap alternative to dedicated Blue-ray player devices, since the video console is subsidized by game sales.

The peak power consumption is reported to amount to 192 W, and even when it is idle, an average power of 160 W is consumed [50]. This is a lot compared to other consoles like the Nintendo Wii, and is in the range of high-end multimedia desktop PCs.

2.3 Scientific Interest

Despite of its advanced technical configuration, the Playstation 3 failed so far to maintain its predecessor's position on the market. Being promoted for a long time, the development of the Cell processor and its embedding into the surrounding system design retarded the actual release on the Japanese market to November 11, 2006 [69]. Surprisingly, the Nintendo Wii being released at the same time could recapture the market by improved gameplay and human interaction interfaces, rather than by pure performance.

Nevertheless, the Playstation 3 is still very widely used as a multimedia center, but also for scientific computing: The Cell processor found its way onto devices exterior to the Playstation 3 since, because IBM also markets it built onto so-called *blades*, a compact construction form for highly performant computation hosts. Being significantly more expensive than the subsidized Playstation 3 however, latter is broadly being used for experiments with the Cell architecture, in particular in non-profit applications like for university research projects.

In this context, several works have already been published, like implementations of the Fast Fourier Transform [71], clustering of several Playstation 3 consoles for experiments with dense and sparse linear algebra [18], a ray tracer implementation [7], or partial differential equation based video compression [44]. In context of this thesis, the suitability with respect to the Optical Flow problem is inspected in more detail.

Chapter 3

Cell Broadband Engine

3.1 History

During the first considerations about the new processor, the research division of IBM soon began to stake out the possibilities and limitations with respect to both the requested performance, but also with respect to the production cost, since a too high prices potentially endanger the success of a device designed for the the adolescent clientele. While it was always clear to the designers that single core solution will hardly fulfill the performance specifications, the actual layout of a potential multi core setting has been substantially discussed about.

Until then, common concepts have already included symmetric multi core settings, this means such with equally designed cores, but to meet performance, production cost, power consumption, and heat dissipation requirements, it was soon clear that traditional architectures do not suffice for this purpose. Many modern video games need noticeably accelerated computations rather than advanced multi-threading concepts traditional settings provide. Correct game physics, particle clouds, or the simulation of fluids or nature are only some examples for such resource-intense applications nowadays widely spread in many games. On the other hand, they are too variable in their actual hardware requirements, such that a pure special-purpose layout did not come into consideration as well. In addition, Sony not only sees its new generation video console to be a pure gaming station, but should also be useable as a media center (cf. Section 2.2), and the processor should hence be capable of realtime media processing as well.

All these considerations finally lead to a fundamentally new design incorporating both traditional components as well as hardware optimized for special purpose computations. The actual development of this new chip has been taken charge of by a consortium consisting of Sony Computer Entertainment Inc. (SCEI) as the content provider and principal, International Business Machines Corp. (IBM) being responsible for the chip design, and finally Toshiba Corp. as a development and high-volume manufacturing technology partner [40].

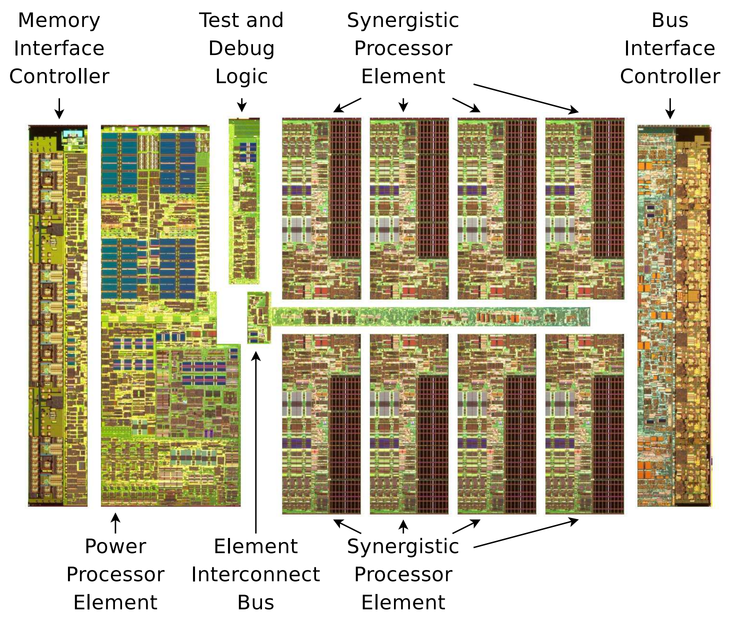

Figure 3.1: The Cell Broadband Engine Processor in an exploded view, structured by its most important features. **Source**: Based on a picture from `http://mnagano.wordpress.com/`.

3.2 Layout Overview

The Cell Broadband Engine is an asymmetric multi core processor with nine cores, whereof eight can be regarded as special purpose hardware. It builds up on a 64 bit hyper-threading enabled PowerPC core, called *PPE*, and is clocked at 3.2 GHz, being designed to run the operating system (cf. Figure 3.1). In section 3.4, it will be further discussed. This PPE is extended by eight 'synergistic' cores optimized for number crunching, but each only capable of one dedicated thread, a so-called *SPU kernel*, to be run. They hence strike a balance between traditional sequential architectures and special purpose hardware, while keeping the range of applications as wide as possible. Section 3.5 will explore these *SPE*s in detail [34].

All cores are extended by memory flow controllers and connected to a ring bus, the so-called *element interconnect bus* (EIB) (cf. section 3.6), which also attaches to the RAM and IO controllers (cf. sections 3.7 and 3.8).

In course of this thesis, a Linux based operating system has been installed on the Playstation 3 to use it for scientific computing (cf. Chapter 4). Unfortunately, out of the eight SPEs on the Cell used in the Playstation 3, not more than six are accessible under Linux. One of the two spare SPEs is blocked due to economic reasons, because Sony likes to keep the option open to build defective chips into the Playstation 3 consoles [50]. Such faulty chips are quite common when they are produced on wafers, since during photolithography, structures under greater incidence angles are usually subject to aberrations. Thus, chip manufacturing becomes more imprecise the further the chip is located away from the center of the wafer, and typical

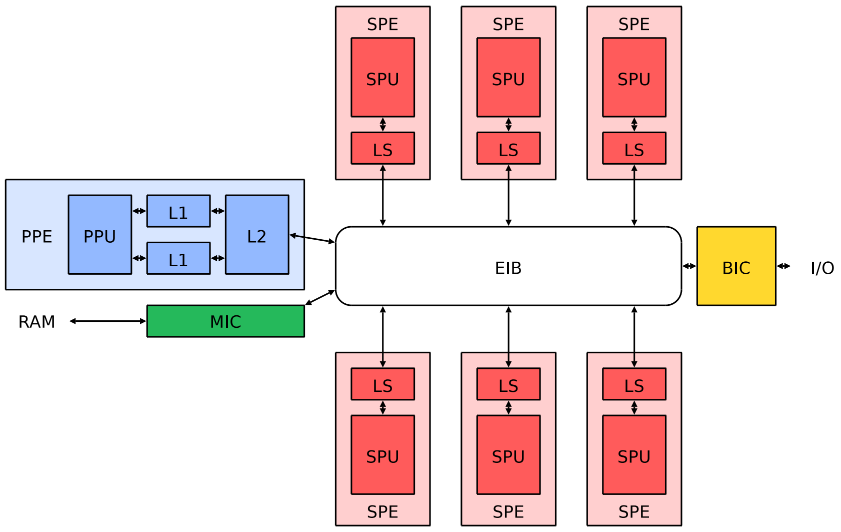

Figure 3.2: Block diagram of the Cell Broadband Engine Processor as it is visible on the Sony Playstation 3 running Linux.

consequences are for instance that faulty chips cannot be run on the intended frequency. If such error affects only one SPE, which is quite probable comparing the aerial fraction of the whole Cell processor, this SPE can just be turned off and though the chip can no longer be sold for IBM's blades, it is still applicable on the Playstation 3. As a side effect, both power consumption and heat generation can be reduced by a noticeable factor.

The second deactivated SPU is actually available in in the shipped firmware, *Game OS*, and can thus be assumed to work, but is nevertheless blocked in Linux. It is reported that this comes from it being used by the internal hypervisor, of which software components are permanently running in the background [25]. Details about the hypervisor and its functionality are described in chapter 4. Figure 3.2 shows a block diagram of the Cell Broadband Engine, restricted to the visible parts on the Sony Playstation 3 under a third party operating system like Linux.

It should again be emphasized that these restrictions are however only present on the Playstation 3. On products released for the high performance market, like IBM's Cell blades, all eight SPUs are available for computation, and development tools officially provided by IBM assume this situation as well.

3.3 Pipelining

All cores of the Cell processor follow the so-called *pipelining* concept, an effective constructional concept of basically all modern processor architectures. However, since particularities in the way this concept is implemented immediately affects the style and performance of programs throughout this thesis, a short excursion to this topic is going to be presented at this point, before detailing on the various components of the Cell Broadband Engine.

In modern processors, the execution of commands is arranged in several distinguishable stages. Each of these stages is processed within exactly one clock cycle, works upon data residing in inter-stage registers and writes its own results to the input registers for the next level where they are then waiting for being triggered by the next cycle. Even though this implementation consumes many clock cycles for one single instruction, there is usually no way around it because of data dependencies related to the memory or the general purpose register set.

However, many unbranched programs offer an opportunity to reduce the amortized amount of time one instruction needs by a significant factor, exploiting missing dependencies between subsequent instructions and the property of the single stages to be run in parallel. While one instruction is in a later stage, another can already be started in the first one without interference. Even though the delay of one single instruction cannot be reduced this way, the running time of a sufficiently long program does in fact converge to almost one clock cycle per instruction. This principle is called *pipelining*.

The execution time of any instruction can hence be divided in two parts, namely the residual time in one stage, which also equals the average time any instruction takes to be executed under the assumption that the pipeline is full, and the *latency* which is made up by the sum of remaining residual times. From a software developer's view, this means that any instruction at first glance needs its plain execution time plus latency to pass through the pipeline. By clever scheduling however, the latency of one instruction can already be used by other independent instructions to get started, and can in the ideal case be entirely compensated thereby.

Unfortunately, this simplified view of the program flow made so far can hardly be established in real programs, which are seldomly linear and built from independent instructions. Though, it is always possible to stall the pipeline to wait for any dependency to be fulfilled, which therefore builds a backup solution to the non-parallel execution whenever it is required. Several problems can even be resolved without stalling the pipeline and are hence a common approach in most modern processors.

A so-called *forwarding logic* allows some intermediate results of any stage to be fed back into earlier stages of the pipeline and hence to be used for succeeding instructions even before they are actually written back into the general purpose registers. Because instructions would normally receive their inputs from the preceding unit, the forwarding logic is in particular meant to catch all resulting deviations from usual program flow and to handle all special cases accordingly, such that the resulting execution process remains consistent to the non-forwarded version of the run. Looking closer to the pairs of stages potentially providing and requesting intermediate results, the total number of those combinations often grows faster than polynomial to the number of stages, which renders the forwarding circuit to become a non-negligible part of the processor layout. However, it is nevertheless extremely worthwhile, since it is in terms of the Cell processor able to reduce the latency of instructions from 22 clock cycles down to two to seven cycles, and thus offers the compiler a greater chance to find independent instructions to be scheduled in between for high performance.

On the other hand, conditional jumps within the program cannot be handled as effective in a pipelined setting as it would be in a sequential one. As soon as a so-called *branch* is actually evaluated, its describing instruction is already residing in a relatively low stage of the pipeline. Following the considerations above, succeeding instructions were already to be inserted into the pipeline. This is a problem however, as choosing any of the two possible branches can reveal to be incorrect when the branch is finally evaluated, thus forcing the processor to reject all instructions already started from the wrong branch and to start again with the correct one. Such scenario is referred to as *branch misprediction* and a still a frequent performance problem on modern processors. Paragraph 3.5.4 goes into more detail on this issue.

3.4 Power Processor Element

The *Power Processor Element* (PPE) is the main core included in the Cell Broadband Engine chip. Designed for running the operating system, it is fully compliant to 64 bit PowerPC processors like they are built into earlier Macintosh computers manufactured by Apple, Inc. It consists of a POWER Processing Unit (PPU), and a traditional cache hierarchy consisting of two 32 kB L1 instruction and data caches, as well as a joint 512 kB L2 cache [19]. The L2 cache connects with 25.6 GB/s bandwidth up- and downstream to the Element Interconnect Bus (cf. Section 3.6), and with 51.2 GB/s in both directions to the two L1 caches. Latter are eventually connected to the PPU using a bus capable of a bandwidth of 51.2 GB/s.

Because PPU and PPE are hard to distinguish from software side, since the cache hierarchy is hidden to the programmer, both terms are often referred to as synonyms. In fact, even the official documentation published by IBM and Sony is inconsistent in this point, in particular when it comes to rather software-related issues [29].

3.4.1 Architecture Overview

The PPU provides two pipelines for interleaved multithreaded computation of synchronous threads, but processes both locally in-order. This means that it does not perform optimized reordering of instructions as this is done in other modern processors [40]. As a result, the pipeline length could be restricted to 23 stages, while instantly reducing the residence time in one stage by about a factor of two compared to earlier designs. These 23 stages divide into six stages for instruction cache and buffer, six for instruction decode and issue, and finally eleven instruction type specific backend stages.

Since this model can be seen as a two-way multiprocessor with shared dataflow, it presents itself as a dual-core processor towards the operating system. Each of these threads can access a 32 element 64 bit general purpose register, a fixed point unit, as well as a load and store unit. A floating point unit extends this setting by a decoupled pipeline for vector and floating point operations, which is designed on a 128 bit wide dataflow. It owns a 32 times 128 bit register file, and can operate on vectors of varying element width, such as 2×64 bit, 4×32 bit, 8×16 bit, 16×8 bit, or 128×1 bit. By processing a whole vector at once, this architecture hence offers the opportunity to use a way of instruction level parallelism besides standard sequential implementations. Details on these so-called *SIMD* extensions are more important in context of the SPUs and are described there (cf. Paragraph 3.5.3).

3.4.2 PPU as an Application Controller

Regarding the role of the PPU with its partition of the overall computing power the Cell processor supplies, it is soon clear that it is mostly concerned with running the operating system, and in particular also the graphics engine, which needs to be entirely emulated in software (cf. Section 4.1). However, its prioritized position with respect to explicit access to the RAM enables it for another important application.

When a program is executed by the operating system, it is for the time being a pure PPU-based task. At runtime, the SPU kernels are then sent to their destination and memory is allocated from PPU side. The PPU is thus always aware of the entire memory load allocated in RAM, and also manages these amounts based on information and requests from the SPUs. Therefore, also the means for synchronization available in the Cell chip are partly PPU-oriented, i.e., synchronization among SPUs primarily takes place by using the PPU as a relay station. These methods are detailed in Section 3.5.6.

This way, the PPU can be regarded as an application controller not only dispatching and collecting SPU threads when a program is started or terminated, but also taking care of necessary synchronization between SPUs, managing allocated memory blocks, and even providing active scheduler capabilities if sophisticated scheduling by means of trigger messages is required at some point. Though it typically does not immediately participate in computations, it still forms both reference clock and memory management unit, and provides the user interface by means of graphical output and system interaction.

Anyway, as experience shows, the actual distinction of jobs better to be assigned to PPU, and those more applicable to the SPU is sometimes nontrivial and can cause severe problems. For instance, if data to be read by the SPUs resides in the PPU's L2 cache, the Memory Interface Controller (MIC) redirects access on this information to the PPU's L2 cache, and bursty traffic can exhaust the PPU's own link to the EIB [48]. In course of this thesis, some of these problems are discussed in more detail (cf. Paragraph 6.4.4).

3.5 Synergistic Processor Element

In contrast to the PPE being eligible for complex multithreading purposes, SPEs are designed for pure arithmetics. Hence, they are no traditional multi-purpose core, but represent a compromise of this type and a special purpose architecture.

The instruction set used is still very close to the PPE correspondent, and the general structure of programs does at first glance not need to be modified too much in order to compile it for an SPE. On the other hand however, several novelties in the architecture cause mandatory extensions in the program flow, and in order to make use of the full performance provided by these cores, a considerable amount of manual work is inevitable. Details of these differences to be paid regard to are discussed in the next paragraphs.

Like for the PPE, SPEs consist of a processor unit called *Synergistic Processor Unit* (SPU) extended by a Memory Flow Controller (MFC) and a cache called *Local Store* (cf. Paragraph 3.5.2). Because the difference between SPU and SPE is primarily founded in a constructional view and can hardly be distinguished from a software side, these two terms are in literature often used as synonyms.

3.5.1 Architecture Overview

The SPUs are designed to compute natively on 128 bit wide SIMD vectors [19]. For this purpose, they are equipped with a 128 times 128 bit general purpose register file. This is either used by 128 bit vectors homogeneously splitted into various element types, or by scalars which reside in the cells aligned to the most significant bits [31]. In latter case, all remaining SIMD slots are filled with meaningless values.

Computations upon these register contents are performed by two execution pipelines, often referred to as *odd* and *even*. The even pipeline contains both the Fixed Point and the Floating Point Unit, and is thus designed for arithmetics. Simple integer operations therein are said to have two cycles of latency for dependent values, single precision operations six, integer multiplications and conversions between integer and float values seven [11]. All those operations take one cycle to execute, such that by intelligent scheduling of independent operations, the latency gap can be bridged by some other instructions. Double precision arithmetics are more expensive, since they cannot be fully pipelined and hence need seven clocks, besides introducing an additional latency of six clock cycles. They are thus about one order of magnitude slower than single precision computations and should be avoided whereever possible. To be nevertheless able to yield results with highest possible effort, several approaches can be found in literature proposing to apply iterative refinement strategies, like the approach of Langou *et al.* [45]. The main idea behind such mixed precision strategies is to compute on single precision values as long as this does not deteriorate the results too much, and to finally refine these intermediate results in double precision. This way, results are reported to be achieved about ten times faster than computations entirely performed in double precision, by keeping the accuracy constant.

The odd pipeline in contrast consists of the Permute, the Branching, the Channel and the Local Store Unit and thus covers the nonarithmetic part. They perform loads and stores, branch hints, channel operations and Special Purpose Register (SPR) moves with six, and all remaining operations with four cycles of latency. The actual execution time of any of these operations takes one additional cycle, each. The division of both pipelines allows to process a pair of odd and even instructions synchronously, i.e. one value can be fetched from the Local Store (cf. Paragraph 3.5.2), while another is currently being processed.

Information about this special configuration is worthwhile, since it enables the programmer to react on this situation by applying optimizations towards a minimization of latencies. Throughout this thesis, this concept is used several times, since it allows to speed up processes just by arranging for a balance between odd and even instructions (cf. for instance Section 6.5.2).

By *channels*, SPUs communicate with other elements on the ringbus. From a hardware perspective, channels are unidirectional links between the SPU and external devices, like the SPE's MFC, and are to a certain degree buffered. This buffer is called the capacity of the channel and describes how many entries can be written at once before further writes are stalling and are thus introducing latency. Reading is always blocking if no data is in the channel [33].

Since all non-local operations like DMA requests or inter-core communication control is performed via dedicated channels, meaningful aliases for those specific calls are provided by `libspe2`. Paragraph 3.5.5 goes more into detail about these intrinsics.

3.5.2 Local Store

Random Access Memory (RAM) is a typically volatile storage technology being applied in most modern computer systems. It is significantly faster than for instance hard disk or remote network resources and returns queried data sets in constant time, this means independently from the position they are stored at, or from their relation to the last set requested. Unfortunately, RAM is also a lot more expensive in production cost and space consumption compared to mass storage devices of equal capacity. Along with the fact that it is eventually erased as soon as the power supply is disrupted, RAM is seldomly used exclusively in a system, but can often be regarded as a cache for the data contents stored on hard disks.

Data needed to compute upon as well as the program itself is copied into the RAM, from where it can then be comfortably accessed. Does the physical capacity not suffice, the hard disk is often still used as a slow backup solution. In terms of memory management, this process is called swapping and nowadays handled in dedicated hardware cycles.

Unfortunately, the modern RAM technology is still quite slow in response and transfer times compared to the speed processors can be run at, which would immediately lead to stalls on open memory requests. This is bad, since typical program characteristics often involve more than one access to a certain variable within a short time span, this means, within a local neighborhood of instructions processed. The memory hierarchy is therefore continued in even faster, but still significantly smaller intermediate levels of caches caches residing on the processor chip itself, and the general purpose register file can be regarded as the fastest, but also smallest unit within this hierarchy.

Parts of this hierarchy are shared by all computational kernels on the different SPUs, as well as by the operating system kernel running on the PPU. Besides any external or internal mass storage media, this holds in particular for the RAM which is also the lowest instance providing shared memory access for different kernels to the same resources. In addition, each processor core is equipped with an own dedicated cache used for intermediate results and fast working copies. On SPU side, these caches are called *Local Store* and have an overall capacity of 256 Kilobytes, which covers both instructions and data being used by the kernel.

This concurrent design is not only one major argument for the performance gain possible to achieve for several programs by porting them from any traditional to this architecture, it also introduces several potential hazards known from multi-threaded software design on conventional computer systems. These are on the one hand so-called race conditions in the narrower sense, describing the necessity to synchronize concurrent access to the same resource such that operations are performed on it in order. But even for independent writing access to a certain memory cell, as it might hold for separable problems, it is inevitable to find a suitable coordination such that results developed by one kernel do not interfer with those written out by another, i.e. that the whole setting remains consistent at any time.

Therefore two main aspects need to be considered at any time, namely availability and validity. While the first term describes the fact that data to be accessed needs already to be loaded into the cache beforehand, latter refers to the version of these data sets with respect to changes performed by other cores.

This problem is also known from traditional architectures, and is in this context solved by a hardware logic. In such conventional managed caches like the one implemented in the PPU, vectors of validity information are kept alongside the cached memory lines to distinguish 'dirty' lines, i.e. such not synchronized to RAM yet, from 'fresh' ones. Whenever a line is requested, the circuit then decides whether the memory line is still valid in cache, and provides the cached version, or fetches the respective line from RAM first. Unfortunately however, this logic works without any knowledge of the algorithm being executed, and does hence not know anything about its characteristics and access frequencies to certain values. It might hence happen that a line is being moved out of the cache, just before it is then going to be used again a few cycles later. On the other hand, lines not to be used again can reside for too long in the cache, thereby keeping other lines from being handled more efficiently. In such situations, managed caches can introduce a high overhead in both time and memory bus usage.

Since the PPU needs to assure categorical conformity with the existing PowerPC specification, it necessarily also needs to rely on a managed cache hierarchy. Meanwhile however, chip designers have not been bound to any of these conventional criteria when creating the layout for the SPUs. To enable programmers to accelerate their software on this distributed architecture and to avoid unoptimal cache situations like described above, they made any transfer of data between RAM and the Local Store explicit. Only instruction fetch, which is uncritical because the program flow can easily be planned and foreseen at any point, is still performed by a circuit.

This allows for very efficient programs, since the programmer can decide at any point which values are meant to reside in the cache, and which are not. More importantly however, developers can already request memory lines, when they are not needed yet, such that they are available when the first request to them is going to be scheduled. A drawback of this method is however, that the entire responsibility over memory management and in particular also data synchronization is inflicted to the programmer, independent of how trivial or how time-uncritical an operation might be. This means that even for operations without performance requirements, any memory interaction needs to be explicitly modeled. This issue can not only expand the programming code significantly and deteriorate its general appearance and lucidity, but can thus also be a potential source of error.

Meanwhile, the Local Store still remains a cache immediately interacting with the Memory Flow Controller, which coordinates access to the physical RAM on a low level basis. In particular, all transfers of data in between these two components must be ensured to be compliant with respect to memory word *alignment*:

Alignment means in this context that both the address and the length of the data block handled must be being a multiple of 128 bits, both in RAM and Local store. This results in the addresses and length specifications to end on a series of at least 4 zeroes in binary representation:

$$128\text{b} = 128\text{b} \cdot \frac{1\text{B}}{8\text{b}} = 16\text{B}, \quad \log_2(16) = 4.$$

This principle is commonly used on almost all architectures and allows for larger addressed storage while still keeping addresses in the range of the maximum bus width supported by the processor. This way, tailing zeroes can be easily made implicit and the released digits can thus be used for a higher range of values possible to process.

Unlike most processors, where correct memory alignment is automatically ensured by hardware circuits and invisible to the programmer, special assembly and high level instructions need to be used on the Cell processor to model these requirements in software. One example for such intrinsic for C and C++ is given by `spu_mfcdma32` provided by `libspe2`, which starts a reading or writing DMA access by internally mapping to a sequence of seven assembly instructions [58]. It forms the basis for most cache operations throughout this thesis. Besides others, this operation takes a so-called *DMA flag* as a parameter, which assigns a unique ID to a DMA process. Technically, it tells the MIC to set the respective bit in a dedicated 32 bit register as soon as the requested data is valid. This way, memory contents can be requested, the waiting time can then optionally be bridged by some arithmetic operations on different Local Store partitions, and finally the service register can be triggered for the loaded data to become valid.

3.5.3 AltiVec SIMD

As the consequence of these considerations, issuing equal operations on potentially different data sets hence causes a high number of similarities in the state of the processor. Not only the instruction fetch and decoding phase is similar, the same holds as well for several stages in the pipeline and in particular for the forwarding logic. Processor designers use these observations to extend their architectures by a relatively simple way to speed up many programs, namely *Single Instruction Multiple Data* (SIMD) extensions in terms of Flynn's taxonomy [22].

Figure 3.3 shows an example of such operation, based on integer vectors:

`(vec_int4){1,2,4,8} + (vec_int4){1,1,1,1}` $\rightarrow$ `(vec_int4){2,3,5,9}`

Since the operation is of homogeneous structure and source and target values are residing in four adjacent memory cells aligned to 128 bits, this instruction does only need one processor cycle to execute.

Instead of working on single values only, the full bandwidth of operations is extended to a multiple of scalar data types, hence yielding vectors of a fixed number of equally typed elements. In terms of the SPUs, the width of operations is fixed to 128 bits, by offering either sixteen **`char`**s, eight **`short int`**s, four single precision floating point numbers (**`float`**) or four **`int`**s, as well as two double precision floating point numbers (**`double`**) and the unsigned

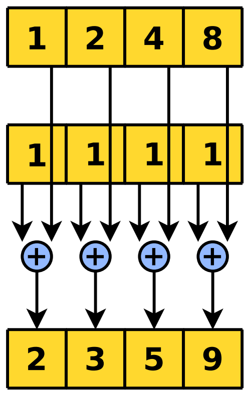

Figure 3.3: Example execution of a SIMD based addition of two integer vectors.

versions of the respective fixed point notations. Except for the double precision extension, this concept has already been used to enhance the performance of PowerPC architectures using a SIMD extension published under the product names AltiVec, or Velocity Engine Multimedia Extension [56, 51]. Originally designed for the PowerPC series produced by IBM and licensed by Apple to be built into the MacBook series notebook computers, the brand name AltiVec is today held by Freescale.

By making parallelism explicit, the programmer can map structural characteristics to the hardware like this has already been possible on traditional vector computers in earlier times. However, while latter computer systems were designed for much higher element counts and were thus often dedicated to special purposes, a significantly lower bandwidth embedded into a classical sequential processor layout creates a compromise between the benefits of both approaches.

Hereby, such vectors need to span in adjacent memory cells and must be aligned to 128 bits. Any instruction issued on these variables is then handled on a per-vector basis, which means that any of these SIMD vectors is in principle treated like scalar values with respect to data flow inside the processor, except for a higher bandwidth and redundant computation units where the full bandwidth is needed.

The instruction fetch and decoding stages involved in processing of such SIMD instruction do not differ between scalar and vector based operations, because the wider data width does typically not project onto a much different instruction format, as still only one single command is described by it. However, the different inter-stage registers and connecting datapaths in the execute and write back phases are indeed necessary to be adapted to the new bandwidth. To perform computations on all partitions in parallel, a sufficiently high number of integer and floating point units must additionally be provided, as well as an extended branching logic connected upstream to become capable of the different element sizes a SIMD vector can contain. Such circuit is needed to sort and split the 128 bit of width into for instance eight 8 bit wide character components, into four single precision floating point numbers, but is also needed in traditional designs to assign floating point numbers to the floating point unit, while any other data type is sent to the fixed point arithmetic and logic unit. Technically, this process typically comes down to a generation of dedicated driver signals activating or activating certain units, or setting them to a mode for the special data type to be processed.

At this point, one major difference between traditional architectures and the Cell processor becomes obvious: While in traditional architectures, SIMD vectors and scalar values are handled differently, the SPUs of the Cell processor always work in chunks of 128 bit, whereby for scalar operations, the rightmost bits are just containing unmeaningful data. This is also the reason for the Cell being marketed as a *Broadband Engine Processor*. Special assembler mnemonics and also immediate high level constructs (cf. paragraph 3.5.5) exist to manage the transitions between scalar valued and vector valued data. Switching between these is indeed very often necessary, because for instance on the one hand, many computations and comparisons on values can be performed in full width, but when it comes to decisional jumps, the branching condition needs to be accumulated as a scalar residing in the first 4 bytes.

3.5.4 Restrictions

For full SIMD support, the SPUs possess a floating point unit capable of four single precision operations in parallel, but only one double precision operation per clock cycle. Because arithmetics is always performed on full 128 bit width, double precision operations cause both double word wide halves to be processed one after the other, independent from whether the second half contains meaningful values or not. This means that even for instructions with scalar range, the second half will still be processed by the SPU, counteracting the pipelining principle: Since both halves need to be fed into the unit one after the other, arising dependency problems are solved in a straightforward manner, namely by stalling both pipelines for additional six cycles to bridge the latency between the first and the second half of the SIMD vector. This reduces the double precision performance significantly compared to single precision computations. In contrast to a single precision pipeline length of seven cycles including six cycles of latency, the double precision pipeline thus lasts 13 clock cycles, whereof again only six can be reused by other instructions. It is hence always advisable to fall back to single precision whenever it is possible, or to use mixed precision approaches (cf. Paragraph 3.5.1).

Besides the lower accuracy in computation and the potential arithmetic errors arising from this fact, the SPU floating point unit does not completely support the IEEE floating point specification, which introduces further errors. In particular, it only supports round-to-zero from the rounding modes specified in the IEEE 754 standard, which is tantamount to truncating the binary representation to the maximal length of the mantissa [30]. This restriction is not imposed to the double precision computations which support the full range of IEEE rounding modes, and only partially applicable to the PPU, supporting both round-to-nearest mode and an emulation mode for SPU accuracy called *graphics rounding mode* [58]. For single precision computations on the SPU however, this deviation affects many C math library functions like `floor`, `exp` or `log`, which finally renders arithmetic results only to approximate the more exact solutions yielded on a x86 based CPU, or the PPU.

Furthermore, the conception of the SPU SIMD extension implies the special values *not a number* (`NaN`) and *infinity* (`Inf`) to be handled like normal numbers, this means like the number their internal representation would suggest. Because all operations on the SPU are performed in SIMD, this again applies to all single precision floating point arithmetics and can introduce further inconsistencies if relevant special cases are not caught in software.

In paragraph 3.3, the performance problem related to branch mispredictions has already been mentioned. In particular, any mispredicted branch causes the speculated branch to be entirely flushed from the pipeline, thus introducing a penalty of approximately 20 cycles [30]. Most modern processors hence possess a branch prediction mechanism by means of hardware lookasides like branch history tables, branch target address caches, or branch target instruction caches. Unfortunately, the SPUs does not utilize any of these techniques, but instead relies on a programmer directed branch prediction. Hereby, the branch speculation is already established at compile time, whereby the speculation later assumes unbranched, i.e. linear continuation of the program at the respective points, if no definite hint by the programmer has been carried out. Developers can hence make use of several remedial measures to reduce mispredicted branches.

The most effective way to do so is certainly to reduce branching to a minimum, by application of techniques like loop unrolling, function inlining or select bit instructions [49]. For loop unrolling, several iterations of a loop are performed one after the other without intermediate branches. In program code, this implies an instruction in the loop body to be coded several times with adapted parameters, while contingent loop counters are always increased by the degree applied. Function inlining means that a function does not allocate a frame on the stack, but the body is embedded into the calling routine at compile time, allowing for both easily readable code and highly performant running behaviour. Select bits instructions are finally a hardware assisted way to entirely avoid branching for particularly short branches, namely by computing both branches towards intermediate results, independent from which branch is actually referred to and then selecting the one intermediate result that would actually be computed if the branch was taken. Latter can be done with a single logic operation and is represented by the `selb` assembly instruction. Of course, this concept only pays off if the time it takes to unconditionally process both ways is still shorter than the expected value for the average loss due to branch mispredictions.

Loop unrolling has a second accelerating effect to the program: Since floating point operations are typically associated with latencies of six cycles, five independent instructions should be scheduled in between two dependent instructions, since these cycles are otherwise allowed to lapse [50]. By a loop unrolling approach with a sufficiently high assortment of independent instructions in the loop body, the compiler can reorder these instructions in a way that they are interleaved and hence use latencies most efficiently.

A second way to reduce the overall number of mispredicted branches is to influence the order conditionals are evaluated, since the processor is designed to assume that always the first branch would be taken. While this is indeed an option for programs written in assembler code or included into the high level language as such, it can cause quite unpredictable behaviour during the code generation, if the software is entirely written in a high level language like C. Depending on the compiler and its optimizations made, the order of independent code fragments can still be shuffled, thus reverting the improvements made by the developer.

To circumvent this behaviour, compilers offer so-called *branch hint instructions*. Here, the programmer can specify what he anticipates the conditional expression to evaluate to, such that the compiler can reorder the branches accordingly. Due to this predefinition at compile time, these instructions are typically referred to as *static* branch hints, distinguishing them from hints evaluated at runtime. Latter are also supported by the SPU, but only if they are explicitly embedded into the program code, represented by the `hbr` assembler mnemonic family. Their own execution needs one cycle and they must at least be placed 19 instructions before the branch is executed, such that the core has enough time to react on this specification. As soon as this manual prediction is correct, penalty-free branching can be ensured. At the same time, a 20 cycles misprediction penalty is imposed on the contrary branch, such that cautious deliberateions at design time are valuable.

Comparing the whole concept of the Cell Broadband Engine to traditional symmetric multi-core processors, the specific design of the SPU implies an entirely different programming style. Even though this fact forms the main argument for the Cell and also explain its fame as an object of research, it should also be mentioned in this section, because programs are in general not as easy portable to this platform as they are to many others. Besides a completely different binary layout which is pointed out in more detail in section 3.9 and a new instruction set for the SPUs, several other aspects underline the pure number crunching objective of the whole SPU design. They are for instance not multithreading enabled and can also not immediately be triggered by the BIOS, thus making them dependent from an operating system running on the PPU. Because the program kernels are always entirely transferred into the Local Store, the SPUs are also fundamentally useless as coprocessors to run system tasks which are typically very bursty in resource allocation and therefore heavily shuffled with other background tasks. However, as soon as a dedicated system task can be identified to be very computation intensive and permanently running, as it might be the case for digital signal processing implementations, this task can be formulated in terms of an SPU kernel and swapped out to one single SPU. Even though this SPU would then no longer be available for joint computations, this method can indeed be worthwhile, because these dedicated processes are typically less constraint to synchronization with the rest of the processor, and can hence be assumed to run with high performance.

3.5.5 SPU Intrinsics

Programming the SPU in a high level language needs to meet different requirements than writing programs for a traditional processor. While for x86 based architectures, branching does not play as an important role and an explicit cache management is at best marginally touched by means of software design patterns, like those optimizing for the cache direction in multidimensional data sets, such considerations are indeed necessary when developing for the Cell, and in particular for one of the SPUs. To be capable of these new machine-dependent features, SPUs do not follow an established instruction set specification, but implement a new restricted instruction set, which shows strong similarities to the instruction set used on the PowerPC platform [32, 33].

Conspicuously, arithmetic and logical operations are always designed to have a scope of the full SIMD vector and therefore replace their scalar counterparts in other assembly languages. In addition, special instructions model data transfers between the leftmost vector entry and the remaining slots, to convert between explicitly scalar and vector-based applications, or shuffling applications. In principle, latter are nothing more than a natural extension of bitwise logical operations on traditional hardware to a 128 bit width, though working on a greater atomic element size. They are helpful for a fast reordering of single bytes whenever across-slot movements need to be performed, and can be combined with conventional bitwise operations to achieve a finer granularity. In Paragraph 6.5.2, these instructions are applied to a concrete example.

In particular, while operations like cyclic shifts still exist on a per-value basis, byte shuffling can be used to move bytes over element boundaries and hence overcome the problem of alignment issues. Without these techniques, operations between two different elements in the same vector could just not be performed at all, since there would be no way to align both values in the same bit range of the processing unit.

However, the increased bandwidth is not the only extension to be considered with respect to the new instruction set. Moreover, the situation of joint computing on SPUs as a pre-stage of special purpose hardware requires an additional group of commands to be implemented as well. Besides control over the memory flow controller taking care about RAM accesses and Local Store management, various synchronization techniques must also be addressable from the software side. These operations are actuated by channels that can be written to or read out, while reading access can be set blocking to wait for an external synchronization or memory event to happen.

Constructing compilers for this setting, one has to be aware of the fact that most convenience accommodations to the programmer come along with performance losses, which holds in particular for the adaption of existing sequential languages to this architecture. On the other hand, it is always desireable to fall back to widely spread languages, to profit from synergy effects from the community and to increase acceptance among developers. In past times, this issue has often been addressed by means of inline assembly code, this means the high level language is indeed used to describe the general program flow, but whenever a resource critical functionality is needed, it is immediately coded in embedded assembler mnemonics. In practice, this technique often leads to whole routines being entirely written in assembly language to ensure a high performance, which can render work to be tedious due to the missing support of loops and other high level structures.

The designers of the SPU development kit abstracted from this problem by giving the programmer high level commands at hand which are guaranteed to immediately map to a dedicated assembler instruction. These special functions are called *SPU intrinsics* and hence fill the gap between actual low level programming and modern, widely spread languages like C or C++. They are extended by a variety of other new commands mapping to a sequence of, instead of only one, assembly instructions. Because both types are quite equal in usage regarded from an application developer's point of view, they will not be further distinguished in the course of this thesis. By including `spu_intrinsics.h` into the program source code, the SPU specific extensions are introduced to the framework.

Making the internal data structure explicit, declaration of data types has been extended by several new SIMD vector data types, while scalar values can still be used. The most common vector types in this context are `vec_float4`, consisting of four single precision floating point numbers, and `vec_int4` being constructed out of four concatenated integer values, as the name suggests. One should however always be aware of the fact that scalar types will be fetched into the first few bits of a 128 bit register and internally processed as a vector with three unmeaningful entries, whereas vector types are explicitly handled as such and are always fetched vector-wise. The compiler observes this difference by means of typechecking and does not allow implicit casts between both interpretations.

In paragraph 3.5.2, the special alignment of data to be synchronized with RAM contents has been outlined. This requirement needs to be heeded all time when allocating storage in the Local Store. For dynamically allocated memory, `_malloc_align` can be used to ensure this fact, while the compiler's alignment attributes have the same effect on local variables. For instance, the following sample code fragment allocates two integers aligned to $2^5 = 32$ Bytes, one of them dynamically and one statically:

```
int *dynamic_int = (int*)_malloc_align(sizeof(int), 5);
int static_int   __attribute__ ((aligned (32)));
```

For simple arithmetic or logic operations, there are dedicated SPU intrinsics available as well. Anyway, these intrinsics are seldomly used, because most infix operator symbols have been overloaded with their respective vector correspondences referenced thereby, and are ususally way easier to read than prefix functions. Code using these symbols looks subjectively better arranged than a pure intrinsic-oriented view which is often noticeably longer and can sometimes even confuse the reader by means of functionally nested terms.

In contrast, memory management and synchronization cannot be performed without any use of SPU intrinsics. Interactions between Local Store and RAM consist of two parts, namely a non-blocking initiation process, later followed by a blocking query for finalization. While memory interactions are running, the SPU can hence continue working on existing data sets to bridge the time gap until the DMA operation comes to an end.

To be able to distinguish different simultaneous DMA accesses usually initiated by the SPU macro `spu_mfcdma32`, they are annotated by an integer tag between 0 and 31. Whenever this access finishes, the respective bit in a dedicated 32 bits register is altered and can then be acknowledged by the blocking finalization call, consisting of a tag mask setting via `spu_writech`, followed by a `spu_mfcstat` query.

3.5.6 Synchronization

For synchronization between SPUs and the PPU, two different methods are offered by the memory flow controllers (MFCs): *Mailboxes* and *signal notification registers* [35]. Every SPU can use three buffered 32 bit mailboxes, whereof two outbound boxes are readable by the PPU and writable by the SPU, and one inbound mailbox can only be written by the PPU and read by the SPU. Among the outbound mailboxes, one causes an interrupt at the PPU side whenever it is written, this is why the PPU can issue blocking reads on this special mailbox.

Writing calls to the SPU's inbound mailbox can be either blocking or nonblocking. On the SPU side in contrast, reading is always blocking, while writing is not as long as the mailbox buffer is not full.

Using this mechanism, SPUs can already be synchronized to the PPU, which hence plays the role of an application controller, external trigger and resource manager. However, since an SPU cannot write to any other SPU's mailboxes, this mechanism does not allow for inter-SPU synchronization without employment of the PPU. Nevertheless, this functionality would indeed be desireable for certain applications, for instance if the PPU can not be guaranteed to be available for synchronization all time.

Here, the signal notification registers come into play. Every SPU owns two more dedicated 32 bit registers in the MFC, which can be written to by any core by means of memory-mapped input/output (MMIO) devices. These registers are unbuffered, but can then be set to accumulating or replacing mode, whereby accumulation refers to data being added by bitwise 'or', while replacing overwrites the existing value instead. The SPU can fetch the whole register contents at once using a channel operation, meaning the value is copied to a Local Store location and the signal notification register is instantly flushed to zero. Depending on the application, it is hence possible to interpret the $2 \cdot 32 = 64$ bits as single 1 bit signals, which can then independently be triggered, accumulated in the SPU's MFC and disassembled by simple masking operations.

Making use of any of these techniques require at least once the participation of the PPU, namely to exchange addresses of the adverse SPUs' interfaces. They are coded into the SPU's context (cf. paragraph 3.4.2) and can either be published to the other SPUs by means of DMA access, mailboxing or signaling, or stored inside the PPU if its permanent role as an application manager is favoured.

3.6 Element Interconnect Bus

PPU, the SPUs, as well as the processor bus and memory interfaces are connected by a ring bus, the so-called *Element Interconnect Bus* (EIB). It consits of four unidirectional rings in total, two of which are running clockwise, and two counter-clockwise. In addition, a data arbiter is connected to all clients regulating the bus access on a hardware level [20].

With a total of nine cores requiring to be connected to each other, a common system bus interface and a RAM interface, traditional front side bus constructions are no longer applicable without losing performance. For instance, a 64 bit wide FSB with 8 transfers per tick and 133 MHz clock frequency is just capable of about 8.5 GB/s of bandwidth, which is too less for the theoretical amount of operations the cores can issue altogether. Therefore a new design was to be found that is able to cope with routine data bursts.

According to David Krolak [5], the lead designer of the Cell's EIB, early evaluations included a crossbar switch potentially to be used for these purposes. This switch would immediately interconnect any pair of clients attached and therefore reduces the communication delay to a minimum. However, such implementations always cover a huge chip area which could just not be afforded in the Cell BE setting. Instead, the mentioned ring bus has been designed, which only occupies a small fraction of the crossbar switch's space requirements, but which is still held interface compatible to the initial plannings.

It can hence always be replaced by a matrix switch whenever future developments render this step inevitable, without necessary core redesigns. Hereby, the client interfaces do not know that there is actually a ring structure underlying, but use a simple handshake protocol invoking the arbiter whenever they try to send data to any other port. They are then granted access to the one bus connection the arbiter then considers to be best suitable in the current setting and just send their data packages into the network, where they are then routed by means of arbitration signals.

As a drawback, the ringbus only connects adjacent clients immediately in a physical sense, which implies several cycles to be taken when different ports need to interact with each other. On the other hand however, the bus can more easily be used to full capacity this way. While for the crossbar switch, most connections are unused and thus idle to an arbitrary point in time, the load in the ring bus is usually significantly higher. From a client's perspective, there are often either own or external data packets running over the bus interface with integrated repeater circuit occupying inbound and outbound connections alike.

The EIB runs at half the processor's clock and has a maximum accumulated throughput of 96 bytes per processor cycle, which is given by 12 client interfaces processing 16 bytes per bus cycle, each. For any of these ports, a theoretical bandwidth of 25.6 GB/s can be achieved per direction. As long as Memory Interface Controller (MIC) and Bus Interface Controller (BIC) are fast enough, all cores hence share a maximal bandwidth of about 102.4 GB/s to and from either of these ports. Assuming equal distribution of these bandwidth concessions over all cores, there are still resources left to be used for the inter-core communication.

However, this theoretical bandwidth of the Cell BE is not always attainable. Since the rings resemble a shared resource, two concurring transactions on the same physical medium can block each other. Even though this problem can be reduced by a manually devised association of kernels to physical SPUs and a detailed analysis of timing diagrams of the different kernels, it can seldomly be completely eliminated, which in particular holds for complex programs.

Another frequent bottleneck is formed by contemption on the interface devices themselves. Is a single device overwhelmed with arriving data packets to process, the queues meant to buffer short data bursts can run full and thus stall other units in their work. This problem can in particular be observed at the memory interface controller providing not more than 25.6 GB/s, if too many memory interactions are issued. This problem is again going to be discussed in context of the Red-Black SOR approach developed in this thesis (cf. Paragraph 6.5.8ff).

3.7 Processor Bus Interface

For the I/O bus driven by the Bus Interface Controller (BIC), the Cell developers decided to license the FlexIO™ system designed by Rambus, Inc., which has then been, according to Rambus' own submissions, the fastest processor bus in the world [4]. While traditional front side bus systems are typically clocked lower than 500 MHz, this new layout allows for speeds up to 8 GHz by applying a circuit solution for flexible phase relationships between processor and bus clocks [72]. In the Cell Broadband Engine, the processor bus is clocked with 5 GHz, providing maximal bandwidths of 35 GB/s outbound and 25 GB/s inbound [19]. Data is hereby exchanged in packets, thus offering the opportunity to make use of dedicated flow control tags passed in packet headers.

In the Playstation 3, this bus only provides access to the RSX graphics card and other system devices, but multiprocessor layouts can in principle be realized as well. In particular, glueless two-way symmetric multiprocessor layouts, as well as four-way symmetric multiprocessor designs using a central switch are officially pronounced by IBM [40].

3.8 RAM Interface

Like for the processor bus, the STI consortium licensed a Rambus, Inc. technology to be used for the memory bus. The used XDR™ memory architecture provides a peak performance of 12.8 GB/s per 32 bit channel, of which two are available at the Memory Interface Controller (MIC) of the Cell BE chip [40]. However, in frequently changing read and write access scenarios, the effective bandwidth can boil down to about 21 GB/s due to introduced overhead when inverting the direction of the data bus. Additionally, one more GB/s is typically spent for usual memory operations like scrubbing, this means inline detection and correction of flipped bits, or memory refreshes. The effective bandwidth can hence be assumed to be about as much as 20 GB/s, if the accesses are equally distributed on all banks.

The maximal RAM size accessible by this system is limited to 512 MB in total, split into 256 MB per XDR channel. The current configuration of the Playstation 3 does currently not exhaust this limit, but is equipped with 256 MB of RAM, of which only about 200 are accessible under Linux (cf. section 2.2).

3.9 Binary Layout

In sections 3.4 and 3.5 it has been shown that the assembly languages of PPU and SPU differ in considerable parts, because both the intended purpose and the general behaviour differ too much to be unified in one software concept. There are hence different compilers for PPU and SPU programs and the binaries created by them are not exchangeable among the two core types. In particular, binaries for the PPU can immediately be executed by the operating system, while SPU kernels first need to be transferred to and dispatched on an SPU and cannot be interpreted by the OS at all. Latter then typically need to rely on memory allocated and initialized by a PPU program, which they are able to access by DMA operations.

To simplify this mechanism, a combined PPU and SPU binary format has been designed. Therefore, fully compiled SPU binaries are embedded into the PPU target, which is furthermore extended by a wrapper first pushing the SPU binaries to their destination, then dispatching the SPU kernels and finally also starting the actual PPU program. This furthermore allows the SPU kernels to be regarded as POSIX threads, as which they can then be debugged using tools like **`gdb`** (cf. paragraph 4.4.1).

As a drawback, these combined binaries cannot be natively tested on the Full System Simulator described in section 4.3, but always require a running linux kernel. However, due to the independence of the actual programs from their later representation, it is always possible to use the single targets apart from each other when this is necessary. In the toolchain, these targets are generated standalone anyway and are later linked together using the **`ppu-embedspu`** tool, such that it is easy just to take the SPU binary and to test it separately if needed.

3.10 Summary

The Cell Broadband Engine is a multi-core processor with promising performance specifications. However, the Local Store available on each SPU must not tempt the programmer to regard the cores as multi-purpose coprocessors or even as clustered computers. Instead, they are actually numerical coprocessors acting on explicit memory working copies and natively operating on a higher bandwidth than usual processors. They are thus already quite close to special purpose hardware.

Especially when it comes to processing of large fields of data in a distributed manner, boundary values typically do not need to be ‘exchanged’ because all cores can operate on a shared memory dataset, but need instead be taken care of by means of appropriate synchronization. To enable DMA accesses, data needs to be aligned by at least 128 bits.

From a local view, this means that particular attention needs to be spent to data validity issues with respect to working copies and the actual memory contents, and that necessary DMA operations are timed accordingly to reduce delays. Working copies always need to be dimensioned as a compromise between moderate bus traffic and the restricted cache size. Whenever operations can be formulated in 128 bits of width, this notation needs to be evaluated against sequential forms by means of running time complexity. Finally, it is worthwhile to always keep the two pipeline scheme of the SPU in mind, to recognize and multiply situations where both pipes are fully occupied to enhance efficiency.

In Chapters 6 and 7, these concepts are taken up again to develop efficient parallel algorithms for the optical flow problem presented in Chapter 5. Beforehand, Chapter 4 is going to show how to divert the Playstation 3 from its intended use to adopt it to scientific computing.

Chapter 4

Development Platform

4.1 Linux

Though the Playstation 3 is primarily designed as a video console, Sony allowed from the start to install a third-party operating system on it, and even offers a dedicated menu entry in the shipped firmware *Game OS* to do so. Sony calls this option *Open Platform*, and maintains a website where an appropriate boot loader to be installed is provided for download [57]. The list of options for this third-party system is however short, since it needs certain extensions to be run on the console and proprietary solutions often lack these features.

However, because the Linux kernel has successfully been running on PowerPC architectures for quite some years and it is furthermore open and easily configurable, several distributors have started to adopt their products for the Cell Broadband Engine and in particular the Playstation 3. They are hereby officially supported by Sony, who maintain a kernel.org subtree containing adapted Linux kernels for the Cell processor [43], as well as the *PS3 Linux Distributor's Starter Kit*, a collection of software tools needed to get Linux to run on the Playstation 3. The first complete package for the end user market was Terra Soft Solution's Yellow Dog Linux 5.0, and the Fedora distribution developed under contributions of RedHat, or recent Ubuntu derivates followed immediately afterwards [50].

Investigating the characteristics of these products, the relatively small software selection provided on the installation media and in online repositories of Yellow Dog Linux does not convince. Especially if the system is meant to be used as both target system and compilation host, i.e. a recent version of the Cell SDK and appropriate compiler toolchains are meant to be run on this platform, Yellow Dog Linux has proven not to be applicable. There are indeed ways to include alien repositories like the one offered by Fedora 5, from which Yellow Dog is derived, but these showed incompatibilities to the original installation and partly also lead to unrecoverable dependency clashes. Particularly, IBM's full system simulator with its graphical frontend in version 2.1 depends on several third party libraries and the conflicts with distribution-related programs can hardly be resolved at all.

Because of these experiences, Fedora 6 has finally been chosen as a development and testing platform for the rest of this thesis, and has been extended by the official Playstation 3

Add-On CD offered by Geoffrey Levand from Sony [43]. These Add-Ons are for instance needed to set the video output of the device, or to reboot into the firmware again.

Since this setting does not provide any kernel sources and the precompiled kernel binaries are optimized for the desktop market rather than for scientific computing, a kernel with personalized configurations has been compiled and installed subsequently. This provides both matching sources to the kernel binary, as well as certain improvements on the performance. By changing from the generic to the personalized kernel, using a lightweight window manager, and reducing background tasks to a minimum, the performance could be increased by up to 19% in some places.

Unfortunately, the third party system does not have full control over all hardware components. Like already mentioned in Section 3.2, a hypervisor restricts access to certain components and thus protects the firmware from being modified and the security policy to be violated under Linux. To do so, it creates a virtualization layer over the hardware allowing access to most devices, like external buses, the sound card, the wireless LAN and bluetooth chipsets, and finally the Cell processor. However, only the hard disk partition reserved to the guest system is visible, and out of the graphics pipeline on the RSX chip, only the framebuffer can be written, but no shaders or other stages of the pipeline are accessible. Because this hypervisor is said not to resemble a pure hardware solution, but to consist of software components as well, it reserves one SPU for itself [25], besides of one SPU entirely turned off (cf. Section 3.2).

While hard disk restrictions constitute no problem, the lower number of SPUs is indeed disruptive, since it restrains the computation power noticeably. Furthermore, the missing access to the graphics pipeline enforces software rendering, and this diminishes the performance of the cores additionally. While for the simple framebuffer driver, all graphics processing is executed on the PPU, several approaches exist to move the load towards the SPUs, like the Mesa Cell Driver developed by Tungsten Graphics, Inc. [61]. Because latter methods offer the opportunity to provide 2-D and 3-D 'hardware' acceleration to pure PPU programs, they are quite interesting for the desktop applications and games. However, since the SPUs are used for intense computations in course of this thesis, and SPUs can only run one single, manually assigned task at once, such solutions are infeasible for the present setup.

Instead, no option remains at this point than accepting a missing acceleration for graphics output and thus to concede performance losses whenever interactive applications involving graphical output are run. Chapter 8 addresses several of these issues in detail.

4.1.1 Installation

The installation procedure for Fedora 6 follows closely the official tutorial published by Sony [59]. First, the device needs to be prepared for hosting a guest system, which can be performed by a dedicated menu entry. In the following dialog, the 60 GB disk can only be splitted into a 50 GB and a 10 GB partition, whereby the assignment to third party system and firmware is up to the user. Since scientific interests are in the foreground, the variant involving 50 GB for Linux has been chosen in this experiment.

On reboot, the Playstation then searches for a **`kboot`** bootloader image on external devices, such as USB memory sticks, and as it has found one, it boots into this minimal system, from where the installation of the ppc64 Fedora 6 DVD edition can then be dispatched. After the installation of the base system and the Playstation 3 specific add-on, which works like being performed on any other traditional architecture, the bootloader configuration file `/etc/kboot.conf` needs to be slightly modified: The symbolic name `/dev/sda1` for the visible harddisk partition must be entered as the root file system, and the video mode used for HDMI graphics output needs to be set to a convenient mode. Latter can be given a try beforehand using the tool **`ps3videomode`**. For the current setup, 1080 p Full Screen PAL mode at 50 Hz seems to be the best mode for the connected Fujitsu-Siemens Scenicview P24-1W monitor.

4.1.2 Adoption for Scientific Computing

So far, the Playstation can already be used as a desktop computer like any other PowerPC architecture. To optimize the system for scientific computations, various changes have been applied to this installation.

Several background daemons have been deactivated, like this is being proposed by Remy Bohmer [54], such as network related services, automated system upgraders, or the printing system. Because for development and testing a graphical user interface is helpful, a window manager both lightweight and comforting should be installed. A good compromise seems to be given by Fluxbox, which has hence been used for the work created in context of this thesis.

Next, a smaller, application-oriented Linux kernel has been created. It has been configured to contain only basic components, support for the Playstation 3 hard drive virtualization and Cell support, as well as support for huge Transaction Lookaside Buffer pages (*huge TLB pages*) like proposed by Buttari *et al.* [18]. Bohmer therefore suggests to install necessary development tools beforehand by using the distribution-specific automatic update tool **`yum`** [54]. At least under Fedora 6, this attempt emerged to a problem, since **`yum`** took several days to resolve dependencies within a group after downloading the packet headers, and finally reported a dependency mismatch. Instead it turned out that most necessary programs are already shipped and installed with the distribution, and the remaining programs can be easily compiled from source.

4.1.3 Kernel Support for Huge TLB Pages

The idea of TLBs is closely related to the way a Memory Management Unit (MMU) works: To ensure that every task has its own dedicated memory range, the operating system maintains an abstraction layer of the virtual memory. This has several benefits: One the one hand, every program can in principle address the total amount of memory like any other while assuming to work as a dedicated task on a distinct processor, but on the other, this system also ensures that one process can accidentally or maliciously write into memory owned by another process. A third convenient side-effect is that by this abstraction layer, size restrictions of the physical memory can easily be handled as well, since dedicated hard disk regions can transparently be used as swap space [52].

In this translation process, so called *page tables* are used as a mapping between virtual and physical addresses. To reduce the size of this hash table, the memory space is hereby typically partitioned into larger elementary blocks called *pages*. This way, only the address of the whole page needs to be translated, whereas the internal structure of such page stays constant and does hence not need to be translated at all. As soon as a request for a certain memory cell comes in, the corresponding page is searched in the page table and translated to a physical address. If this address is currently in RAM, the respective contents can immediately be returned. For swapped pages on the hard disk however, a page fault interrupt is thrown, and the operating system first exchanges for instance the least recently used page in RAM with the requested page. As soon as latter page is entirely present in RAM, the process can proceed like before.

Nevertheless, page translations can be quite expensive, since page tables nowadays typically consist of several hundreds of thousand entries and requests for a particular entry need some time. This is especially bad, since programs locally often work on a very restricted amount of variables which then need to be translated on and on again. Here come the TLBs into play: Acting as a cache for translations and residing inside the CPU, they store a certain amount of the least recently issued translations and are able to return those very quickly when they are needed.

Such TLBs are a critical resource, since they can only store a certain amount of page translations, which implies that the larger page sizes are used, the more memory can already been mapped inside the TLB and does not need to be resolved in page tables. On the other hand, larger pages also increase local fragmentations of variables inside the pages and do thus only pay off if the data structures allocated are relatively big.

In recent linux kernel versions supporting the huge TLB pages extension, two sizes of page tables can be used simultaneously. On the one hand, these are the 4 kB sizes standard pages automatically invoked if no special call is issued, and on the other, data structures can explicitly be allocated inside a virtual file system called `hugetlbfs`, where data is then instantly residing in greater pages of typically 4 MB [46].

With a kernel capable of this technique, it is finally open to the programmer to speed up computations by explicitly allocating structures in huge memory pages. Performance gains of about 5% are reported on traditional hardware, and Cell programs are said to benefit even more, especially if large structures are involved [18, 46, 6]. Unfortunately, this issue could not be addressed in course of this thesis yet and is thus to be left as future work (cf. Chapter 9).

4.1.4 Cell Broadband Engine Specific Characteristics

Linux sees the Cell processor primarily as a dual core PowerPC 64 architecture, which comes from the two independent PPU pipelines being interpreted as dedicated cores (cf. Paragraph 3.4.1). The SPUs are in contrast not natively accessed by the kernel, but are instead mounted as a virtual file system called *`spufs`* [8]. This system is very comparable to for instance the `procfs` tree present in most Linux distributions, and similar to this typically being mounted on `/proc`, `spufs` is by convention mounted on `/spu`.

As soon as a SPU thread is being started, a new folder corresponding to this SPU kernel is created inside this file system, and this folder then contains six devices: `ibox`, `mbox` and `wbox` are the interrupt and standard outbound mailboxes, as well as the inbound mailbox, respectively, seen from PPU side. `mem` describes the SPU's Local Store, which can be memory mapped to issue DMA transfers in form of copy instructions, and `regs` maps to the status, channel registers and problem state registers. `run` is finally needed as a control device to execute a SPU kernel.

Upon this structure, SPU kernels can in principle be run without the need of any external library. By generation of a new folder inside the `/spu` tree, all devices inside this folder are automatically generated, and after transferring the SPU kernel binary into the Local Store, the program can be executed by a single write operation to `run`. For programs involving communication via mailboxes, the program flow can then be controlled by reads and writes to the mailbox and register files. By deletion of the folder, the kernels are disposed again.

Fortunately, these interactions are encapsulated in the `libspe2` C library, which provides high-level instructions and meaningful aliases for common bit patterns to be passed as arguments to low level functions. Using this library, the programmer usually does not explicitly see the underlying low-level structures. However, since SPU kernels are sometimes run independently from any PPU task, where they can for instance fill the role of a sound processor or a stream data processor in a streaming model following the nomenclature of Kahle *et al.* [40], they need from time to time to be executed immediately by the operating system. For these stand-alone kernels, a deep knowledge of the internal processes and register configurations is inevitable. Anyway, because the architecture is fairly well documented, this is no problem at all [8, 31, 33, 35].

4.2 Compiler and Toolchain

IBM offers a software development kit (SDK) for download, containing a compilation framework, the `libspe2` library, documentation, examples and two different C and C++ compilers width their toolchains [36]. One option provided are the GNU C compiler and GNU C++ compiler collections, which are also used in context of this thesis. Established on traditional architectures, this program suite has been slightly adapted to the PPU, and is extended by new compilers for the SPU instruction set, called `spu-gcc` and `spu-g++`. Furthermore, it is supplemented by utilities like `ppu-embedspu`, which embeds SPU object modules into PPU binaries. The second option are C and C++ compilers developed by IBM called XL C and XL C++, respectively. They are less common on the desktop market, but are quite popular in the supercomputing area. Within the SDK's automated installation script, any of both options can be selected, which causes the default compiler and toolchain for all programs to be automatically set.

Both toolchains are also available as cross-compilers for x86 based host systems, which allows developers to write and compile their code entirely on an external system, and to use the actual Cell Broadband Engine chip only when the program is finally to be run in a time critical setting.

Within this framework, predefined makefile footers for both SPU and PPU side are provided to compile software projects. In a makefile for a PPU program with embedded SPU code, only the PPU programs to be compiled, custom compiler options and an identifier for the corresponding SPU kernels need to be specified, and the actual compiler calls are then issued inside the footer file included by the last line of the makefile. Similarly, SPU programs only require a list of affected SPU kernels, optional compiler flags and the identifier for the corresponding PPU kernel. The actual calls to the compilers and compilation scripts are then hidden from the programmer as well.

During such compile process, each single SPU kernel is first compiled for itself. The resulting object modules are then embedded into a PPU library with the name of the identifier chosen in both the SPU and PPU makefiles. During the compilation of the PPU program, this library is just included and the single SPU kernels are then dynamically linked to placeholder variables previously declared **`extern`** in the main program.

4.3 IBM Full-System Simulator

For debugging and testing purposes, IBM gives developers a simulator for the Cell architecture at hand, which is available for both x86 as well as for ppc based host architectures. It is hence in particular possible to execute this simulator on the PPU of an actual Cell processor, even though the performance achieved is quite low. The underlying simulation model is said to describe exactly the internal processes taking place on the real hardware, such that the behaviours of a software simulated and a hardware driven program can be assumed to be exactly the same [37].

This so-called *Full-System Simulator for the Cell Broadband Engine* allows programmers to test their software on common Intel or AMD desktop or notebook processors, such that the development process on alien host systems can be entirely brought to an conclusion. The programs developed and tested in such setups are immediately runnable on an actual hardware, and this makes this software interesting for setups where software developers have no direct access to the target architectures, like it is the case in supercomputing centres with IBM Cell Blades, where tasks are dispatched from within a load balancing queue.

However, even if a Cell chip is available, the simulator can be very beneficial, since it grants access to both the simulation, as well as to the simulator itself. This means, not only the results of the simulated program can be output as this would be the case on actual hardware implementations, but also internal states of the simulator can be output at any time for analysis purposes. This includes information about bandwidth usage for several buses, timing information, pipeline stalls, detailed register states etc. In the included graphical frontend, most scalar-valued data is displayed in animated bar charts plotting the value at a continuously evolving time, and since the simulation speed can be interactively controlled, many hazards attract attention by visual changes in the graph and can be immediately backtracked and matched to certain code fragments. Several views, like the general PPU-centered perspective, but also SPU-internal views, offer the opportunity to learn more about the internal behaviour of the own program, and even the interactions between the cores and are indeed helpful to optimize the actual SPU kernel to the finish [39].

For instance, data hazards occurring in a loop generate regular patterns in the graph for instruction fetches, and page faults appear as sudden collapses of load and store bandwidths. At least for concise programs, these observations immediately lead to concrete timing or scheduling problems occurring at runtime, and those can in most cases then be counteracted by little changes in the program flow, or by introduction of algorithmic optimizations like loop unrolling techniques.

The simulator supports two different run time modes. On the one hand, it can natively execute standalone SPU kernels, since there is no operating system kernel running on any SPE that those programs would need to interact with. Those SPU binaries can be assumed to run on a plain core simulator without any additional environment restrictions.

The second mode, which has exclusively been used in context of this thesis, is however more sophisticated. PPU programs are not possessing immediate access to the PPU hardware, but they are running on top of and in interaction with a running operating system kernel. This implies several intricacies, like virtual memory and page translation, anomalies in reading and writing bus operations as they are present for the communication between cores, system library calls, or multitasking. Under these circumstances, it is inevitable to run a concrete Linux kernel within this setup, and since analysis output should still be available, it is insufficient to use the kernel of the host system. Instead, the entire operating system must be simulated in software, which is in particular time consuming if the simulator is running on a PPU host system, since the performance of the PPU is only about comparable to an Athlon CPU with 1.33 GHz [60].

For these purposes, IBM ships a small Fedora Linux kernel with a minimal configuration, which is executed in the simulator. When this virtual system has booted, commands can be entered at a shell prompt. For the simulation to take place in a deterministic environment, actual devices like the host system hard disk are natively inaccessible from within the simulation, and the simulated user home directory is initially empty. To transfer programs into the virtual environment, a special **`callthru`** command can be used to copy data from the host harddisk to the simulated system. Unfortunately, all contents transferred are session-specific and are hence lost when the simulation ends.

This fact can be tedious when rapidly changing binaries or testing data needs to be synchronized. In these situations, it has turned out to be helpful to have a shell script on the host system, which is first manually transferred into the simulator at the beginning of each session, and which can then be used to issue all further **`callthru`** commands needed for a certain experiment.

4.4 Debugging

Working on the Playstation 3, one faces the problem of a proper framework to debug programs. Besides simple tracing of bugs occurring at some point in the program flow, one also often wants to inspect or modify the values of variables involved.

Obviously, single-threaded and sequential attempts are less likely to work, as the architecture differs significantly to usual x86 based computers. Besides the multi-core structure implying debugging of remote processors, i.e. such not running the operating system, the different instruction sets also require a suitable debugger to be adaptive to heterogeneous binary formats meshing with each other in alternation.

To enable debugging symbols within the executables, they need to be compiled adding the **`-g`** flag to the GNU C Compiler (**`gcc`**) command line.

4.4.1 GDB

In their Cell Software Development Kit (SDK), IBM includes a version of the GNU Debugger (**`gdb`**) which has been specially adapted to the Cell platform. GDB comes in two binaries, namely **`ppu-gdb`** and **`spu-gdb`**.

Latter expects a SPU binary to process and is hence incompatible to most parallel programs on Cell, which typically consist of PPU code having SPU binary code embedded (cf. Section 3.9). Instead, it has been designed to be used for the development of dedicated SPU binaries such as Digital Signal Processor (DSP) firmware, or other programs working without immediate interaction with the PPU. Kahle *et al.* refer to this programming pattern as the *Device extension model* [40].

In contrast, **`ppu-gdb`** constitutes a combination of the usual PowerPC **`gdb`**, with which it must not be confused, and **`spu-gdb`**. Being executed upon a PPU application with embedded SPU code, it separates the respective parts from each other and then creates a transparency layer including SPU programs as generic POSIX threads.

From an external view, **`ppu-gdb`** looks very similar to the GNU Debugger on x86 based systems. The software is executed by

```
$ ppu-gdb <binary file>,
```

and soon prompts for commands:

```
(gdb) ▮
```

which can be answered by several instructions to set or manage breakpoints, or to control the program flow, such as

```
(gdb) run <arguments>
```

to finally dispatch the program. This thesis does not go into details of the various possibilities **`gdb`** offers, because the webpage of the GNU Debugger proposes a comprehensive documentation [24]. Instead, the main differences to generic single-threaded x86 architecture based debugging are going to be briefly sketched.

At this stage it is already possible to set breakpoints within files belonging to an embedded binary, namely one later to be dispatched on a SPU. Even though this file is not linked to the PPU executable in any way itself, **`ppu-gdb`** offers to set this breakpoint as soon as this file is finally loaded. This allows to set dynamical breakpoints even in those sections of the program that are probably not loaded at the beginning and thus avoids intricate manual determination of the respective point in time the corresponding library is finally being loaded.

As mentioned before, SPU programs are visible as POSIX threads within the PPU binary embedding them. To get a list of currently running threads, one can issue the command

```
(gdb) info threads
```

while the program is stalled, which returns a list of the form

```
<Thread number> Thread <Thread ID> (LWP <LWP ID>)
<position>,
```

where the thread currently observed is marked by a leading asterisk (∗). Hereby, `<Thread number>` denotes an identifier to be used for changing the current thread:

```
(gdb) thread <Thread number>,
```

`<Thread ID>` is the internal thread ID assigned by the Linux kernel, and `<LWP ID>` its corresponding lightweight process ID.

Latter comes from the fact that Linux threads are usually not handled as fully-fledged system processes with a dedicated memory range and a restricted number of status information tags for the scheduler, but appear as parts of the main program, thus sharing resources. Even though these so-called *light-weight processes* (LWP) are hence vulnerable to memory leaks, they are indeed immediately scheduled by the kernel, which makes these implementations fast compared to solutions scheduled by high-level software. Because SPU programs are displayed within **`ppu-gdb`** as if they were usual POSIX threads, they are therefore also addressed by a LWP ID.

Finally, `<position>` gives the head of the execution stack the respective thread was stalled at. It typically consists of the source file, a line and the function it is currently in, or a symbol within an external library if it has just been executing any callback function.

As for sequential programs, it is now possible to interrupt execution at any time, and to inspect the stack or involved variables. However, own experiments revealed that this interruption might in some cases be responsible for corruptions of the inter-SPU signaling mechanism, which can cause related errors like deadlocks to occur on program continuation, even though the program works if it is not being interrupted.

Another anomaly becomes obvious when the system fails with an illegal instruction signal (SIGILL) being returned to the Linux kernel. This signal often turns out to be the the outcome of a segmentation fault (SIGSEGV), i.e. a reference to an invalid memory cell, occurring within an SPE binary. Anyway, since this situation occurs deterministically, it can just be irritating to programmers, but does not have any effect on the debugging process.

More severely, **`ppu-gdb`** can sometimes been observed to return wrong backtraces for failing runs of a program. This problem occurs even for binaries without automatic optimizations performed at compile time, i.e. for those compiled with an **`-O0`** flag set. For instance, an assignment of local variables within an SPU kernel was once marked as a location for a segmentation fault, even though no pointer arithmetics was involved at all. An analysis of the generated assembly code revealed that the respective instruction had been scheduled at some other location in the routine to counteract dependency-related latencies (cf. Paragraph 3.5.1), and to utilize both odd and even pipelines equally. Due to this fact, the reported and the actually failing instruction were hence simultaneously in the two pipelines, even though their high-level representations are several lines of code apart from each other, and **gdb** just seemed to have the wrong pipeline accused for this fault. Without fully understanding the internal processes taking place at this point, there is hence nevertheless evidence that the problem can be traced back to the novel design of the SPU architecture, rather than to bugs in **gdb**.

Another problem caused by the asymmetric multi-core architecture occurs when the programmer handles synchronization messages improperly inside the SPU kernels: From time to time, timing-related errors did not occur inside the debugger, while they in fact did outside. For these purposes, the kernel can be instructed to create a core dump, which is performed by setting the maximal core dump size to infinite: **ulimit -c unlimited**. Whenever the program aborts or runs in segmentation faults without being run in a debugger framework, the operating system kernel creates a core dump file, which can later be inspected by **ppu-gdb**:

```
ppu-gdb --core=<core file> <program>
```

4.4.2 Data Display Debugger (DDD)

The Data Display Debugger (DDD) constitutes a graphical frontend for GDB under the X window environment. Besides full GDB functionality, it furthermore offers ways to display complex data structures in a UML-like representation, to track the program flow in both the assembly and the high level language in parallel, to observe variable contents by clicks on the respective identifier in the source code, and to manage breakpoints interactively.

In order to introduce the special debugger to **ddd**, the frontend needs to be invoked with **ppu-gdb** as debugger argument:

```
$ ddd --debugger ppu-gdb <program binary>
```

This overrides the default setting to use the PowerPC **gdb**.

One major advantage of **ddd** over the command line tool is however that it can display assembly code and the high level language in parallel, such that several errors can be more easily understood on the first view, by referring to the generated low level code.

4.4.3 Other Tools

It unfortunately turns out that the gdb based tools are less usable than their x86 based counterparts. As described above, it is often not possible to make them find the position the program actually fails, because even if the compiler is instructed to use no optimization, i.e. it is given the **-O0** flag, it reorders independent assembly instructions to reduce delays in the processing pipeline. This can for instance cause a call to a system library to be spread over an area of many instructions, being interleaved with many other lines of code, which makes it hard to determine the actual point of failure.

Different tools known from sequential development like Valgrind are not available for this platform at all [63]. This renders it hard to detect memory leaks, array range overflows, or uninitialized variables. Instead, the Cell developers propose the Full-System Simulator as a replacement, which is discussed in the next paragraph.

4.4.4 Full-System Simulator

The Full-System Simulator presented in Section 4.3 also provides many ways to debug Cell programs. On one hand, register contents and the program counter are always available, such that the program can be inspected in detail while it is being run. In particular, the simulator hints programmers to a frequent error source in Cell programs, namely misaligned DMA operations, which occur as a Bus Error (SIGBUS) whenever this program is executed on the real architecture, and returns the wrong alignment requested. Additionally, the program can be stepped instruction-wise as this is known from debuggers like **gdb**, which gives a pretty good method at hand to trace smaller programs. For larger algorithms however, a real debugger is helpful.

Because of this, the simulator provides a **gdb** service interface, which allows **gdb** to attach to a certain port calling

```
(gdb) target remote :<port>
```

from within the debugger. By this technique, at least standalone SPU kernels can be traced from external **gdb**, or even graphical **ddd** sessions.

Even though this system seems quite convenient and is also strongly advertised throughout official manuals published by IBM, several practical problems occurred during the development of software pieces for this thesis. Because it takes several minutes to boot the relatively small Fedora Linux kernel shipped by IBM within the simulation, this method has often turned out to be less effective for debugging than simple approaches on the actual host system. In particular, since the algorithms observed for this thesis are rather complex, it takes a long time until the simulator finally approaches problematic sections. By running the simulator on the PPU of the Playstation 3 Cell processor, this effect is even amplified.

Anyway, the general setup involving a simulated Linux kernel and own programs running as child processes upon this kernel turned out to be quite hard to handle. Since the main process simulated is the Linux kernel, and not the designed program, attempts to attach external debuggers to such child processes have been unsuccessful so far. A second inconvenience is given by the virtual environment of the simulator. Though it indeed makes sense to have a fully isolated testing environment, the tedious file transfer between host system and simulation (cf. Section 4.3) causes long lead times before a simulation can be started, and often does not pay off for the insights such simulation could provide.

Finally, the graphical output of this environment is not transparently handled via the X server of the host system, but an entire X server would need to be simulated whenever an application, like those developed in context of this thesis, needs graphical output. However, since this piece of software is quite complex and would slow down the simulation dramatically, an alternative approach has been pursued when necessary: For debugging purposes, an alternative frontend to the developed application has been designed, which gets by without any graphics libraries and works only based on console interaction. Though it does not visualize algorithmic errors, it could at least be used for several runtime problems like buffer overflows. Unfortunately however, the effort necessary to maintain such branch of the program has in general not proven to be justified.

4.4.5 Summary

So far, several debugging techniques have been described, whereby the focus has been laid on methods applicable to combined binaries for PPU and SPU. Besides these, the official IBM debugging manual presents several techniques rather oriented on standalone SPU kernels, which are on the one hand quite easy to debug, but hard to simulate [42]. Since interactions of those kernels with the PPU side either need to be manually supervised or deactivated during debugging, this method is unfortunately infeasible in many cases tested in this thesis.

The available debugging tools can since still be regarded to be quite restrictive. In fact, the most effective way to debug programs turned out to be given by rather unsophisticated methods, like textual debugging output from within the program, simplification of the problem by selective deactivation of potentially failing code partitions, or sometimes even analysis of the generated assembly code. All these techniques have in common that they often do not call a defect by its name, but only hint to a logical subset of the complex problem as the source for the error. Since these hints represent however the most reliable information available in this context, they have indeed by worthwhile at many points. Anyway, to confirm suspicions established based on these observations, **gdb** has turned out to be very powerful again, since it allows to set watchpoints on changes of variables.

The most sophisticated and architecture-focused tool, the Full-System Simulator, unfortunately turned out to be less effective in application, especially with respect to smaller errors, since the initialization time is just too long and the results for distributed applications are also often not too convincing when run in a distributed setting under an emulated Linux kernel.

This way, debugging still causes a high amount of manual work and errors are sometimes hard to locate. Simple issues like buffer overflows or underflows, which are nowadays automatically detected on traditional hardware by using tools like Valgrind, still need to be found by hand. For complex algorithms, this fact turns out to slow down the development process noticeably, since the defect and the actual failure in terms of the taxonomy proposed by Zeller [74] are in most cases not joint at one single point in the code.

In addition, there is no automatic or half-automatic tool available to track synchronization problems among SPUs, since they are of logical type and at best express in secondary errors. These defects can typically only be found by a simplification of the program, i.e. by introduction of a special treatment for all involved multithreading participants: By delaying a certain SPU, or by manually scheduling it before some other one, bugs can be made explicit and deterministic, and thus provide hints to the actual source of the error.

Chapter 5

Optical Flow

5.1 Motivation

In the field of Computer Vision, motion detection and classification represents a central problem in the understanding of the environment. Because data captured by various sensors typically only describes projections of the reality rather than providing a dense field of information, reliant methods are needed to estimate the information from the raw data acquired. Based on the observation that the human visual system is capable to solve very similar problems in many situations of the everyday life, video cameras are a popular sensor used.

Motion within subsequent frames of an image sequence can be displayed as a dense vector field describing the movements of certain illumination patterns relative to the image plane, which corresponds to a projection of displacements performed by objects captured. Those vector fields are called *Optical Flow Fields*, or *Optical Flow*.

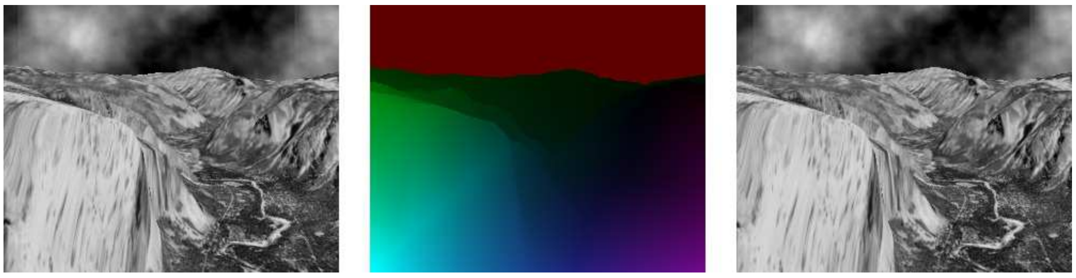

Figure 5.1: Left and **right**: Frames 8 and 9 of the Yosemite sequence created by Lynn Quam. **Middle**: Flow field ground truth in color coding.

Figure 5.1 visualizes such flow field between two frames of a well-known testing data set, namely the so-called *Yosemite sequence*. This artificial image sequence sized 316×252 pixels displays a camera flight trough a textured 3-D model of the Yosemite valley and has originally been created by Lynn Quam [9]. On the left and on the right, frames 8 and 9 of the sequence are depicted, and in between, the flow field is visualized in a color coding used by Bruhn in context of his PhD thesis [15]. The direction is thereby expressed in terms of a shade on the color circle, with red pointing to the right, yellow to the top, green to the left, and blue to the bottom (cf. Figure 5.2). The intensity of a certain point characterizes the length of the displacement vector, normalized to the maximal movement encountered.

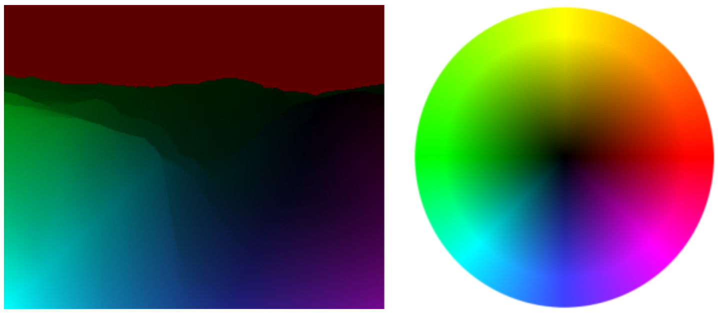

Figure 5.2: Color coding used for the visualization of optical flow fields. **Left**: Flow field ground truth of the Yosemite sequence, between frames 8 and 9. **Right**: Color circle depicting the assignment of colors and intensities to directions and normalized lengths of the vectors in the respective points. **Author:** A. Bruhn [15].

It turns out that man-made models estimating the Optical Flow are in general more restrictive in comprehension than human beings are, which comes from the fact that they lack background knowledge about underlying structural information. Instead, they are bound to different illumination intensities measured in every pixel of the input image, and the relative change thereof. Latter fact makes them dependent from perturbations in illumination, which are either related to actual changes in the lighting of a scene, or to noise that comes from different numbers of photons hitting the respective part of the image sensor.

5.2 Horn and Schunck Model

The model proposed by Horn and Schunck [27] provides a global differential method to estimate the flow field as the minimizer of an energy functional.

Let $f(x, y, t) \in \Omega \times \mathbb{R}^+$ be a grey valued image sequence, whereby (x, y) describes a point in the Cartesian image domain Ω, and $t \in \mathbb{R}^+$ is the evolution of time. Because images are typically subject to noise, more stable and reliable results can be achieved by extending the support of the model to a small neighborhood of the respective pixels. This is done by a convolution of the input images with a spatial Gaussian kernel K_σ of variance σ:

$$g(x, y, t) = K_\sigma * f(x, y, t).$$

In these smoothed versions, corresponding points in both frames have to be found to estimate the motion performed in the regarded timestep. One very simple assumption to identify such pairs of pixels in two successing frames is to assume them to have the same grey value, i.e.

$$g(x + u, y + v, t + 1) = g(x, y, t),$$

or

$$g(x+u, y+v, t+1) - g(x, y, t) = 0,$$

where $(u(x,y,t), v(x,y,t))^T$ denotes the displacement vector at the location (x,y,t). Assuming $(u,v)^T$ to be small and g to be sufficiently smooth, a first order Taylor expansion of g around the point (x,y,t) yields

$$g_x u + g_y v + g_t \approx 0,$$

with $g_n(x,y,t) := \frac{\partial}{\partial n} g(x,y,t)$ being a partial derivative with respect to a variable n. This approximation leads to the linearized *Optical Flow Contraint*, which is given by setting the term to zero:

$$g_x u + g_y v + g_t = 0.$$

In terms of the Horn and Schunck model, this constraint is incorporated by means of a least squares fit. This can be written as a minimization of the energy functional

$$E_D(u,v) = \int_\Omega (g_x u + g_y v + g_t)^2 \, dx \, dy.$$

The integrand of this functional is referred to as the *data term*.

Unfortunately, this equation system is underconstrained, since two unknowns are attempted to be solved with only one equation: Only the *normal flow* parallel to the spatial image gradient $\nabla g = (g_x, g_y)^T$ can be estimated, while the displacement in perpendicular direction can take any value. As a result, the proposed solution does not necessarily need to agree with the ground truth, but can deviate significantly without violating the constancy assumption, which can deteriorate the accuracy of the flow field significantly.

This issue is known as the *aperture problem*, and many alternative data terms have been suggested in the past to reduce the impact thereof. For instance, assuming constancy of the spatial image gradient as proposed by Uras *et al.* additional orientation information is introduced into the model [62]. This has proved to yield better results for linear motion, like this is the case for translations. On the other hand however, this approach seems to be less suited for rotational motion, since such movements involve orientation changes which are not covered at all. Other data term models use different constancy assumptions involving analysis of the Hessian, the Laplacian or the gradient magnitude [66].

Another drawback being independent from the chosen data term has not been addressed so far: As soon as the gradient ∇g vanishes, i.e. $|\nabla g| \approx 0$, E_D drops towards zero, which leaves little impact on the flow field $(u,v)^T$. To overcome this problem, an additional assumption is necessary. Observing the ideal solution for the flow field to acquire, one finds that it is largely smooth, except for the boundaries of objects moving relatively to the background. This motivates to constrain the smoothness of the solution as well, which is again formulated as least square fits with respect to the first derivatives of the flow field components:

$$E_S(u,v) = \int_\Omega \left(|\nabla u|^2 + |\nabla v|^2\right) dx \, dy.$$

In contrast to the data term, this integrand is called the *smoothness term*.

E_D and E_S are now combined to a joint energy functional, whereby the smoothness term is weighted by the regularization parameter α:

$$E(u,v) = \int_\Omega \left((g_x u + g_y v + g_t)^2 + \alpha \left(|\nabla u|^2 + |\nabla v|^2 \right) \right) dx\, dy.$$

By 2004, Brox *et al.* supposed a new variational approach combining both brightness and gradient constancy assumptions with a spatiotemporal smoothness constraint and non-quadratic penalizers [13]. To be capable of large displacements, this model does not involve linearizations by design. This is particularly interesting, because cameras often suffer from undersampling, this means the frequency single images are captured at is too low to have a smooth evolution in time. However, if coarse-to-fine strategies are used, it can indeed by worthwhile to simplify the process by linearizations on coarser scales, where all significant displacements can considered to be small. So far, this method is among the best with respect to the *average angular error*, a global error measure judging deviations from the ideal solution which is nowadays a common comparison among different methods.

In the context of this thesis, only the simplest approach using brightness constancy is covered. It turns out that this model is quite imprecise compared to others, but its fairly manageable complexity allows for focussing on algorithmic adaptations to the new hardware setting. With the insights gained by this method and the routines developed, an extension of the program towards a more sophisticated model should in principle not require too many new architecture specific optimizations, but most operations can certainly resort to modules already developed.

5.3 Motion Tensor Notation

In his PhD thesis, Bruhn proposed an alternative way of writing the data term [15]: Let $w = (u, v, 1)^T$. The data term of the Horn and Schunck model under brightness constancy assumption can hence be written as

$$\begin{aligned} (g_x u + g_y v + g_t)^2 &= \left(w^T (g_x, g_y, g_t)^T \right)^2 \\ &= \left(w^T \nabla_3 g \right)^2 \\ &= w^T \nabla_3 g \, \nabla_3 g^T w \\ &=: w^T J(\nabla_3 g) w. \end{aligned}$$

Using this notation, the energy functional for the Horn and Schunck model then reads:

$$E(u,v) = \int_\Omega \left(w^T J(\nabla_3 g) w + \alpha \left(|\nabla u|^2 + |\nabla v|^2 \right) \right) dx\, dy.$$

Because the tensor $J(\nabla_3 g)$ uniquely describes all constancy assumptions made in the model, which are only represented by the brightness constancy assumption in this particular case, it is referred to as the *motion tensor*. Whenever the underlying assumption set is changed, it is sufficient to merely replace the motion tensor by its new representation, which preserves the flexibility of the general notation described above. As one can see from its definition, the motion tensor only depends on the image data. It hence offers the opportunity to be precomputed as soon as the image sequence is available, and does not need to be modified in course of an iterative solver for the resulting linear system of equations. This observation renders the technique interesting towards an efficient implementation on computer systems.

Bruhn showed that arbitrary assumptions or combinations thereof can be written in this particular notation, and that the matrix yielded is always symmetric positive semidefinite [15].

5.4 Combined Local-Global Method

Extending the model of the previous section with ideas proposed by Lucas and Kanade [47], Bruhn *et al.* suggested to assume that the flow field to be estimated is smooth within a neighborhood of size ρ [17]. All pixels within such neighborhood hence contribute to similar flow field characteristics. In this case it is feasible to include this neighborhood into the computation of the least squares fit, and to weight the single contributions by the distance to the location being estimated.

In terms of the Horn and Schunck model, this amounts to a convolution of the motion tensor with a Gaussian kernel K_ρ of variance ρ:

$$E_\rho(u,v) = \int_\Omega w^T \left((K_\rho * J(\nabla_3\, g))\, w + \alpha \left(|\nabla u|^2 + |\nabla v|^2 \right) \right) dx\, dy.$$

The authors showed that results yielded by this technique are in general more stable under Gaussian noise than the unmodified Horn/Schunck model is.

5.5 Minimization

Minimizing this energy functional comes down to solving its Euler-Lagrange equations [21]. In terms of the motion tensor notation, they are given by

$$\begin{aligned} 0 &= \Delta u - \frac{1}{\alpha} \left(K_\rho * J_{11}\, u + K_\rho * J_{12}\, v + K_\rho * J_{13} \right) \quad \text{and} \\ 0 &= \Delta v - \frac{1}{\alpha} \left(K_\rho * J_{12}\, u + K_\rho * J_{22}\, v + K_\rho * J_{23} \right). \end{aligned}$$

Like the first derivative of ordinary functions must necessarily vanish at minima, these equations enforce the first variation of $E_\rho(u,v)$ to vanish. Additionally, reflecting Neumann boundary conditions need to hold for both u and v:

$$\begin{aligned} 0 &= n^T \nabla u \quad \text{and} \\ 0 &= n^T \nabla v, \end{aligned}$$

where n denotes the normal vector perpendicular to the image boundary.

5.6 Discretization

The parallel implementation on the Cell Broadband Engine developed in this thesis closely follows a sequential program from Bruhn and Weickert and which is used in the Correspondence Problems in Computer Vision lecture at Saarland University [16]. This program written for a usual x86 architecture is hence a good comparison for the overall running time with respect to different configurations and settings, and provides a measure for the speedup achieved by applying the parallel processor. The discretization process has therefore been kept similar to this implementation.

In order to solve the Euler-Lagrange equations numerically, they need to be discretized with respect to a rectangular grid with spacing h_x and h_y. In principle, this leaves options for different discretization attempts on u and v, on the Motion Tensor entries $J_{11}, \ldots, J_{33}$, and on the Laplacians Δu and Δv. The flow field components u and v are sampled according to the grid spacing:

$$\begin{aligned} u_{i,j} &= u\left(h_x(i-1), h_y(j-1)\right) \quad \text{and} \\ v_{i,j} &= v\left(h_x(i-1), h_y(j-1)\right), \end{aligned}$$

with $i \in \{1, \ldots, M\}$ and $j \in \{1, \ldots, N\}$. This is feasible, because the regarded model only involves spatial approaches, but no spatiotemporal attempts. In the latter case, one would need to consider temporal discretizations as well.

As shown above, the motion tensor is composed from products of derivatives in each entry. Hence, it can be discretized as a product of discretized derivatives, which defers the problem to finding suitable discrete representations of first order derivatives. All programs developed in course of this thesis, as well as the sequential implementation, use a finite difference scheme involving central differences with fourth-order error approximating the spatial derivatives at the respective points. This scheme results from a Taylor Expansion around these locations in the piecewise linear interpolation

$$\tilde{g}(x,y,t) = \frac{g(x,y,t) + g(x,y,t+1)}{2}$$

between to subsequent frames of the presmoothed image sequence, and can be written as

$$\begin{aligned} g_x(i,j,t) &\approx \frac{\tilde{g}(i-2h_x,j,t) - 8\tilde{g}(i-h_x,j,t) + 8\tilde{g}(i+h_x,j,t) - \tilde{g}(i+2h_x,j,t)}{12h_x} \\ g_y(i,j,t) &\approx \frac{\tilde{g}(i,j-2h_y,t) - 8\tilde{g}(i,j-h_y,t) + 8\tilde{g}(i,j+h_y,t) - \tilde{g}(i,j+2h_y,t)}{12h_y}. \end{aligned}$$

In contrast, the temporal derivative is computed by a forward difference between two frames:

$$g_t(i,j,t) \approx \frac{g(i,j,t+1) - g(i,j,t)}{2}.$$

Inserting these discretized derivatives into the respective entries of the motion tensor yields the discretized motion tensor.

In a last step, the Laplacians of u and v need to be discretized. Writing $\Delta u = (u_x)_x + (u_y)_y$ and using central differences on halved grid sizes $\frac{h_x}{2}, \frac{h_y}{2}$ for both the inner and the outer derivatives, the Laplacians can be approximated by

$$\Delta u \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} \quad \text{and}$$

$$\Delta v \approx \frac{v_{i-1,j} - 2v_{i,j} + v_{i+1,j}}{h_x^2} + \frac{v_{i,j-1} - 2v_{i,j} + v_{i,j+1}}{h_y^2}.$$

Eventually, these discretizations can now be used to formulate the discretized Euler-Lagrange equations as

$$0 = [J_{11}]_{i,j} u_{i,j} + [J_{12}]_{i,j} v_{i,j} + [J_{13}]_{i,j} - \alpha \sum_{l \in \{x,y\}} \sum_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l(i,j)} \frac{u_{\tilde{i},\tilde{j}} - u_{i,j}}{h_l^2} \quad \text{and}$$

$$0 = [J_{12}]_{i,j} u_{i,j} + [J_{22}]_{i,j} v_{i,j} + [J_{23}]_{i,j} - \alpha \sum_{l \in \{x,y\}} \sum_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l(i,j)} \frac{v_{\tilde{i},\tilde{j}} - v_{i,j}}{h_l^2},$$

where $[J_n]_{i,j}$ describes the discretized motion tensor entry J_n at grid location (i, j), and $\mathcal{N}_l(i, j)$ represents the four-neighborhood of pixel (i, j). Note that in the latter notation, boundary conditions are already implicitly handled.

5.7 Sequential Implementation

Porting of Bruhn and Weickert's implementation to the PPU can be performed quite easily, because the PPU in fact resembles a multi-purpose architecture. However, anomalies imposed by the processor itself or the Playstation 3 video console must always be considered.

Due to its PowerPC nature, the PPU is designed as a big-endian architecture, unlike x86 processors, which are designed little-endian. This primarily comes into account when file input and output operations are considered and non-native variable widths are used during this process.

The second restriction is related to the special setting within the Sony Playstation 3. As described in Section 4.1, the graphics pipeline is not accessible for the programmer, which immediately results in a lack of 2-D and 3-D hardware acceleration. To visualize the effect of different solvers for the discretized Euler-Lagrange equations, Bruhn and Weickert use an OpenGL frontend drawing flow field representations to the screen, whereby the notation presented in Section 5.1 is used. There are however approaches circumventing this lack of 2-D acceleration by outsourcing computational load to the SPUs, like this is done by the Mesa Cell project [61]. Anyway, because these cores are meant to be used for computations for the optical flow problem, the graphics computation has been decided to remain on the PPU. Hence, an SDL frontend involving simple 2-D drawing operations has been developed to replace the OpenGL frontend.

With these minor changes, the program can be run on the PPU. In the following chapters, reference benchmarks for both Pentium 4 and PPU platform are presented (cf. Sections 6.2, 6.3, 7.2, 7.3).

Chapter 6

Successive Over-Relaxation

6.1 Theory

In the previous chapter, it turned out that the optical flow problem actually comes down to solving a large linear system of equations of the form

$$A\,x = b,$$

where A is the regular positive semidefinite and symmetric coefficient matrix sized $n \times n$, and x, b are the variable and constant right hand size vectors, respectively, with n elements, each.

6.1.1 Gaussian Elimination

One straightforward way to solve such system is given by the Gaussian elimination method usually known from undergraduate courses. Here, each row is used to eliminate exactly one unknown from all other rows, and this step is repeated until each row only contains at most one unknown. From this configuration, the solution can then be immediately read off.

This method is exact within the numerical precision of the computation, but it also reveals one major drawback. Because the matrix is processed n times, the overall algorithm has a runtime complexity of $\mathcal{O}\left(n^3\right)$ – which is typically infeasible for sufficiently large systems of equations.

6.1.2 Iterative Solvers

A good compromise is offered by so-called iterative solvers. Instead of directly aiming at a solution, they are initialized with some arbitrary value which is meant to be as close at the desired solution as it can be predicted at compile time. In several iteration steps, this intermediate result is then successively refined towards the correct solution, until the values achieved are sufficiently precise. These stopping criteria can either be constraint by a fixed number of steps based on experience, or depend on certain conditions evaluated during the iteration. As a termination condition, the value of the residual $r = Ax - b$ is often consulted, since it describes a simple confidence measure of the intermediate solution.

Because both coefficient matrix A and right hand side vector b remain constant in this case, the linear system can now be written as

$$A\,x^k = b,$$

where k describes the iteration step. x^0 can for instance be set to 0.

It turns out that even though results seldomly match the solution, such algorithms are indeed worthwhile for many fields of application, including the optical flow problem. In the following, three iterative methods are presented.

6.1.3 Jacobi Method

Are the entries on the main diagonal of A different to 0, the ith row can immediately be solved for x_i, which yields for all $k > 0$ the *Jacobi method* [12]

$$x_i^k = \frac{b_i}{a_{i,i}} - \sum_{\substack{j=1 \\ (j \neq i)}}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^{k-1}.$$

It can be shown that this method initialized with an arbitrary vector x^0 converges against the correct solution, as soon as the row sum criterion or the column sum criterion are fulfilled.

6.1.4 The Gauss-Seidel Method

When having computed one element of the vector x^k, it is obvious to already use it for the computation of the next element. Indeed, applying the iteration rule

$$x_i^k = \frac{b_i}{a_{i,i}} - \sum_{j=1}^{i-1} \frac{a_{i,j}}{a_{i,i}} x_j^k - \sum_{j=i+1}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^{k-1}.$$

also converges against the solution, and yields in general faster satisfying results than the Jacobi method does. It is called the *Gauss-Seidel method.*

6.1.5 Successive Over-Relaxation

The Gauss-Seidel method can be written in a slightly different form, namely by explicitly expressing the changes each iteration step applies [73]:

$$x_i^k = x_i^{k-1} + \left(\frac{b_i}{a_{i,i}} - \sum_{j=1}^{i-1} \frac{a_{i,j}}{a_{i,i}} x_j^k - \sum_{j=i}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^{k-1} \right).$$

It turns out that the step width can not only be damped while perpetuating convergence, but can even be amplified. For relaxation parameters $0 < \omega < 2$,

$$x_i^k = x_i^{k-1} + \omega \cdot \left(\frac{b_i}{a_{i,i}} - \sum_{j=1}^{i-1} \frac{a_{i,j}}{a_{i,i}} x_j^k - \sum_{j=i}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^{k-1} \right)$$

can be proved to converge. For $\omega > 1$, this method is called the *Successive Over-Relaxation*, short *SOR*. Especially when choosing ω close to 2, this process converges significantly faster than a pure Gauss-Seidel setting. In the following, different parallel approaches based on this SOR technique are presented.

Rewriting the Euler-Lagrange equations developed in Section 5.6 in terms of the SOR solver, the following iteration rule is yielded:

$$u_{i,j}^{k+1} = (1-\omega)\, u_{i,j}^k + \omega \frac{[J_{13}]_{i,j} + [J_{12}]_{i,j}\, v_{i,j}^k - \alpha \left(\sum\limits_{l \in \{x,y\}} \sum\limits_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l^-(i,j)} \frac{u_{\tilde{i},\tilde{j}}^{k+1}}{h_l^2} + \sum\limits_{l \in \{x,y\}} \sum\limits_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l^+(i,j)} \frac{u_{\tilde{i},\tilde{j}}^{k}}{h_l^2} \right)}{-[J_{11}]_{i,j} - \alpha \sum\limits_{l \in \{x,y\}} \sum\limits_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l(i,j)} \frac{1}{h_l^2}}$$

and

$$v_{i,j}^{k+1} = (1-\omega)\, v_{i,j}^k + \omega \frac{[J_{23}]_{i,j} + [J_{12}]_{i,j}\, u_{i,j}^{k+1} - \alpha \left(\sum\limits_{l \in \{x,y\}} \sum\limits_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l^-(i,j)} \frac{v_{\tilde{i},\tilde{j}}^{k+1}}{h_l^2} + \sum\limits_{l \in \{x,y\}} \sum\limits_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l^+(i,j)} \frac{v_{\tilde{i},\tilde{j}}^{k}}{h_l^2} \right)}{-[J_{22}]_{i,j} - \alpha \sum\limits_{l \in \{x,y\}} \sum\limits_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l(i,j)} \frac{1}{h_l^2}}.$$

$\mathcal{N}_l$ again denotes the two-neighborhood of the point i, j alongside l, and $\mathcal{N}_l^+, \mathcal{N}_l^-$ represent the right and bottom, as well as the top and left neighbors, respectively. Note that this discretization thereby already includes handling of the boundaries.

6.2 Reference Implementation on Pentium 4

In order to appraise the achieved performance gain on the Cell Broadband Engine with respect to usual PC platforms, the following reference benchmarks have been made with Bruhn and Weickerts reference implementation, using the same measurement techniques as they are applied to novel programs later in this thesis (cf. Paragraphs 6.4.3, 6.5.8, and 6.6.3). Reference machine is a Pentium 4 processor also clocked with 3.2 GHz and executing the program on one single core.

Benchmarking on the Cell Broadband Engine, one eventually faces the problem of accurate time measurement. On the PPU side, traditional measurement of CPU clocks needed is usually a good approach, and on SPU side, a static code analyzer called *SPU Timing Tool* can be run to make pipeline-related statements about execution times for the SPU kernels. However according to IBM, “static analysis outputs typically do not provide numerical performance information about program execution. Thus, it cannot report anything definitive about cycle counts, branches taken or not taken, branches hinted or not hinted, DMA transfers, and so forth." [30]

Processor clock cycles are nevertheless problematic, even if they were easy to acquire: Since inter-core communication depends on the relative status of the SPU kernels and the PPU program with respect to each other, memory or synchronization latencies can hardly be traced by these methods. Following these considerations, the C `gettimeofday` routine compliant to POSIX.1-2001 is used instead for a pure PPU-centered timing via system time measurements. The returned time in seconds and microseconds is however subject to interferences with the operating system or background tasks. To compensate for these variations, each measurement performed is being taken ten times, and the arithmetic mean of the single outcomes is used as an appropriate result.

Table 6.1: Performance (FPS) of 100 SOR iteration steps on the Pentium 4 3.2 GHz over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\to \infty$	**25**	**50**	$\to \infty$
79 $\times$ **63**	76.56	76.80	77.03	72.03	72.44	72.86
158 $\times$ **126**	19.30	19.15	19.00	17.74	17.64	17.55
316 $\times$ **252**	4.70	4.68	4.65	4.13	4.12	4.11
632 $\times$ **504**	1.15	1.15	1.14	0.94	0.94	0.93

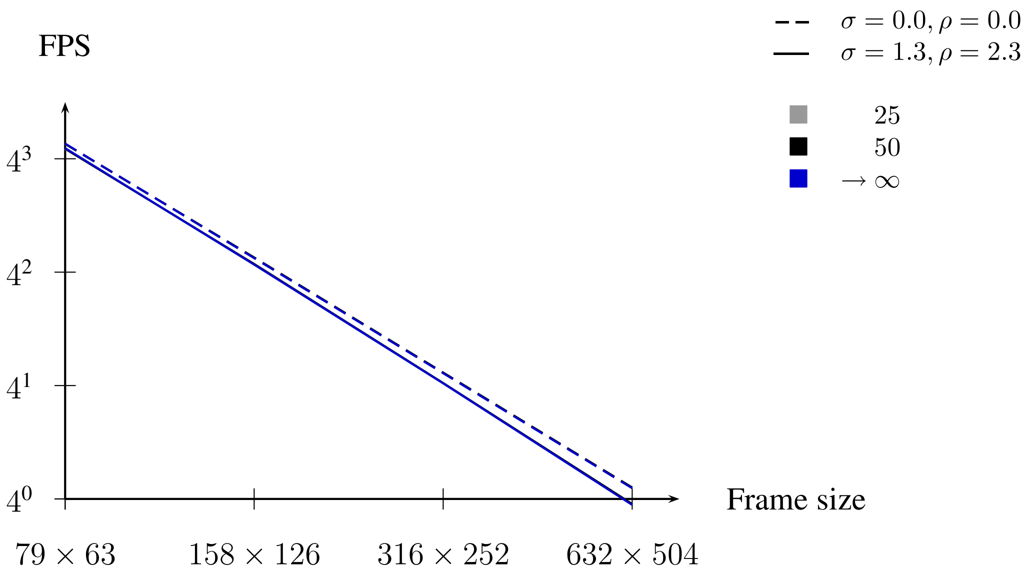

Figure 6.1: Performance (FPS) of 100 SOR iteration steps on the Pentium 4 3.2 GHz over different image sizes. Note that the different test intervals yield almost identical results, and their graphs are thus overlapping.

6.2.1 Impact of Different Image Sizes

In this experiment, frames 8 and 9 of the Yosemite sequence (cf. Figure 5.1) have been rescaled by factors of 0.25, 0.5, and 2 with respect to the edge lengths. The rescaled images hence contain 0.0625, 0.025, and 4 times as much pixels as the original version.

Without loss of generality, runtime benchmarks are throughout this thesis only going to be evaluated against this one pair of images. This is feasible, since no new model is proposed, but all improvements are of pure algorithmic nature, and different runtimes of the algorithms are independent from the actual contents of the problem processed.

This setting will later be interesting to see whether a parallel program distributing load to the SPUs will show preferences towards particularly large or small images. Optionally, the initial image and the motion tensor have been blurred by convolutions with Gaussian kernels. The variances of these kernels have been set to $\sigma = 1.3$ and $\rho = 2.3$ for the input frames and motion tensor, respectively, regarding the original image size of 316×252 pixels. For the other sizes, the variances have been scaled according to the edge length ratio of the respective variant to the original image. This means, the pair of variances (σ, ρ) in the 79×63, 158×126, and 632×504 pixels large setting has been chosen as $(0.325, 0.575)$, $(0.65, 1.15)$, and $(2.6, 4.6)$, respectively.

To detect one-off expenses, all tests have been made over intervals of 25 and 50 entire runs including equation system setup and optionally also presmoothing, and finally the difference is regarded as an extrapolation towards infinitely large measurement intervals. From this point on, latter interpolation will be denoted by $\to \infty$. Like mentioned before, every value refers to an arithmetic mean of ten single measurements.

Throughout this thesis, graphs referring to Bruhn and Weickert's Pentium 4 reference implementation are printed in a shade of blue.

Figure 6.1 and Table 6.1 depict the results of these measurements. Note that the FPS axis is scaled logarithmically, such that the ratio of runtime and pixel number remains easily comparable.

Noticeably, the test interval length has negligible effect to the general behaviour, which can be explained by the relatively fast memory allocation on sequential hardware and by the lack of any special initialization routines. Iterating over a certain code block just means to encapsulate it into some loop construct, and there are typically no further adjustments necessary. In context of a parallel architecture, this assumption can not always be made without ensuring that certain constraints are fulfilled, as it is shown later in this chapter, and in Chapter7.

One also observes an almost perfect linear scaling between the runtime and the number of pixels processed. It will later be shown that this behaviour is not self-evident in a Cell implementation as well.

Table 6.2: Performance of different SOR iteration steps on the Pentium 4 3.2 GHz over images of size 316×252 pixels. Columns denote test intervals averaged over. Rows display the time per frame, the time per iterations, the frames per second (FPS), and the iterations per second (IPS), respectively, for 1, 100 and 200 iterations, each.

		No Smoothing			**Smoothing**		
	it	**25**	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
	1	11.04	11.99	12.95	39.59	40.70	41.80
t [ms]	100	212.70	213.76	214.82	241.93	242.63	243.33
	200	418.38	419.10	419.83	447.45	448.20	448.94
	1	11.04	11.99	12.95	39.59	40.70	41.80
$^t/_{\mathrm{it}}$ [ms]	100	2.13	2.14	2.15	2.42	2.43	2.43
	200	2.09	2.10	2.10	2.24	2.24	2.24
	1	90.61	83.37	77.20	25.26	24.57	23.92
FPS [s^{-1}]	100	4.70	4.68	4.65	4.13	4.12	4.11
	200	2.39	2.39	2.38	2.23	2.23	2.23
	1	90.61	83.37	77.20	25.26	24.57	23.92
IPS [s^{-1}]	100	470.15	467.81	465.50	413.35	412.15	410.96
	200	478.03	477.21	476.39	446.98	446.23	445.49

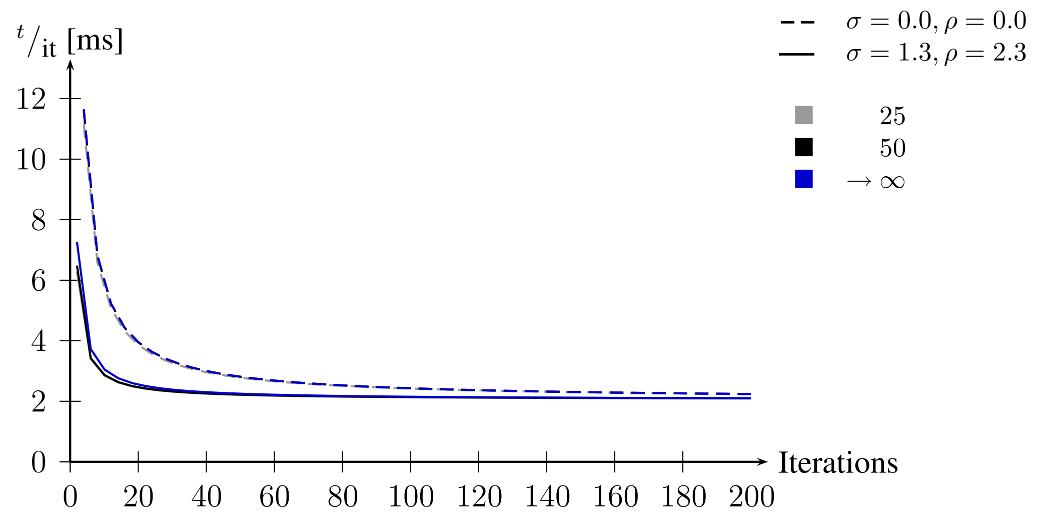

Figure 6.2: Time per iteration step on the Pentium 4 3.2 GHz over images of size 316×252 pixels. Again, the length of the test interval has little influence on the result, such that the respective graphs overlap.

6.2.2 Impact of Different SOR Iteration Counts

Since the SOR solver resembles a ‘classical’ iterative concept improving the initial guess for the solution with every step, it is worthwhile to state the runtime for different iteration numbers. Depending on the application, such insights might help to design a satisfying compromise between time and accuracy of the solution.

Table 6.2 and Figure 6.2 show the results of this test. The experiment reveals a significant constant investment at the beginning related to equation system setup and optional presmoothing, which almost vanishes if more than about 80 iteration steps are performed. Again, the graphs for different test intervals are overlapping, which indicates that no one-off expenses with respect to the whole benchmark setup are present.

6.3 Performance on the PPU

In the following, the benchmarks from the previous section are repeated on the PPU to be later consulted as a second reference value besides the original implementation on the Pentium 4. To distinguish them from each other, the PPU based graphs are always printed in green throughout the course of this thesis.

6.3.1 Impact of Different Image Sizes

Here, the resampled versions of frames 8 and 9 of the Yosemite sequence are again evaluated by an SOR solver. As before, optional smoothing with variances $\rho = 2.3$ and $\sigma = 1.3$ for the 316×252 pixels large version and appropriate values for the differently sized frames are applied to the motion tensor and the input images, respectively. However this time, the program is entirely run on the PPU.

Figure 6.3 and Table 6.3 depict the results of this benchmark. Comparing them to those achieved on the Pentium 4 with equal clocking, they are only about 0.3 times as high, which also substantiates the subjective impression that the PPU is quite slow being used as a Desktop processor. Indeed, the performance of the PPU is often compared to an Athlon CPU with 1.33 GHz [60].

6.3.2 Impact of Different SOR Iteration Counts

For the second benchmark, the frames per second count for 1, 100 and 200 iterations of the SOR solver has been evaluated on the PPU.

The results are given in Table 6.4 and Figure 6.4. Again, the performance compared to the Pentium 4 is rather poor, but the general appearance of the curve is still comparable to the previous case, which underlines the general purpose architecture of the PPU.

Table 6.3: Performance (FPS) of 100 SOR iteration steps on the PPU over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
79 × **63**	23.48	24.01	24.57	22.33	22.53	22.73
158 × **126**	5.35	5.36	5.36	4.97	4.98	4.99
316 × **252**	1.30	1.31	1.31	1.17	1.17	1.16
632 × **504**	0.32	0.32	0.32	0.27	0.27	0.28

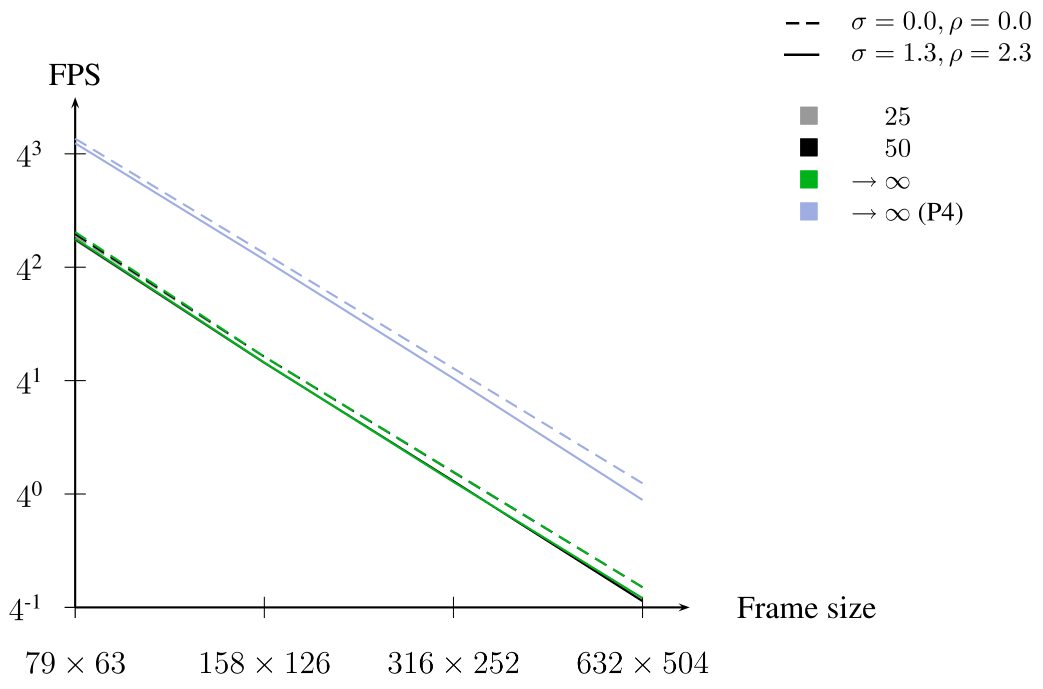

Figure 6.3: Performance (FPS) of 100 SOR iteration steps on the PPU over different image sizes. The Pentium 4 reference graphs are depicted in light blue. Like on the P4, the graphs for the different test intervals overlap.

Table 6.4: Performance of different SOR iteration steps on the PPU over images of size 316×252 pixels. Columns denote test intervals averaged over. Rows display the time per frame, the time per iterations, the frames per second (FPS), and the iterations per second (IPS), respectively, for 1, 100 and 200 iterations, each.

		No Smoothing			**Smoothing**		
	it	**25**	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
	1	41.04	40.44	39.83	132.13	132.19	132.26
t [ms]	100	768.02	764.43	760.84	854.35	856.73	859.11
	200	1497.61	1496.77	1495.93	1587.78	1590.58	1593.39
	1	41.04	40.44	39.83	132.13	132.19	132.26
$^t/_{\text{it}}$ [ms]	100	7.68	7.64	7.61	8.54	8.57	8.59
	200	7.49	7.48	7.48	7.94	7.95	7.97
	1	24.36	24.73	25.11	7.57	7.56	7.56
FPS [s^{-1}]	100	1.30	1.31	1.31	1.17	1.17	1.16
	200	0.67	0.67	0.67	0.63	0.63	0.63
	1	24.36	24.73	25.11	7.57	7.56	7.56
IPS [s^{-1}]	100	130.21	130.82	131.43	117.05	116.72	116.40
	200	133.55	133.62	133.70	125.96	125.74	125.52

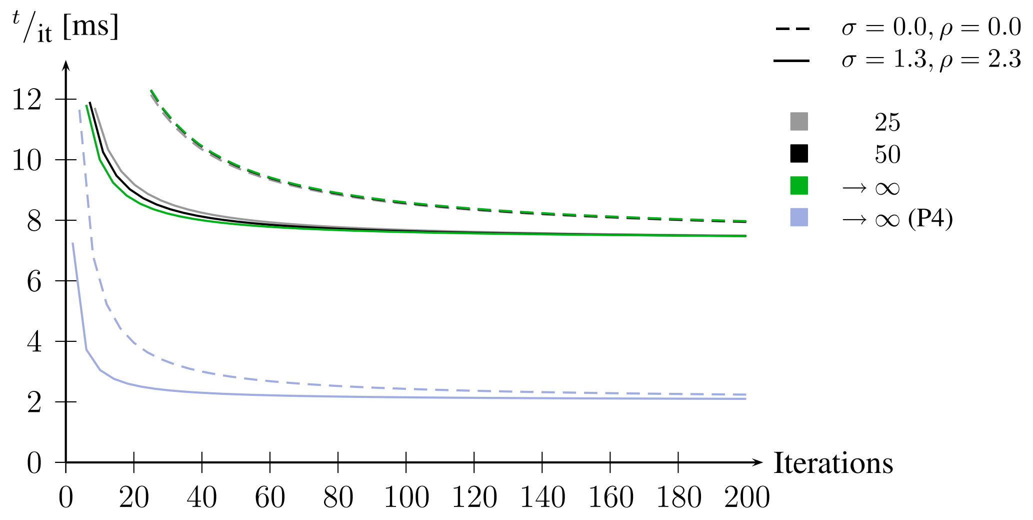

Figure 6.4: Time per iteration step on the PPU over images of size 316×252 pixels.

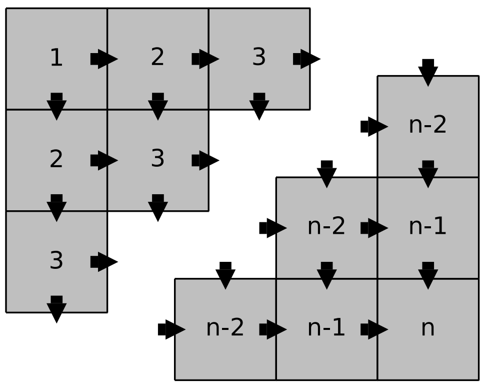

Figure 6.5: Dependency diagram in the simple Block SOR parallelization attempt. Tiles equally labeled can be computed at once.

6.4 Block SOR

6.4.1 Motivation and Idea

The first parallelization attempt transfers the basic sequential SOR solver to a distributed algorithm consisting of a spatial decomposition variant processed by the SPUs, while the setup of the equation system is still left residing on the PPU. Hereby, the space is divided into tiles of 32×32 pixels in size, thus creating a form of memory level parallelism (MLP). Each block is fed into an SPU implementation of a standard SOR solver for the Horn and Schunck model with the CLG extension (cf. Section 5.4). This parallelization attempt follows closely the *Computational acceleration model* presented by Kahle *et al.* [40].

In a global scale, this parallelization attempt still requires the image to be processed from top left to bottom right, which is especially important for the boundaries betwen the tiles. It implies that for the ideal setting, the problem scales to as many cores as there are tiles available on a grid diagonal of the image: Starting with the top left corner tile, the dependencies for two more blocks, namely the bottom and right neighbors, are fulfilled, which by themselves create the basis for their three right and bottom neighbors, and so on (cf. Figure 6.5). This way, whole diagonal arrangements of tiles meeting at the corners can in theory be processed at once, which at the first glance suggests a diagonal-wise execution.

On closer examination however, this first assumption needs to be restricted. The limited number of six available SPUs on the PlayStation 3 (cf. Section 4.1) would cause the program to fully occupy all SPUs in only one out of six cases, namely when the number of tiles along one grid diagonal is exactly a multiple of six. For a better scheduling, the reordering process depicted in Figure 6.6 is being applied, which in principle allows parallel execution of several pieces without unnecessary stalls, as long as dependencies are still met.

In particular, this means that for grid diagonals of less than six blocks, the number of unoccupied SPUs throughout the process does not change at all. The naive way to do so while respecting dependencies is to always start with the first SPU, and to process as many additional blocks by neighboring SPUs as there are currently cleared to be executed. In the beginning,

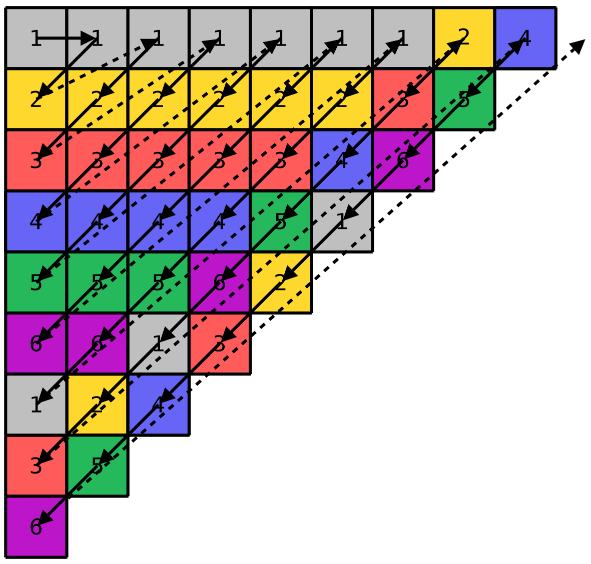

Figure 6.6: Scheduling in the simple Block SOR parallelization attempt. Note that dependencies between blocks are still an issue.

this leads to a situation, where the first line is entirely processed by the first SPU, the second by the second SPU, and so on. Due to the equal block size and algorithmic complexity, the time needed for the computation of one block can hereby be regarded as a constant throughout the algorithm.

However, as soon as the grid diagonals become longer than six blocks during further execution, there are more tiles cleared for being processed than SPUs are available, which allows SPUs to already start with the next diagonal if they are no longer needed in execution of the current one (cf. Figure 6.6). This method is efficient, but it also requires a more sophisticated tracking and scheduling of positions, because one SPU is no longer restricted to dedicated lines, but to blocks spread all over the frame.

Since the SOR solver typically involves many iteration steps, this would mean that for the present scheme, SPUs are unoccupied at the beginning and the end of each iteration due to unmet dependencies, which introduces performance leaks. Extending the dependency graph over iterations however, one easily sees that there are still many fulfilled dependencies in the following step available, such that idle SPUs can in principle already start with the next iteration, even though one step is not entirely finished yet: Instead of a strict dependency between the frames of consecutive iteration steps, every block only depends on three other blocks, namely its top and left neighbors, as well as its own predecessor in the previous iteration. As soon as all of them are valid, the respective block can already be processed, independent from whether it is officially in the iteration step currently processed, or not.

For sufficiently large frames, this observation immediately increases the number of approved tiles significantly: While the algorithm processes the bottom right corner of the frame in one iteration step, it is already allowed to start with the top left corner in the next iteration.

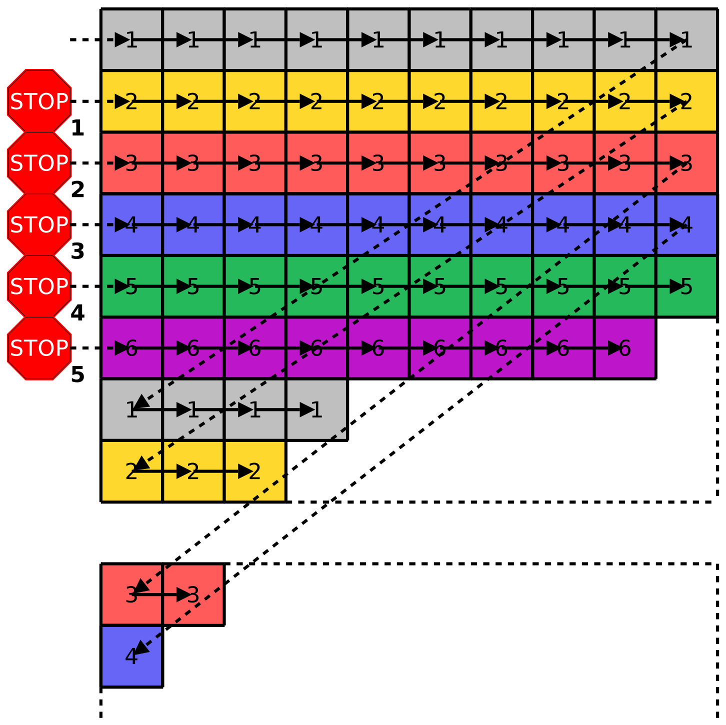

Figure 6.7: Simplified scheduling in the simple Block SOR parallelization attempt.

In particular, as the length of diagonal arrangements increases by as many tiles in the top left corner of the subsequent iteration step as it decreases in the bottom right corner of the preceding one, surplus resources can fully be invested into the next iteration step. This way, SPUs are only unoccupied to the beginning and the end of the whole process.

A drawback of this method is however that an additional administrative data structure or inline scheduling would be required at PPU side keeping track of the blocks already processed and those able to be executed. The first and the last iteration step would require particular attention, and for the iteration steps in between, six different scheduling patterns can in principle occur.

Taking all these insights together, scheduling can be further simplified without losing too much performance. Introducing the notation of a *virtual y axis* for scheduling, the frame representation over several iteration steps is assumed to be in consecutive vertical order. For instance, if a frame is split into eight lines of blocks, the ninth block line is then defined to equal the first line of the second iteration, and so on (cf. Figure 6.7).

This simplification not only sustains the line-wise setting already observed in the top left corner of the frame in the first iteration step, it also allows SPUs to schedule themselves without interaction of the PPU, just by adhering synchronization among them. Hereby, SPUs are assigned all tasks for blocks residing in lines of which the number equals the SPU's ID modulus six. The first SPU is hence meant to process the first, the seventh, the thirteenth line of tiles, and so on.

As long as one line of blocks is processed, no further dependencies need to be actively respected, because synchronizations holds the diagonal-wise processing idea automatically upright and dependencies from the left are trivially fulfilled by sequential execution. As soon as one line of blocks is completely processed, the next line can be computed by one addition modulus the height of the image, followed by an optional dead cycle insertion, if the frame is very small and the temporal dependencies are not yet met.

Dead cycles are maintained in one counter variable for each SPU kernel. They are initialized by zero to six, respectively, to ensure delayed dispatching at the very beginning and decreased whenever a synchronization takes place and they have not yet reached zero. Is the variable set to zero and a synchronization signal comes in, the next block is processed.

6.4.2 Synchronization

Following the considerations above, in principle only one synchronization signal is needed, broadcasted immediately after all participating SPUs have finished one block, each. However, on closer examination, this constraint already reveals to be too restrictive, because it forces a larger amount of SPUs to run to exactly the same times, instead of simply ensuring a strict ordering among them. This is not only a cosmetic issue, but can also immediately affect the running time of the algorithm: All blocks are of same size and can be assumed to be constructed from similar data. If SPUs processing those blocks were exactly synchronized, memory accesses by different SPU kernels are likely to coincide, thus creating bursty traffic on the memory interface. For sufficiently large data sets to acquire, this could lead to memory stalls and thus to artificially introduced performance losses.

Such strict ordering between SPUs can even be established easily, namely by introducing a token specifying the one SPU maintaining the highest dependency in the current setting. As soon as this SPU finishes its work, a new, lower dependent task is assigned to it and the token is passed to the next SPU in order. Additionally, it is usually desireable to uphold a continuously filled schedule for all SPUs, instead of allowing lower constrained SPUs to outrun. Latter will at some point at a later stage cause stalls anyway, when this SPU then needs to wait for others to finish.

Algorithmically speaking, this setting can most easily be created by exchanging synchronization information with all SPUs in a round-robin manner (cf. Figure 6.8). Focussing on one SPU, the PPU first waits for a status report of this SPU, which is can either be `STAT_RDY` or `STAT_FIN`, depending on whether the SPU expects a new clearance signal from the PPU, or has already finished its whole part for this frame, respectively. As all previous dependencies are already met at this stage, the PPU immediately responds with `SIG_CLR`, indicating that the SPU kernel may now process the next tile.

Finished SPU kernels are not automatically shut down, but wait for tasks for new frames to arrive. The PPU can place an information record in a previously defined shared memory section containing details about where to find the new frame, its dimensions, parameters and finally the iteration number chosen by the user. When this data set is valid, the PPU then sends a `SIG_TASK` signal to the SPU's mailbox which will acknowledge with `STAT_RDY` when it is ready to start the computation (cf. Figure 6.9).

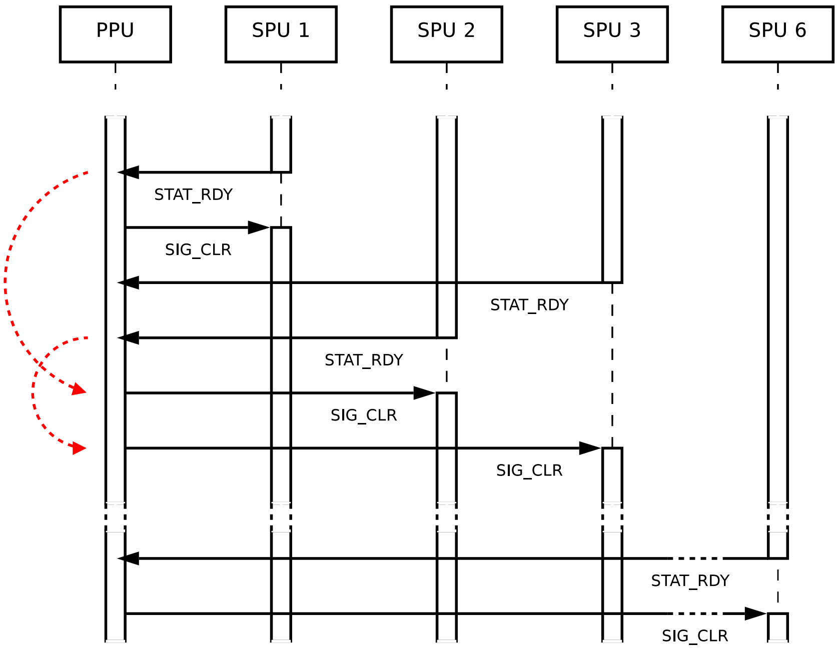

Figure 6.8: Bus protocol for Block SOR, excerpt from a running computation. In this example, SPUs 1, 2 and 6 finish in order, SPU 3 out of order. They are nevertheless redispatched in order again.

To shut down a SPU kernel, the signal `SIG_REL` can be emitted. It is not acknowledged anymore, but SPU threads are immediately terminated upon mailbox readout, independent from whether the current computation is already finished, or not.

6.4.3 Benchmarks

Performance on Different Image Sizes

In a first performance test, different image sizes have been fed into the Block SOR solver for the Horn and Schunck model with the CLG extension running on six SPUs. The goal of this experiment is to both show how the algorithm behaves compared to the sequential Pentium 4 implementation, and to see whether it is able to yield significant speedups, given other frame sizes.

Therefore, frames 8 and 9 of the Yosemite test sequence have been resampled to quartered, halved, and double edge size. The benchmarks include 100 iterations of the solver, including the setup of the equation system, each. Optionally, both input frames and the motion tensor entries are convolved with Gaussian kernels of variation σ and ρ, respectively, whereby the values have been chosen such that the general appearance is preserved over all sizes. Hereby, the variances have been set to $\sigma = 2.3$ and $\rho = 1.3$ for the original frame size of 316×252 pixels, and have elsewise been resampled analogously to the respective edge lengths.

For the benchmark, the running time of the first 25 and the first 50 frames has been eval-

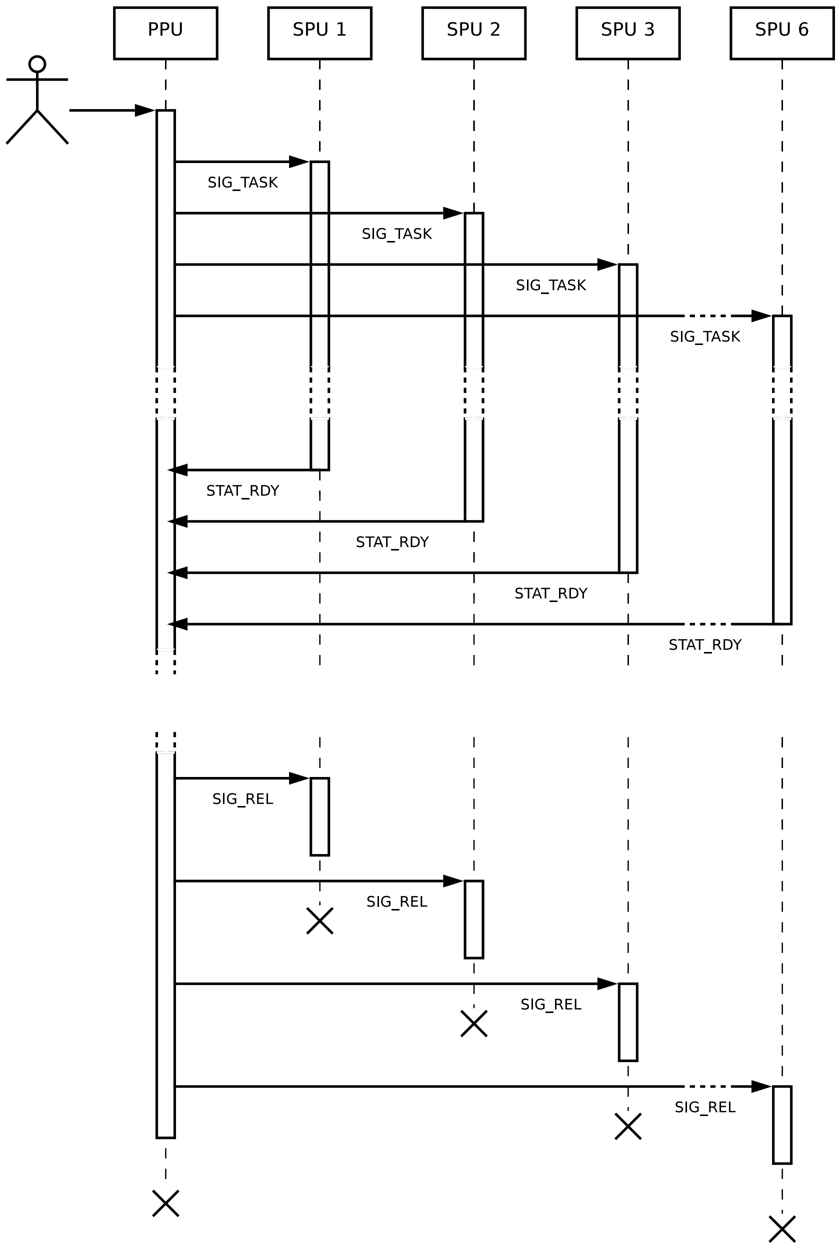

Figure 6.9: Bus protocol for Block SOR, (re)initialization for a new frame and termination sequence.

uated over a sequence of ten independent measurements, and averaged. These values, as well as the difference thereof, is then normalized to one single frame and inverted to yield the frames per second count. The difference, i.e. the measurement over the second 25 frames, is no longer subject to one-off expenses like memory allocation. It can hence be regarded equivalent to an extrapolation to infinitely large measurement intervals, and is indeed a feasible measure with respect to a realtime setting, where multiple frames can be assumed to continuously arrive from an external device or over an ethernet connection. In such realtime setup, the frame rate is likely to be situated somewhere in between the measurements with and without smoothing, because the smoothed version of one frame can here already been acquired from the predecessing time step.

Table 6.5: Performance (FPS) of 100 Block SOR iteration steps on 6 SPUs with PPU based equation system setup over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\to \infty$	**25**	**50**	$\to \infty$
79 × 63	49.03	50.08	51.17	43.69	43.84	43.98
158 × 126	23.40	23.65	23.91	17.49	17.32	17.14
316 × 252	6.21	6.20	6.18	3.93	3.95	3.97
632 × 504	1.47	1.47	1.48	0.79	0.79	0.78

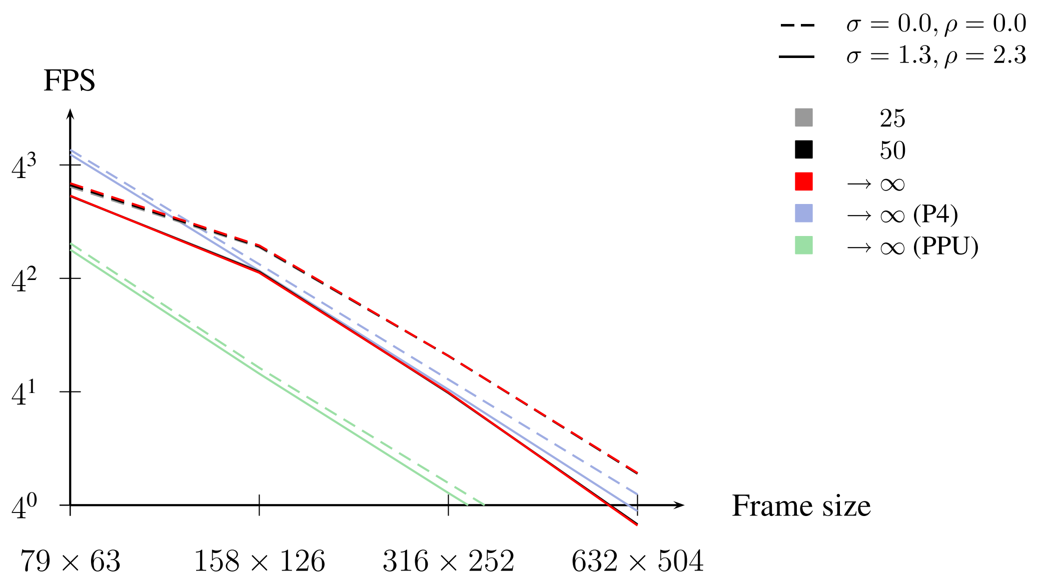

Figure 6.10: Performance (FPS) of 100 Block SOR iteration steps on 6 SPUs with PPU based equation system setup over different image sizes. Reference measurements for P4 and PPU are plotted in blue and green, respectively.

Table 6.5 and Figure 6.10 show the results of this benchmark. Noticeably, the length of the test interval has little impact on the measured values, in particular for the non-smoothing setting. This can be explained by the SPU kernels being restarted for every single test within one measurement, as well as by the fact that only the pure solver has been parallelized. One also sees that this problem scales bad towards small images sizes, which comes from the fact that a certain amount of time is needed until the SPU kernels are moved to their destinations, and this time typically does not pay off during the computation.

Indeed, the reverse conclusion stating almost linear decrease with a rising number of pixels, given greater frames, is in general true. Note in this context that both axes in Figure 6.10 are actually scaled logarithmically with base 4, considering the number of pixels in the respective images. The general appearance of the yielded graph is hence not distorted at all.

However, experiments with larger image sizes such as 1264×1008 pixels caused the joint RAM consumption of program, operating system and background daemons to exceed the physical memory limit of 256 MB, which forces the operating system to swap data out to the harddisk. The benchmarked results in this case are only about half as high as expected, and are thus not really comparable to the values given above.

This problem can unfortunately hardly be addressed by algorithmic means, since the current running time is mainly founded in the availability of precomputed linear system coefficients. As soon as these need to be recalculated in every iteration step, the performance will immediately drop significantly. Instead, considerations about RAM upgrades might be worthwhile in this context.

Comparing the results to the Pentium 4 benchmark (cf. Paragraph 6.2), they turn out to be quite comparable for reasonable image sizes, which already implies a speedup to the PPU setting. However, for very small images, the overhead introduced by SPU kernel initialization, as well as by the fact that only very few SPUs are permanently occupied, Block SOR performs noticeably worse than the sequential Pentium 4 benchmark.

Scaling with the Number of SPUs

The next experiment is meant to point out how well the algorithm scales over different numbers of SPUs. It has hence been restricted to a total number of $1, 2, 3, 4, 5$ and 6 SPUs, and was again evaluated against frames 8 and 9 of the Yosemite test sequence, this time fixed at the original size of 316×252 pixels. Like before, the measurement was averaged over ten independent benchmarks, each.

Table 6.6 and Figure 6.11 depict the results of this experiment. Like in the previous benchmark, the measured values are independent from the length of the test intervals. Obviously, the non-smoothing version scales almost linearly with the number of SPUs, because the PPU based equation system setup does not come into account too much considering 100 solver iterations. However, the PPU based smoothing process indeed deteriorates the performance of the second test series significantly. Assuming a higher number of available SPUs, the frame rate can in particular be expected to converge against a constant value, because the less time the pure solver consumes, the more will equation system setup and smoothing dominate the whole process.

Table 6.6: Performance (FPS) of 100 Block SOR iteration steps on different SPU counts with PPU based equation system setup over images of size 316×252 pixels. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\to \infty$	**25**	**50**	$\to \infty$
1 SPU	1.38	1.38	1.38	1.23	1.24	1.24
2 SPUs	2.60	2.60	2.61	2.10	2.11	2.11
3 SPUs	3.70	3.67	3.65	2.76	2.76	2.75
4 SPUs	4.68	4.70	4.72	3.36	3.34	3.32
5 SPUs	5.59	5.62	5.66	3.76	3.78	3.81
6 SPUs	6.21	6.20	6.18	3.93	3.95	3.97

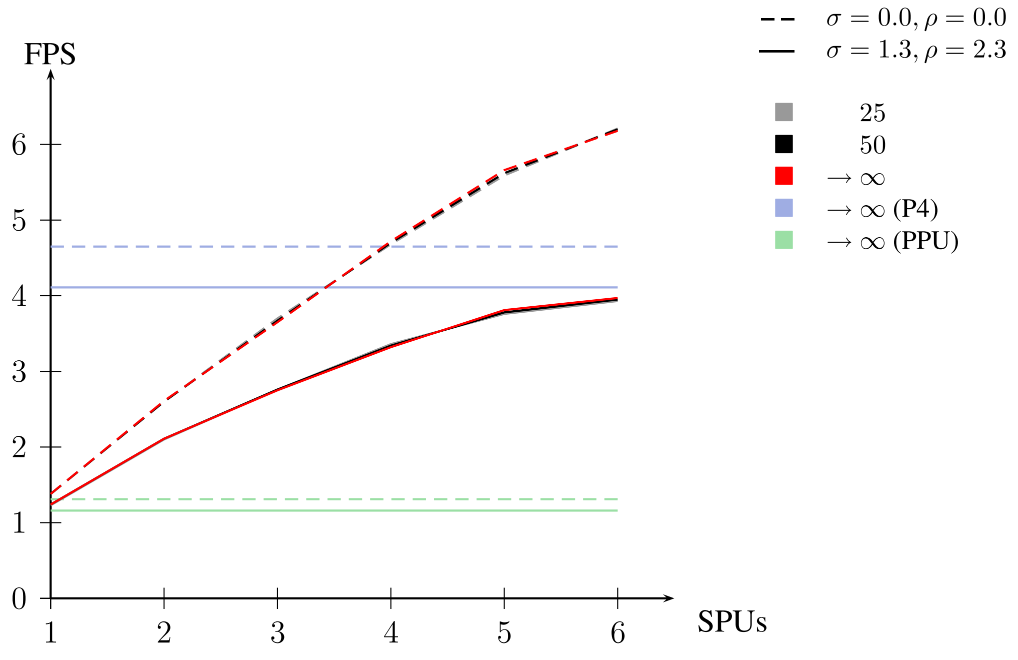

Figure 6.11: Performance (FPS) of 100 Block SOR iteration steps on different SPU counts with PPU based equation system setup over images of size 316×252 pixels. The reference benchmarks on P4 and PPU are again plotted in blue and green, respectively.

This benchmark clearly shows that the main problem is situated in the general design of the parallelization strategy rather than in a poor distribution over the available cores: Taking the theoretical performance of one single SPU into account, the technical eventualities are not fully exhausted yet. Even if the setting would perfectly scale over all cores, the gained speedup would still be way below satisfaction. Various reasons accumulate to this problem:

Since the SPUs are designed as a broadband architecture operating at 128 bits of width, all unused SIMD slots actually diminish the performance. However, the current concept can hardly be adopted to parallel execution, which is due to data dependencies within the formulation of the problem.

Additionally, because of the fixed block size, the runtime cannot be expected to increase linearly with the number of pixels, but will show sawtooth-like anomalies depending on both if the number of scheduled block rows is a multiple of the available number of SPUs, and on the percentage of scheduled block pattern actually covered and occupied by the image. Latter refers in particular to unoccupied block partitions at the bottom boundary of each frame per iteration step, and is related to forced idle-times during the strict scheduling.

To achieve higher capacity utilizations, algorithmic changes and an entirely new concept are inevitable. One approach to this problem is proposed in the next section.

Performance on Different Iteration Counts

In this last experiment, the impact of different iteration counts of the solver is determined, which creates a higher comparability with respect to further approaches presented in context of this thesis. Since different application fields often have different accuracy requirements, it is indeed worthwhile to chose the method depending on the actual needs, and simple approaches like the current one could in some cases still outperform more sophisticated ones. Therefore, the performance of the algorithm has been benchmarked for 1, 100 and 200 solver iterations, and as before, ten independent measurements have been issued for each scenario to stabilize the result.

The results of this experiment are described in Table 6.7 and Figure 6.12. Again, the length of the test interval only marginally influences the overall behaviour, especially for sufficiently large numbers of iterations. Since a perceptible partition of the runtime is accounted by the setup of the linear system which is still performed on the PPU, rather low frame rates are achieved for few iteration steps. However, with an increasing number of iterations, every single step is significantly cheaper than on the sequential reference implementations, such that the method pays off in these cases.

Comparing the benchmarks with those issued on the pure PPU, a speedup of between 3.4. to 4.7 could be achieved. This sounds a lot a first glance, but unfortunately, it is hardly more than a pure compensation of the losses introduced when porting the program to the PPU. Indeed, with respect to the Pentium 4 benchmark, the Block SOR method performs about as good, which is still way below the aimed goal to actually speed up the computations noticeably.

Table 6.7: Performance of different Block SOR iteration steps on 6 SPUs with PPU based equation system setup over images of size 316×252 pixels. Columns denote test intervals averaged over. Rows display the time per frame, the time per iterations, the frames per second (FPS), and the iterations per second (IPS), respectively, for 1, 100 and 200 iterations, each.

		No Smoothing			**Smoothing**		
	it	**25**	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
	1	37.79	39.90	42.02	133.20	132.56	131.93
t [ms]	100	159.08	161.36	163.63	254.26	253.01	251.76
	200	283.54	284.32	285.10	376.99	375.45	373.90
	1	37.79	39.90	42.02	133.20	132.56	131.93
$^t/_{\text{it}}$ [ms]	100	1.59	1.61	1.64	2.54	2.53	2.52
	200	1.42	1.42	1.43	1.88	1.88	1.87
	1	26.46	25.06	23.80	7.51	7.54	7.58
FPS [s^{-1}]	100	6.29	6.20	6.11	3.93	3.95	3.97
	200	3.53	3.52	3.51	2.65	2.66	2.67
	1	26.46	25.06	23.80	7.51	7.54	7.58
IPS [s^{-1}]	100	628.60	619.73	611.12	393.29	395.24	397.21
	200	705.36	703.44	701.52	530.51	532.70	534.90

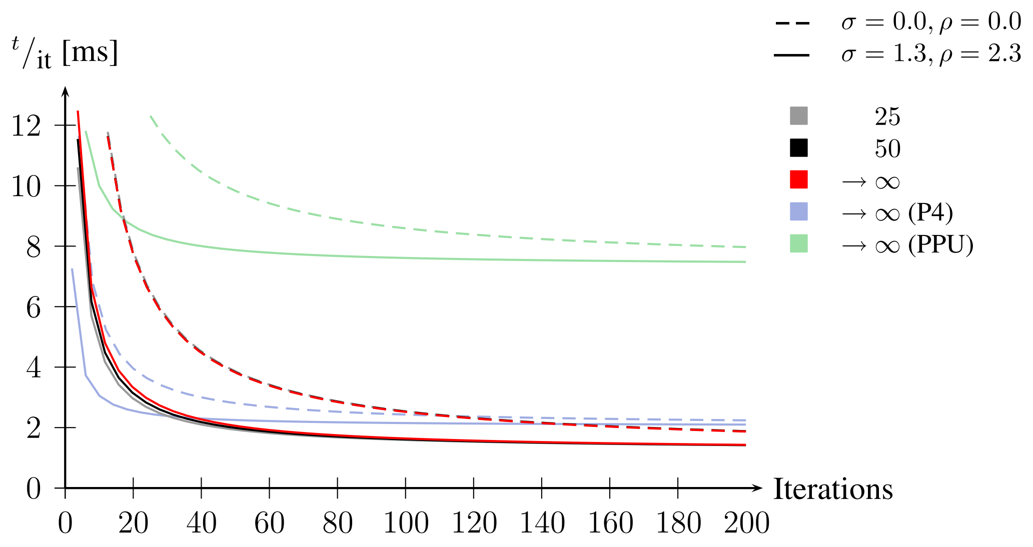

Figure 6.12: Time per iteration step on 6 SPUs with PPU based equation system setup over images of size 316×252 pixels. P4 and PPU reference values are plotted in light blue and green, respectively. Note that the choice of the test interval is of little impact again.

6.4.4 Discussion

Even though this algorithm is in principle scalable to a high level and can thus benefit from any available SPU, it also reveals some drawbacks. As described above, the image is segmented into equally sized tiles which are then processed individually. From a local perspective, the sequential SOR algorithm has however not been altered at all, i.e. each pixel still depends on four adjacent pixels, of which the left and top neighbors already need to be advanced to the next timestep. While this solution does not require any further assumptions besides a reliable scheduling of inter-tile processing, it also prohibits any form of instruction level parallelism, and especially excludes SIMD application (cf. Paragraph 3.5.3). This Problem is addressed in the next section.

Reading and writing DMA calls are always scheduled on the beginning and the end of one block computation, respectively. While these requests are pending, the SPU is entirely idle, which is an immediate consequence of the whole parallelization attempt: Because writing DMA requests must necessarily be finished before the synchronization signal is sent and reading requests can soonest be issued after such call, there is no other workaround to this other than overhauling the entire concept.

In addition, the data structure underlying this technique has not been optimized for this purpose yet. For this first experiment, the parallelization was introduced without deviating too much from the sequential setting to stay relatively comparable to this version. Because data sets have been allocated in a column-wise, i.e. not in a block-wise manner like it is convenient in sequential implementations, one block must still be fetched in many single memory interactions instead of being requested in one large bunch of data. Optimized reorderings would on the other side introduce additional constant computational overhead before and after computing a frame, which will doubtfully pay off for small iteration counts.

Within this setting, only the pure iterative solver has been parallelized. Even though it makes out a significant amount of the time needed to compute flow fields for a frame, other significant parts are given by the setup sequence of the linear system of equations, and by optional presmoothing steps at different stages. All of these tasks are still left to the PPU, not only involving missing parallelism, but also possibly introducing further memory-related delays: Parts of the coefficients computed on the PPU may still reside in its managed L2 cache and are hence fetched immediately from there when requested by an SPU, instead of from the RAM [48]. The PPU's MFC can hence be a bottleneck if many SPUs synchronously access those resources.

Even severe performance losses can occur on bottom or right image boundaries. The blocks are designed not to overlap more than one frame to minimize special case treatment. This situation would occur as soon as an image including boundaries is not a multiple of the block size. Instead, the image is padded by unmeaningful information to fit entirely into this scheme. Because this information is locally discarded, SPUs processing those partly occupied tiles finish earlier than others and remain idle for the rest of the iteration.

A similar issue can be found on a larger scale if the image is either very small, or if the last line of blocks executed in the last iteration step is not handled by the last SPU. While in the first case, there is no point in time where all SPUs are simultaneously working, some SPUs will be idle during the end of the computation, because they are not assigned a new line, while others are.

Taking all these considerations together, it becomes obvious that even by proposing solutions to certain performance issues, severe problems remain. Instruction level parallelism can for instance only hardly be ensured without radical changes to the underlying memory structure. Nevertheless, despite of only moving in about the range of the sequential Pentium 4 implementation by means of average frames per second numbers (cf. Paragraph 6.4.3), this method at least proved that a speedup of roughly the factor of four can be achieved compared to a pure sequential implementation on the PPU, which can be considered about as powerful as one SPU. The technical investment remains thereby manageable and also provides comparability between usual SOR solvers on conventional architectures and the parallel realization on the Cell Broadband Engine. In the next section, this comparability is partly abandoned in favour of advanced approaches to some of the problems mentioned above.

6.5 Red-Black SOR

6.5.1 Motivation

Motivated by the idea of a parallelization attempt on both memory (MLP) and instruction level parallelism (ILP), any trivial modification of the basic SOR algorithm can immediately be withdrawn, due to the strict dependency of any inner pixel from its left and top neighbors. Any successful application of SIMD instructions would require four independently processable pixels, which could for instance be located along a grid diagonal.

Since SIMD operations can only be performed on adjacent memory cells, a reordering step becomes inevitable. This linearization could for instance look similar to the scheduling proposed in Figure 6.6, just projected down to pixel scale. However, destroying the local context of each pixel this way is not beneficial at all: Using a four-neighborhood describing the relations established by the system matrix, spacial neighbors are no longer in trivial relationship to a regarded pixel like they are without reordering. Thus, not only complex lookup functions are involved, but any ILP speedup can potentially be lost as well, if the neighbors of a vector are not aligned in a vector by themselves.

Instead, an alternative variant of the SOR algorithm is used, namely *Red-Black Successive Over-Relaxation*. The underlying idea of this method is well-known in literature and bases upon a reordering of the processing order by means of a checkerboard pattern. Figure 6.13 depicts such arrangement. The entries are processed according to an increasing label number, while those equally colored can be executed at once. Because of these missing dependencies, equally colored elements can be distributed on all six SPUs performing these entries in parallel. From a global view, this parallelization attempt is again a concrete implementation of Kahle's *Computational acceleration model* [40]. Different to Block SOR (cf. Section 6.4) however, instruction level parallelism can additionally be applied.

A central observation leading to this approach are the trivial diagonal entries of the stencil applied in the standard SOR algorithm. They only allow direct information flow to be axis

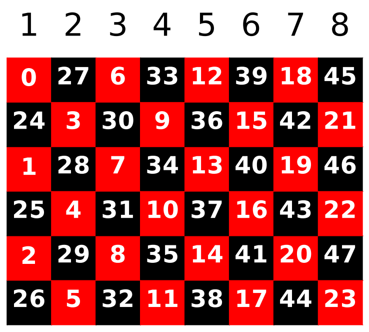

Figure 6.13: The Red-Black reordering scheme.

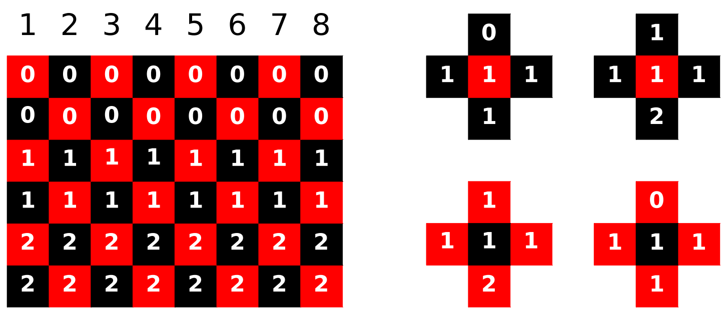

Figure 6.14: Left: The Red-Black reordering scheme, with generalized pixel indices. **Right:** Different neighborhood configurations are observeable.

aligned, or viewed from another perspective, information exchange between two pixels only meeting at a corner involves at least one intermediate neighboring pixel. The algorithm exploits this fact by splitting the computation into the exclusive update of red pixels, followed by pure black updates (cf. Figure 6.13). All pixels of equal type can therefore be computed at once, offering the opportunity of a high scalability to multiple cores, as long as changes between the red and black phases are correctly synchronized with all cores.

6.5.2 Reordering

In order to make use of instruction level parallelism, the red-black ordering scheme needs to be further analyzed and the pixels to be categorized into groups of similar behaviour. Similar pixels can then be grouped and used as SIMD vectors. Naturally, the red and black distinction creates a first criterion, because both groups are never touched simultaneously following above considerations.

Figure 6.14 shows a copy of Figure 6.13, but with pixel indices generalized to arbitrary image sizes. The general way how pixels are processed is hereby not touched, though. On the right, configurations for 4-neighborhoods visible in the scheme are displayed. Depending on whether a pixel is located in odd or even columns and on whether it is red or black, the top and bottom pixels can either be found in the corresponding location of the other vector and its

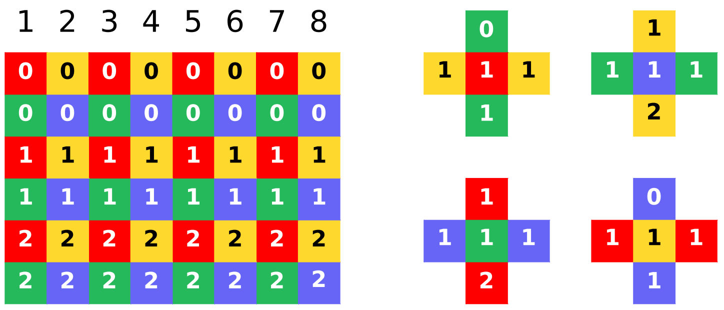

Figure 6.15: Left: The Red-Black reordering scheme with explicit distinction by neighborhoods. **Right:** Each color has unique neighborhood configurations.

predecessor, or in the corresponding pixel and its successor. To explicitly mark this difference throughout the scheme, these differences are expressed by two more colors (cf. Figure 6.15). This notation is equivalent to keeping the two-color red-black scheme and simultaneously applying different treatments based on the column currently being processed.

To use full instruction level parallelism, similar, i.e. homogeneously processable data must in any case be aligned in adjacent memory cells, thereby rendering actual reordering by means of memory operations inevitable. The data format chosen in context of this experiment is based on a double-column approach, this means that entries of each odd and the successing even row are mingled. In particular, assume two columns marked with a color coding as depicted in Figure 6.15. All entries marked red will be moved into the first half of the left column, the rest of this column is filled with blue denoted values. Analogously, the right column will be composed of the yellow entries, followed by the green ones. Local ordering is hereby preserved.

Even though this system might seem chaotic on first glance, it is indeed advantageous to reordering within single lines, for instance: The method described above consolidates whole columns of black or red entries in terms of the traditional red-black coding, which causes only every second row to be read and only every second row to be written when a four-pixel stencil is applied later for the solver. It hence forms a convenient compromise between blocks large enough for faster DMA transfers and little restrictions to the resulting image dimensions, which simply require the image to be padded by one bogus line if it has an odd width.

The naive way to address such reordering executed on SPUs is to fetch two columns of data, to assign the single floating point entries to their new locations, and to write these lines back. It could however experimentally be shown that this process can even be accelerated using the full bandwith of the SPU:

Data is thereby looked at in 256 bit blocks, and two columns are always processed in parallel to be able to exchange information (cf. Figure 6.16). A preliminary goal of this algorithm is to have each of the four 128 bit blocks in this setting homogeneously colored, because then these blocks can easily be sorted into the target arrays as it would have been performed on single 32 bit values above.

To get those blocks, the four 128 bit vectors regarded are first locally shuffled, whereby for the odd, i.e. upper vectors, the green or blue components are moved to the top, while the red or yellow components are in the other two vectors moved to the beginning. Both can be realized by the `spu_shuffle` intrinsic, which only needs one instruction per vector.

Afterwards, the first two elements of both vectors in a row need to be exchanged. This can easily be done by a masked XOR swap, whereby v_1 and v_2 denote the two vectors, and $\oplus_b$ and $\wedge_b$ are bitwise XOR and AND operators, respectively:

$$\begin{aligned} v_1 &\leftarrow v_1 \oplus_b v_2 \\ v_2 &\leftarrow v_2 \oplus_b \left(v_1 \wedge_b \left\langle 1^{64}0^{64}\right\rangle_2\right) \\ v_1 &\leftarrow v_1 \oplus_b v_2 \end{aligned}$$

This step indeed yields the desired result:

$$\begin{aligned} v_1 &= (v_1 \oplus_b v_2) \oplus_b \left(v_2 \oplus_b \left((v_1 \oplus_b v_2) \wedge_b \left\langle 1^{64}0^{64}\right\rangle_2\right)\right) \\ &= v_1 \oplus_b (v_2 \oplus_b v_2) \oplus_b \left((v_1 \oplus_b v_2)[0..63] \left\langle 0^{64}\right\rangle_2\right) \\ &= v_1 \oplus_b \left((v_1 \oplus_b v_2)[0..63] \left\langle 0^{64}\right\rangle_2\right) \\ &= (v_1 \oplus_b (v_1 \oplus_b v_2))[0..63]\, v_1[64..127] \\ &= ((v_1 \oplus_b v_1) \oplus_b v_2)[0..63]\, v_1[64..127] \\ &= v_2[0..63]\, v_1[64..127] \end{aligned}$$

and

$$\begin{aligned} v_2 &= v_2 \oplus_b \left((v_1 \oplus_b v_2) \wedge_b \left\langle 1^{64}0^{64}\right\rangle_2\right) \\ &= v_2 \oplus_b (v_1 \oplus_b v_2)[0..63] \\ &= (v_2 \oplus_b v_1 \oplus_b v_2)[0..63]\, v_2[64..127] \\ &= v_1[0..63]\, v_2[64..127] \end{aligned}$$

All vectors are at this point already consisting of equally marked elements and vectors on even positions are of ascending local order. In a last step, odd vectors need hence to be shuffled into the final layout, which involves exchanging the two 64 bit halves with each other.

As mentioned before, the newly created 128 bit values now only need to be copied to their final location. This whole algorithm implemented on all SPUs runs in 7.43 ms in average, which is measurably faster than the naive value based method taking about 12.50 ms with 8-way loop unrolling on two columns at once. Both benchmarks are based on measurements over seven matrices with 316×252 entries, each.

At first glance, this observation seems counter-intuitive, because an element-wise reordering is performed in both cases, though on different scales, and the SIMD based method includes even more explicit operations. For one iterateration step processing 16 single elements, latter method needs $4 + 8 + 2 + 4 + 8 = 26$ instructions for shuffling, masked swap, odd vector shuffling, four pointer operations, and eight load/store operations, respectively. Low-level assembly instructions used thereby are 'shuffle bytes' (`shufb`), `xor`, `and`, add immediate

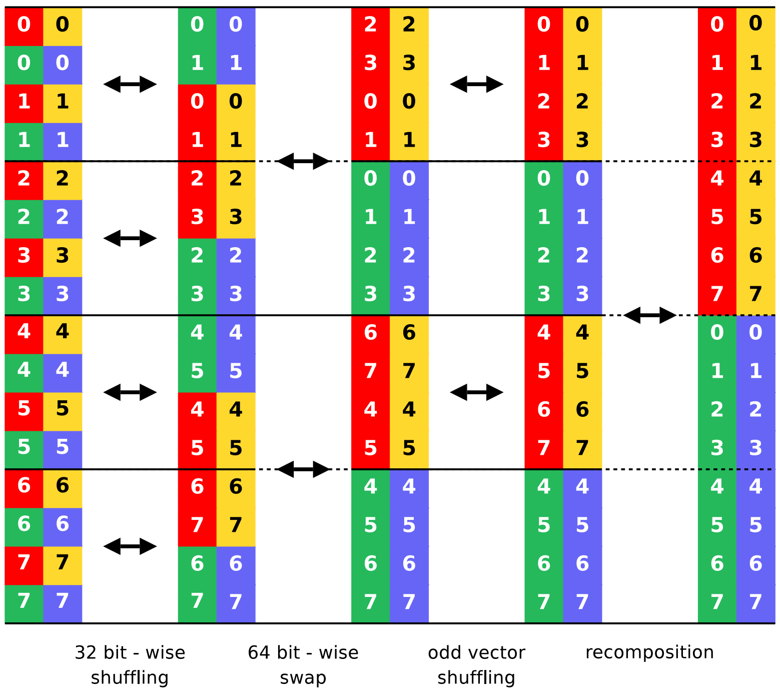

Figure 6.16: The vectorized reordering process for the multi-colored interpretation.

(`addi`), and load or store quadword (`lqd` / `stqd`) [33]. Because all vectors reside in the Local Store, all of these operations can be executed in at most one cycle. Load/store operations on the one hand and arithmetic computations on the other are even processed by different pipelines (cf. Paragraph 3.5.1), which allows down to 18 cycles for one iteration step.

However, the element-wise approach in contrast needs far more load and store operations: The largest coherent block of loaded and stored data consists of 32 bit in width, which basically forces the compiler to translate into 16 word loads (`lfs`) and 16 word stores (`stfs`), in addition to at least four pointer operations (`addi`) per iteration. Again assuming simultaneous load/store and arithmetic processing, at least 32 cycles are still involved. Taking this insight into account, the observed speedup can indeed be adequately explained.

6.5.3 Memory Representation

Comparing the actual solver step to the setup phase of the equation system, the requirements to the memory structure are quite different. Since convolutions with Gaussian kernels are global operations and for high variances, the dimensions of nontrivial elements in the kernel become quite large, a reordered set of thus misplaced elements is rather counter-productive. Since this observation is also partially true for derivative approximations which are after all easier to handle due to their limited support, it is convenient to issue the whole coefficient computation on ordered values, then to reorder them for an efficient use inside the solver, and to finally backtransform the solution.

Matrix-valued data can be held a coherent memory block which for instance simplifies the handling of DMA requests for successing blocks. Two meaningful ways of realization can be considered in this sense, namely a row-before-column ordering, which corresponds to the usual way of reading, and a column-before-row ordering, which allocates vertical neighbors in adjacent memory cells. Both methods turn out to be equally suitable for these algorithms, such that columns have been chosen to be allocated at a stretch.

For the processing, this suggests column-wise operations, whereby DMA read requests are issued on whole columns, these are then modified while residing in the Local Store of an SPU, and finally they are written back to RAM. Since memory chunks transferred between RAM and Local Store need to be aligned to 128 bit (cf. Paragraph 3.5.2), different image heights can hence represent a problem. This is why all columns are *ab initio* forced into an alignment, which implies that for some image sizes, bogus values are appended at the bottom boundary, which need to be respected for memory operations, but which must never be processed upon during arithmetic operations, since these inconsistencies introduced by uninitialized value might propagate over the whole image and thus deteriorate the solution.

For reordered vectors, even higher alignments are necessary, since the reordering step (cf. Paragraph 6.5.2) spreads meaningful data over 256 bits of width, even though they might then be interleaved with padded bits. Both reorderings are hence additionally charged with a potential conversion between different effective memory dimensions.

Fortunately, row lengths need never to be expanded, even when a second field follows seamlessly. Since the whole frame size is an integer multiple of column lengths, and these columns are sufficiently aligned, the same holds automatically for the whole image, too.

6.5.4 Smoothing

To render the solution more robust against noise and high frequent structures causing misinterpretations, two intermediate results can be convolved with a Gaussian kernel (cf. Sections 5.2 and 5.4). On the way towards an SPU implementation, one soon realizes that a convolution in one direction is typically very expensive, while the other is not. This comes from the fact that unlike for derivative approximations, the size of the cutoff convolution kernel is unknown *a priori*, i.e. at compile time. However, the separability of the convolution offers the opportunity to consider the process alongside the axes independently from each other.

Alongside the cache direction no problem arises, as long as the maximal size of the convolution kernel is for instance bounded to the image dimensions. In this case, columns can simply be fetched into the center of a three times as large Local Store representation, and Neumann boundaries are applied by mirroring the affected vector elements into the outer thirds. Locally, the algorithm does then not differ from sequential implementations at all: For all pixels within the inner third, a sufficiently large neighborhood is weighted with a precomputed convolution mask and summed up into a fresh column vector. Latter vector is then written back to RAM. Because the storage representation has been extended sufficiently wide across the top and bottom boundaries, no further special case distinction is necessary.

Regarding this scheme, it turns out that instruction level parallelism attempts do not pay off in this setting, due to the inhomogeneous problem parametrization in the four potential slots of

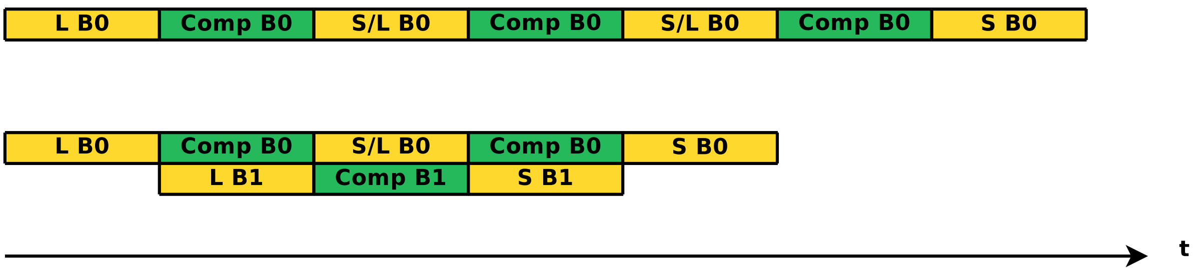

Figure 6.17: Double buffering for DMA operations (yellow) and computations (green). By backgrounded load and store operations, the actual runtime (bottom) can be significantly reduced with respect to the accumulated time (top). Note that two buffers B0 and B1 are needed if double buffering is to be applied.

both source vector and convolution mask. In the current implementation, columns are hence loaded one after the other, and the convolution is explicitly coded in a nested iteration over single values summing up weighted neighborhoods, before the whole line is finally written back.

The other direction perpendicular to the cache direction deserves closer attention, in particular because the whole neighborhood can in general not been kept in the Local Store and does in fact need to be continuously fetched. Essentially, two different DMA operations in terms of access frequency need to be considered: Columns are only sporadically written back, namely when a neighborhood has entirely been summed up in a weighted manner. In contrast, this neighborhood needs to be loaded column-wise beforehand, which typically happens far more often.

At least for the loading of new lines, which is issued quite frequently, DMA operations are going to be simultaneously issued with computations, such that memory operations are backgrounded. This concept is often referred to as *double buffering* and uses the structural independence of MFC and SPU (cf. Figure 6.17) [38].

This guarantees little suffering from RAM latencies and possible bus contemptions at the Memory Interface Controller, since smaller latencies are more easily compensated for, since the time windows are generally stretched by the hopefully dominating computation step performed simultaneously. For simplicity, the low frequent DMA storing operation is still formulated in a single buffer variant, but in the course of the Full Multigrid solver (cf. Chapter 7), a more flexible framework is going to be presented allowing to interleave the remaining DMA operations as well.

6.5.5 Setup of the Equation System

Compensating for drawbacks realized in the Block SOR setting (cf. paragraph 6.4.4), several concepts are incorporated into the iterative Red-Black SOR solver and its equation system construction step.

First, a DMA double buffering approach is consequently pursued for all relevant data sets.

To answer the requirements imposed by this method, larger memory blocks processed by one SPU are necessary. This allows longer vectors to be loaded and stored at once, and thus minimize the overhead introduced by initiation or termination of DMA requests.

Such larger regions are no problem anyway, due to independent computation opportunities on equally colored values. In the course of this experiment, the problem has been divided into stripes of at best equal width, whereby each of these stripes is processed by one dedicated SPU. Synchronization hence takes place only twice per iteration, namely after each red and each black processing cycle, to ensure correct synchronization between the last column of one and the first column of the next core.

This division has further advantages: Because no SPU is privileged over any other, the load is equally distributed onto all available resources, except for at most one surplus columns per core which comes from the fact that images are not necessarily a multiple of six in width. Due to this fact, the time spans where SPUs are idle and waiting for synchronization can be minimized. In paragraph 6.5.7, these attempts are pointed out in detail.

For this approach, two independent buffers are allocated in the Local Store, each of them spanning all elements needed to compute one column in the reordered setting. While one column is processed, the complementary data set is refreshed by means of DMA accesses to the RAM and vice versa.

A second concept pursued in detail is the full native SIMD width for most operations. However, in contrast to the actual solver (cf. Paragraph 6.5.6), which is entirely implemented in full SIMD width, certain operations like convolutions alongside the cache direction are still formulated in a scalar manner.

Like derived in Section 5.6, the equation system can be set up as follows. First, the motion tensor entries are computed based on derivative approximations. Afterwards, several precomputations can be performed on the tensor with respect to its position in the final equation system to save time during the iterative solver step.

Lets analyse all datasets needed for derivative computations first. For the y direction, columns can in principle be fetched one after the other. However, unlike for the smoothing convolutions with Gaussian kernels of arbitrary variance, the width of the convolution is known in this setting, since the Taylor expansion applied has a fixed order of five. This does not only offer the opportunity to hardcode resulting boundary situations and thus save computation time in the iteration steps, but it also allows a much more efficient notation in the x direction:

Here, it is hence convenient to process data with an offset of two columns to the loading step, and to keep a total of five columns in the Local Store. This provides enough information to compute the x derivatives of the whole central column, since every element thereof can resort to a five elements wide neighborhood. Furthermore, because the y derivative only depends on the central column itself, it can instantly be determined as well.

More precisely, not even both input frames need to be held valid in five columns, but an interpolated intermediate version suffices, as the spatial derivatives does not depend on any other value.

In contrast, the temporal derivative rely on both input frames and cannot be reconstructed only based on the interpolated intermediate solution. However, since the latest interpolated column in the ring buffer needs to be computed out of both frames and they are therefore fetched into the Local Store anyway, it is convenient to compute the temporal derivative on the fly.

Regarding instruction level parallelism, one immediately sees that the interpolation step, the derivative computation in x direction, as well as the the temporal one all describe homogeneous operations. This is why they can easily be written in 128 bit width, i.e. as SIMD operations.

In y direction, such optimization is problematic, since the algorithm is inhomogeneous with respect to a SIMD vector and mask for the neighborhood weights would need to be misaligned by different amounts. Hence, this particular operation is most effectively computed in a scalar fashion, and to minimize special cases at the boundaries, the columns of the five-element ring buffer are designed more spaciously at both ends of the column to allow for the boundary pixels being actually mirrored into these cells. This way, boundaries do not need any separate treatment at all. In x direction in contrast, the algorithm indeed differentiates those columns from inner ones, since left and right boundaries need to be paid attention to anyway: Here, offset DMA operations must be scheduled appropriately, and so it is just convenient to embed boundary cases into the respective algorithm.

Writing the computed derivatives back, and then fetching them again to set up the motion tensor is rather expensive and does not pay off at all. Instead, one can use the fact that all derivatives are in the Local Store once, despite of the temporal one being two steps ahead. To join them at one point in time, latter can hence be delayed by a three-element ring buffer written to in a round-robin manner. Out of the three derivatives for one particular column, the motion tensor can entirely be computed and written back to RAM.

Despite of the five and three columns lasting ring buffers described above and used for internal computations, the variables for the input frames and the output tensor only needs to be dimensioned as two columns for each vector involved in a DMA operation, since one column suffices for internal computations, and the other represents the second buffer to apply double buffering. Scheduling image sizes up to about 1024 pixels in height, 22 entire columns of single precision **`float`** values are only occupying about 88 kB of Local Store, which is still affordable.

Furthermore, this routine is applicable for one more optimization, which can be used to save many DMA operations at another point: In Paragraph 6.1.2, iterative solvers have been presented, and their property to rely on some initial solution has been discussed. To reduce the total number of solver iterations needed until a satisfying solution is yielded, the directional flow field components are often both initialized with zero, since this represents the mean of possible solutions. Instead of issuing an initial dedicated call over all columns, this action can be interleaved into the motion tensor computation step, where it turns out to be a lot cheaper:

When the flow field components u and v are to be initialized, basically only one column residing in the Local Store, which only needs to be set to zero for one time, must be written into all columns of the target vectors. Since no arithmetics is involved in this step, the process would come down to a exhaustive use of the MFC, while keeping the SPU idle. This is where the present routine comes into play: It consists of many expensive arithmetic operations like multiplications, and the time needed by them can this way comfortably be used to issue various DMA interactions in the background, like the mentioned initialization step.

When this routine comes to an end, the motion tensor can optionally be smoothed by the routine described in Paragraph 6.5.4. Afterwards, one last optimization is applied, before the system is given into the actual solver: The denominators of the discretized SOR equations (cf. Section 6.1.5) are constant over the iterations. Instead of dividing by them in each step, they can be precomputed and written as a multiplicative value. This way, one can profit of the significantly faster computations of multiplications taking seven cycles, compared to those of divisions which need to be written in software and are thus way more expensive.

6.5.6 Solver

Like for the linear system setup above, the main concepts replacing low performant program parts are again applied to the solver. Two redundant data sets for all vector-based values allow double buffering for DMA operations and computations, which is in particular interesting for values only used once per semi-iteration in the solver. Besides these, there are values accessed more often, like the intermediate solution of the previous time step. For instance, during a ‘black’ computation, some right neighbor of one pixel is to the same time top and bottom neighbor of a black pixel residing in the next column, and is eventually a left neighbor to black pixel of the column right to the previous. Hence, pursuing a left-to-right strategy, it is indeed worthwhile to keep such column for a longer time in cache.

These memory management functions, i.e. conditional loading of reusing of columns, and finally the writing of result vectors, is encapsulated in dedicated functions. The calling program only needs to request a certain column from the loading routine, and this function then checks whether the request can immediately be answered. If the requests come too frequently, it blocks until the data is available and valid. Then, the column needed in the next step is already requested from RAM, or copied from an existing cache copy to be available when the next request comes in. Similarly, the writing routine will block if the last DMA operation has not been finished yet, and returns as soon as the DMA request has been dispatched to the MFC. The major part of the actual memory synchronization hence happens in the background while arithmetic operations are performed in the program. From this perspective, these DMA routines can hence be seen as software implementations for cache management circuits, and this idea will be seized on in more detail in context of the Full Multigrid solver (cf. Chapter 7).

On left and right image boundaries, as well as at transitions between stripes, these memory function additionally need to handle special cases: On the left, no DMA operation is currently running when the first request arrives, hence the initial setup needs to be established by fetching one whole column, waiting for to arrive, and finally starting the next access to continue with the base cases afterwards. Similarly, the last writing request on the right needs to be immediately terminated to sustain the consistency.

In this strict column-wise setting, the construction of pointer hierarchies over matrix-valued data sets makes little sense. The whole problem cannot be held in the Local Store anyway – two input frames and a 2-D flowfield tensor for a 312×252 pixels large setting would already require about 1.3 MB of data entirely filling the six SPU's Local Stores, which even worsens if intermediate results and the SPU kernel itself is additionally considered. Pointers to single columns would therefore only come into play when the respective columns are involved in DMA operations, which always occur on the whole dataset by means of an iteration. In these cases, pointers can efficiently be carried along by one addition per iteration. Pointer hierarchies are hence entirely discarded for the sake of speed and small Local Store loads.

Like mentioned earlier (cf. Paragraph 6.5.2), the chosen format for reordered vector data is still not perfect when it comes to computation, because in the reordered data format, exactly one out of top and bottom neighbors is still residing one element offset to the current element processed. Talking of SIMD-based algorithms, this is equivalent to vector misalignments.

The naive approach to fall back to a scalar notation at this point does typically not pay off, compared to a solution re-establishing perfect alignment: The *SPU Language Extension* documentation [58] mentions a misaligned fetch only involving three SPU intrinsics. Based on these, two inlined functions for misaligned fetches of 32 bits to the left and the right have been included in the framework. Again assuming simultaneous load and arithmetic operations and shift amounts known at compile time, each of them only introduces three additional cycles of delay.

The basic idea is in both cases to combine the logically left shifted version of the left vector with a logically right shifted variant of the right vector by bitwise OR. In particular, the left misaligned vector can be yielded by

```
*output = spu_or(spu_slqwbyte    (*(input - 1),  12),
                 spu_rlmaskqwbyte(*input,        -4));
```

while the load instruction misaligned by 32 bits to the right is described by

```
*output = spu_or(spu_slqwbyte    (*input,         4),
                 spu_rlmaskqwbyte(*(input + 1), -12));
```

Processing a column in SIMD width causes problems on the top and bottom boundaries, since a distinctive treatment of boundary values is associated to massive deployment of branching. Instead, boundaries are first processed as if they would be inner entries, and because the Flow field boundaries are usually set to zero to simplify the solver step, these potentially wrong values are just cancelled out afterwards by a precomputed, 128 bit wide binary mask and a 128 bit logical AND operation.

However, the general structure of the solver does not differ too much from standard sequential solutions, though. Besides the described changes and additional memory management routines, an additional counter is needed determining which of the four 'colors' in terms of the extended red-black scheme is currently being processed. By a case distinction, the matching solver algorithm for the current color incorporating the respective misalignments and data scopes is selected. All in all, the code remains fairly well readable, since standard arithmetic operations are overloaded with their vector based correspondences and operations and at many points, no explicit prefix notation using SPU intrinsics is hence necessary for arithmetics. Within the DMA routines consisting of MFC commands however, the special hardware related context is indeed clearly visible. This explicit division of architecture-specific code from rather platform-independent parts has proven to be both fast and convenient to write and debug, and is hence going to be pursued in more detail in course of the Full Multigrid implementation (cf. Chapter 7).

6.5.7 Synchronization

Solver

Observing the boundary dependencies between stripes, two essential relations become obvious. The most restrictive of both is that the rightmost column of one stripe in the last semi-iteration necessarily needs to be processed before the leftmost column of the right neighboring stripe. Because stripes can be assumed to take equally long for one semi-iteration, the dependencies will not be met significantly earlier than closely before they are needed. Another dependency can be found at the same boundary, but in opposite direction: The leftmost column necessarily needs to be finished before the rightmost column of the left neighboring stripe in the following iteration semi-step is started. Taking sufficiently wide stripes into account and assuming almost simultaneous dispatching times for all stripes within one iteration semi-step, these dependencies can be regarded to hold as soon as the first constraint is fulfilled.

Synchronization can hence be simplified to an approach very similar to the one presented in context of Block SOR (cf. Section 6.4.2). One token is handed around from SPU to SPU and sent away as soon as one stripe is being started to process. Figure 6.18 depicts the synchronization routine, while abstracting from the PPU still being the communication provider which comes from still using the mailboxing concept.

At this point, the synchronization attempt invoking signal notification registers (cf. Section 3.5.6) would offer a convenient solution. Instead of charging the PPU with the distribution of synchronization tokens, the SPUs could easily forward the token they received to the next core. However, experiments with this technique using **`libspe2`** were unsuccessful.

Following the information found in the SPE Runtime Management API documentation, the first Signal Notification Register within the Problem State Area of each SPU's Memory Flow Controller has been memory mapped as soon as the kernels have been started [28]. Information about the mapped memory address is then passed to the respective predecessor in terms of the token propagation, using the mailboxing technique. As soon as all SPUs are ready, the first SPU is given the `SIG_CLR` signal, which starts the protocol. The PPU can at this point start to wait for the threads to return, because it is not involved in signaling at all.

Experiments showed that for the first few rounds, everything works as intended. However, after some iterations, messages seem to be misrouted. To visualize this problem, a test case has been designed, using debugging output whenever a message is sent away. Such report consists of the sender's and the receiver's IDs, as well as a unique message content. Receivers additionally acknowledge any incoming message by what they view to be a duplicate of the transmission report, such that the programmer can see whether a message has been correctly delivered, or not. During the first rounds, messages are pairwise identical, which at the time being proves the concept to work. After a while however, messages continuously arrive in registers belonging to SPU kernels which should not be in immediate conversation, following

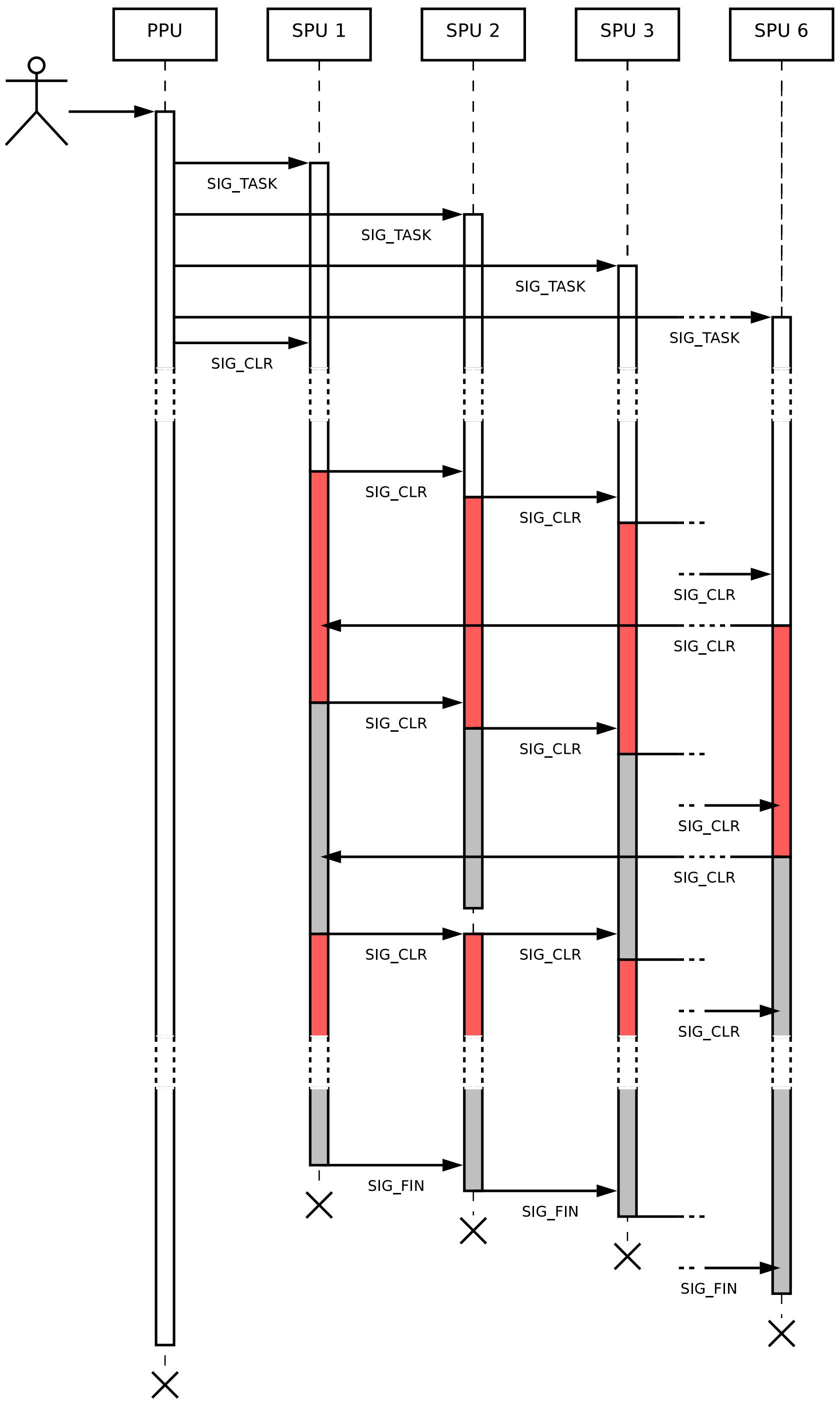

Figure 6.18: Synchronization for the Red-Black SOR solver. The different coloring of execution rectangles depicts the current partition processed.

the protocol. For instance, a message originating at SPU 1 and heading for SPU 2 surprisingly arrived at SPU 5, which should never happen since the problem state register addresses of SPU 5 have never been published to SPU 1. It should hence be technically impossible for SPU 1 to reach SPU 5 by this method.

A web search in the web unfortunately revealed that the concept of signaling does not seem to be officially supported by the developers any longer, at least not under SDK 2.1. In the sample program suite distributed with the Cell Broadband Engine SDK 2.1 [36], no example makes use of this technique. The official developerWorks Forum maintained and moderated by IBM lists repeated requests of software developers concerning a reference implementation for the performance results yielded by Chen *et al.* [19], which are said to be hardly achievable using the first SDK version [2]. Regrettably, these requests are unanswered by IBM officials since August 2006.

Because of this, the bus protocol has preliminarily been changed into a mailboxing approach until further information about working implementations with **`libspe2`** is published. Until then, signalling is emulated by using the PPU's inbound mailbox as a gateway to the successing SPU, thus again charging the PPU for communication management. The `SIG_FIN` signal sent around to make SPUs finalize is instantly used by the PPU to determine a closing protocol and to switch to the thread termination phase.

Preparation Phase

Besides the pure solver and the reordering steps described in Section 6.5.2, the equation system setup has been parallelized as well. However, in context of this experiment, the four kernels have been kept separated from each other to provide modularity.

Reordering does not need any synchronization at all, due to the locally restricted support of the computation and the missing dependencies between processed blocks. The PPU can just dispatch the charged SPU, and wait for it to return. For the setup of the equation system, this holds in principle as well, except for two actions: The convolutions optionally smoothing both the initial image and the computed motion tensors work on a global scale. In particular, as the image derivatives involve a neighborhood as well, data validity needs to be ensured in all locations covered by the stencil. Afterwards, when the motion tensor is to be convolved with a Gaussian kernel, its computation must again necessarily be terminated before doing so. Hence, four synchronization stages are passed:

Initially, the programs are started one after the other and first perform a convolution in x direction, if necessary. Results are hereby written into a spare vector. Because this operation overlaps stripe boundaries, the input needs to be kept stable until this stage is left by all other SPUs. Here, all SPUs report their process completion to the PPU by sending a `STAT_RDY` signal, which reactivates the kernels by returning `SIG_CLR` as soon as all state reports arrive. While convolving in y direction, the images are instantly written back into their initial memory cell. Following this process, the SPUs again need to be synchronized against each other to make sure the values spatial derivatives are computed on are actually valid, i.e. correspond to the entirely smoothed image. The messages exchanged are hereby the same as before.

The next operation with theoretically global support is the smoothing step of the motion tensor. As before, one such full synchronization is needed in between the stage convolving all tensor entries in x direction, and the process of doing the same in y direction. However, different to the input image smoothing, where all SPUs were trivially synchronized due to being started up, they now need to be explicitly adapted to each other. This comes from the fact that each SPU needs to rely on the complete motion tensor to be already fully computed before issuing any convolution operation on it. Here, no subsequent synchronization is necessary, because the following operation, a precomputation of actual linear system components, only has local support. It can thus work on available data, regardless from the status of the neighboring SPUs.

6.5.8 Benchmarks

Performance on Different Image Sizes

For the current Red-Black SOR solver implementation, the scaling behaviour over different image sizes becomes really interesting, since loading of the four kernels is quite time consuming and certainly penalizes small image sizes more than large ones. Like for Block SOR, frames 8 and 9 of the Yosemite sequence have been rescaled, and optional presmoothing with variances of $\sigma = 2.3$ and $\rho = 1.3$ has been applied for the original frame size of 316×252 pixels. For differently sized images, these variances are again scaled according to the edge lengths.

Indeed, as Table 6.8 and Figure 6.19 reveal, the performance is quite dependent from the image size. For small images, the parallel program even turns out to be way less performant than a simple sequential implementation on a Pentium 4. In contrast, it really benefits from ordinary or larger sizes, where it yields a speedup factor of about 3.1.

Anyway, like already stated in course of the Block SOR experiment, this statement needs to be partly put into perspective, since the 256 MB of RAM built into the Playstation 3 actually delimitates the maximal dimensions to be effectively computed to a little more than 640×480 pixels. Hence, this solver does at least at first glance not seem to be able to unveil its full potential, even though the performance gain sounds promising already.

Scaling with the Number of SPUs

Regarding the behaviour over changing numbers of SPUs, the current Red-Black solver implementation should certainly suffer from overhead introduced by transfers of kernel binaries to the SPUs. Compared to Block SOR however, it generally manages with less synchronization messages and the equation system setup step is parallelized as well. Depending on which of these arguments actually dominates the process, the Red-Black method should hence scale about as good as the Block SOR method, even though the basis given by the frames per second yielded on one single SPU should be shifted upwards already.

Unfortunately, these expectations can only partly be confirmed. Table 6.9 and Figure 6.20 indeed show a scaling behaviour comparable to Block SOR for small SPU numbers, but for the variant without smoothing, choosing more than four SPUs even deteriorates the results

Table 6.8: Performance (FPS) of 100 Red-Black SOR iteration steps on 6 SPUs with equation system setup over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
79 $\times$ **63**	25.69	25.71	25.72	24.90	24.87	24.84
158 $\times$ **126**	23.12	23.22	23.32	20.52	19.03	17.74
316 $\times$ **252**	14.12	14.35	14.59	9.14	9.28	9.42
632 $\times$ **504**	5.16	5.33	5.51	2.20	2.22	2.24

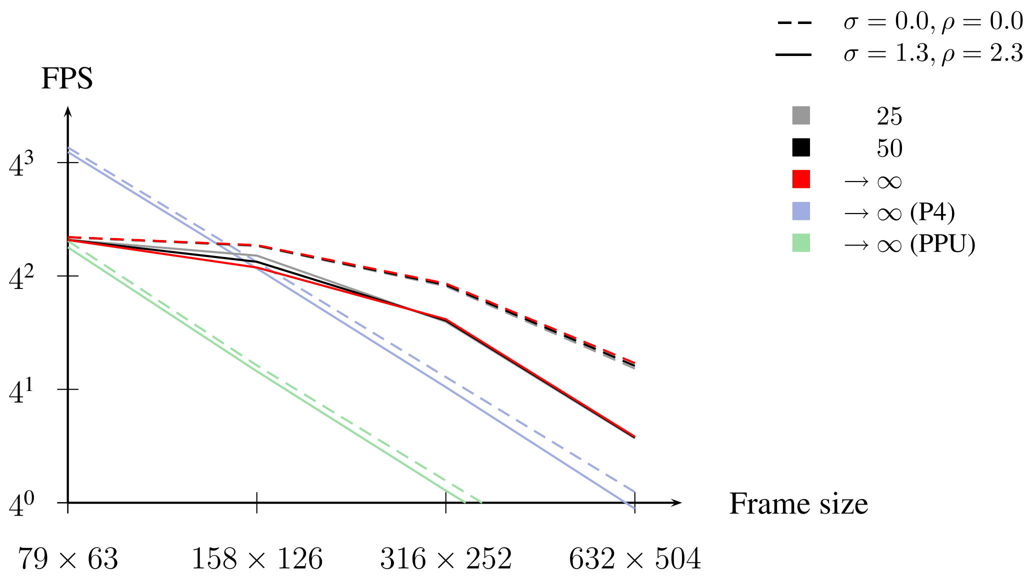

Figure 6.19: Performance (FPS) of 100 Red-Black SOR iteration steps on 6 SPUs with equation system setup over different image sizes. Reference measurements for P4 and PPU are plotted in blue and green, respectively.

again. The reason for the smoothing variant still benefiting from a larger number of SPUs can hence been attributed to the convolutions representing operations with a sufficiently high complexity to absorb the losses suffered in the actual solver step, or in the general framework:

On the one hand, the problem size might still not be huge enough for a spatial decomposition with the Red-Black approach. Since 'red' and 'black' steps are processed independently, a data set corresponding to only half a stripe is effectively processed in between each pair of synchronization messages. For frames sized 316×252 pixels consisting of 79 632 pixels, such entities are hence not larger than about 6600 pixels, and since the Red-Black solver is in addition entirely processed in full SIMD width, four pixels can always be processed in parallel. With respect to the runtime of such stripe, sequential operations on about 1700 pixels are hence a reasonable comparison. However, this value is already in the same range with the Block SOR tile area, which has been chosen to 1024 pixels (cf. Section 6.4). A lower synchronization overhead can hence not necessarily be taken for granted.

On the other hand, copying SPU kernels to their destinations and dispatching them can be quite expensive. Since this process is issued four times per run, the overhead introduced by these reinitializations represents a comprehensible explanation as well. To estimate the impact of this option, the kernels are combined in the course of the next section (cf. Section 6.6), where this setting is then also re-evaluated with respect to the scaling over SPU cores.

Finally, because data requests issued on Element Interconnect Bus and Memory Interface controller become more dense the more SPU kernels are running in parallel, a physical limit might already be exhausted at some points. Even though the general amount of data moved is rather low, the almost perfect inter-core synchronization in this setting causes similar memory requests to be issued simultaneously. The associated bursty traffic can indeed cause the MIC to become a bottleneck.

Performance on Different Iteration Counts

Since the actual solver has been formulated in 128 bits of width, every single step should be noticeably cheaper than on a sequential architecture. For iterations ranging from 1 to 200, the relative loss in the framerate associated to an increasing iteration count should hence read lower. Additionally, the value measured for one iteration step should be noticeably higher than for Block SOR, since the constant expenses for reordering should be more than compensated by the parallelized equation system setup phase.

Table 6.9: Performance (FPS) of 100 Red-Black SOR iteration steps on different SPU counts with equation system setup over images of size 316×252 pixels. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
1 SPU	6.18	6.22	6.26	2.58	2.59	2.59
2 SPUs	10.87	10.99	11.12	4.89	4.92	4.95
3 SPUs	13.80	14.01	14.23	6.75	6.81	6.87
4 SPUs	15.39	15.61	15.83	8.28	8.33	8.37
5 SPUs	15.34	15.18	15.03	9.00	9.02	9.05
6 SPUs	14.12	14.35	14.59	9.14	9.28	9.42

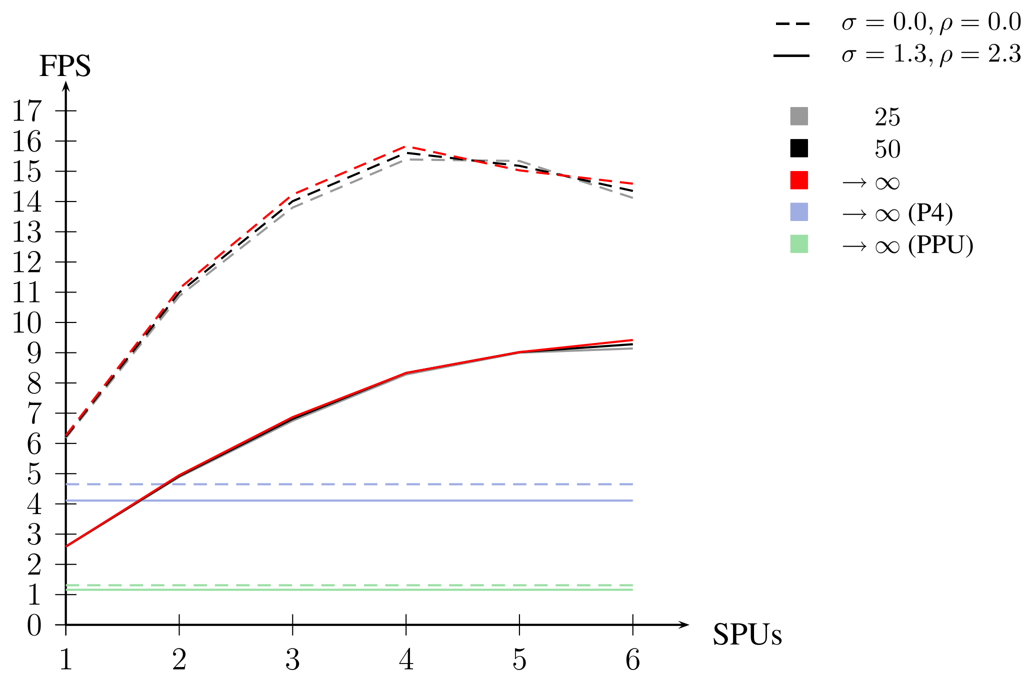

Figure 6.20: Performance (FPS) of 100 Red-Black SOR iteration steps on different SPU counts with equation system setup over images of size 316×252 pixels. The reference benchmarks on P4 and PPU are again plotted in blue and green, respectively.

Table 6.10: Performance of different Red-Black SOR iteration steps on 6 SPUs with equation system setup over images of size 316×252 pixels. Columns denote test intervals averaged over. Rows display the time per frame, the time per iterations, the frames per second (FPS), and the iterations per second (IPS), respectively, for 1, 100 and 200 iterations, each.

		No Smoothing			**Smoothing**		
	it	**25**	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
	1	36.10	34.88	33.65	74.30	73.03	71.76
t [ms]	100	70.82	69.69	68.55	109.38	107.80	106.21
	200	106.49	105.05	103.62	143.64	142.11	140.59
	1	36.10	34.88	33.65	74.30	73.03	71.76
$^{t}/_{\text{it}}$ [ms]	100	0.71	0.70	0.69	1.09	1.08	1.06
	200	0.53	0.53	0.52	0.72	0.71	0.70
	1	27.70	28.67	29.71	13.46	13.69	13.94
FPS [s^{-1}]	100	14.12	14.35	14.59	9.14	9.28	9.42
	200	9.39	9.52	9.65	6.96	7.04	7.11
	1	27.70	28.67	29.71	13.46	13.69	13.94
IPS [s^{-1}]	100	1411.96	1434.99	1458.79	914.22	927.69	941.56
	200	1878.11	1903.81	1930.22	1392.40	1407.32	1422.55

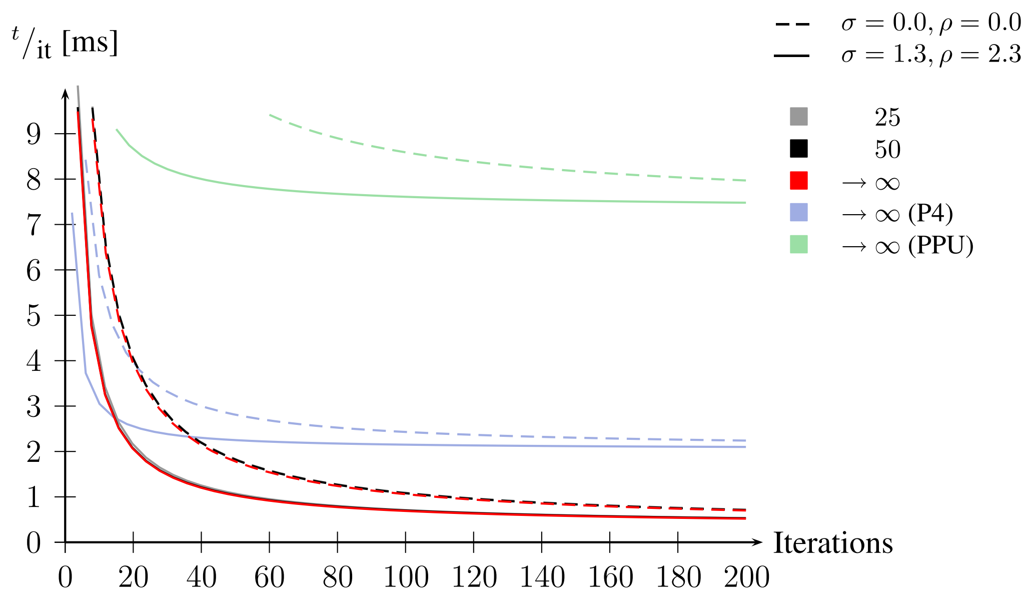

Figure 6.21: Time per iteration step for the Red-Black solver on 6 SPUs with equation system setup over images of size 316×252 pixels. P4 and PPU reference values are plotted in light blue and green, respectively.

Indeed, Table 6.10 and Figure 6.21 report the performance for one single iteration step to be below the Pentium 4 reference benchmark, but this relationship already inverts at about 20 iterations. With 100 steps, Red-Black SOR is already about 2.3 times faster than the SOR solver on the PC platform, and is the same factor ahead of Block SOR, which lies level with the Pentium 4 implementation in these ranges. Here, a massive constant overhead in time can be observed, which expresses in a slower convergence to a minimal time per iteration step. This fact can be attributed to the equation system setup and presmoothing, as well as with a certain amount of time the SPU kernels take to be transferred to their destination.

6.6 Red-Black SOR with Joint Kernels

The modular SOR solver presented in the last section already grants insights to the real performance of the Cell Broadband Engine. However, several issues diminishing the performance still remain. The most evident consequence of these problems is the peak performance for 316×252 pixel large frames being achieved with four, instead of six SPUs. To allow qualified guesses about the causes, the modular design of the SPU kernels is first to be disestablished in favour of one large kernel per SPU, which is then being kept alive for the entire test sequence and over several frames. This way, the latency introduced by loading SPU kernel binaries on the six cores can entirely be disregarded, which should increase the performance measurably. If significant performance losses were still visible then, they could definitely be assigned to a bad ratio of the scheduling overhead and the problem size.

Since the major part of the process has been described before (cf. Section 6.5), only the necessary changes are discussed at this point.

6.6.1 Reordering

Due to their little overlapping, the two reordering steps have already been developed in a separated manner, which allows to simply copy them to the respective positions of the new kernel. However, they have so long been designed to work on double-lines, instead of stripes. Thereby, one core processed every sixth pair of lines, which has been feasible so far, because no synchronization was needed during this process anyway.

Anyway, in this new setting it is beneficial to unify areas processed by one core, since it saves one synchronization step in between the former kernel transitions. For instance, an SPU finishing to reorder one stripe can immediately start solving it this way, without synchronizing with the PPU again. Because both solver and equation system setup work best on a stripe layout, the support of the reordering steps has been changed to stripes as well.

6.6.2 Synchronization

Leaving SPU kernels alive for more than one frame involves introducing a merged bus protocol. After the validity of a new task description is declared to the SPUs sending a `SIG_TASK` notification, they start computing the equation system coefficients in a distributed manner.

Similar to before (cf. Paragraph 6.5.7), four points in time need to be protected by full synchronization steps, i.e. the PPU waits for all SPUs to return `STAT_RDY`, before it releases the lock by broadcasting `SIG_CLR`. Again, these critical points are in between the separated convolutions of the input frames in x and y direction, immediately thereafter, just before the convolutions of the motion tensor entries are performed, and finally in between the two different directional convolution processes.

The first reordering step can immediately pitch in as soon as the equation system is entirely set up for the corresponding stripe, and it can also immediately lead over to the solver stage, because no global dependencies are to be fulfilled in any of these cases. In there, the token based method with one-sided race conditions proposed in the previous section (cf. Paragraph 6.5.7) is pursued. As soon as the last solver step is completed, each SPU immediately starts the final reordering process, and since the supports of all former kernel partitions has been unified beforehand, no additional synchronization step is necessary at the transitions again. However, to inform the PPU about when the data computed by the SPUs is valid, each process sends one terminal ready message back immediately before going idle.

Once initialized, this sequence of messages may be exchanged arbitrarily often. As soon as a kernel termination is required, however, the PPU just needs to send a `SIG_REL` token instead of a new task notification, and the SPUs terminate immediately.

6.6.3 Benchmarks

In the following, the benchmarks performed in Paragraph 6.5.8 are repeated in the new setup and compared to the previously achieved results.

Scaling with the Number of SPUs

Since the performance leak in the last method primarily leapt to the eye when observing different numbers of SPUs charged with the program, this experiment is of course the most interesting in this context. Compensating for one main issue, the results are not only expected to exceed the previous results, but is also meant to show a strictly monotonically increasing number of frames being processed per second, when more and more SPUs are added to the setting.

Table 6.11 and Figure 6.22 indeed confirm these anticipations. Surprisingly, the variant without smoothing performs even up to 1.6 times better than the previous attempt, which finally accumulates to a total speedup of a factor of about 5.2 compared to the Pentium 4 based SOR method. With a peak performance of 24.60 FPS, this implementation can be called a realtime method.

More importantly however, the runtime actually benefits from any of the six SPUs, even though the algorithm can still be observed to converge to some constant value. Since this can have various reasons, the next benchmarks will provide some clarity.

Observing the plots of this experiment, a slight anomaly can be noted when exactly four SPUs are processing the problem. Analyzing this issue, no convincing argument could be found so far. Most likely, the program by coincidence runs into a bus contemption at some

Table 6.11: Performance (FPS) of 100 Red-Black SOR iteration steps using joint kernels on different SPU counts with equation system setup over images of size 316×252 pixels. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
1 SPU	6.24	6.28	6.32	2.52	2.52	2.53
2 SPUs	11.79	11.95	12.12	4.73	4.74	4.76
3 SPUs	16.78	17.20	17.63	6.73	6.75	6.77
4 SPUs	19.31	19.79	20.29	8.24	8.34	8.44
5 SPUs	22.64	23.30	23.99	9.73	9.89	10.06
6 SPUs	23.03	23.79	24.60	10.40	10.59	10.78

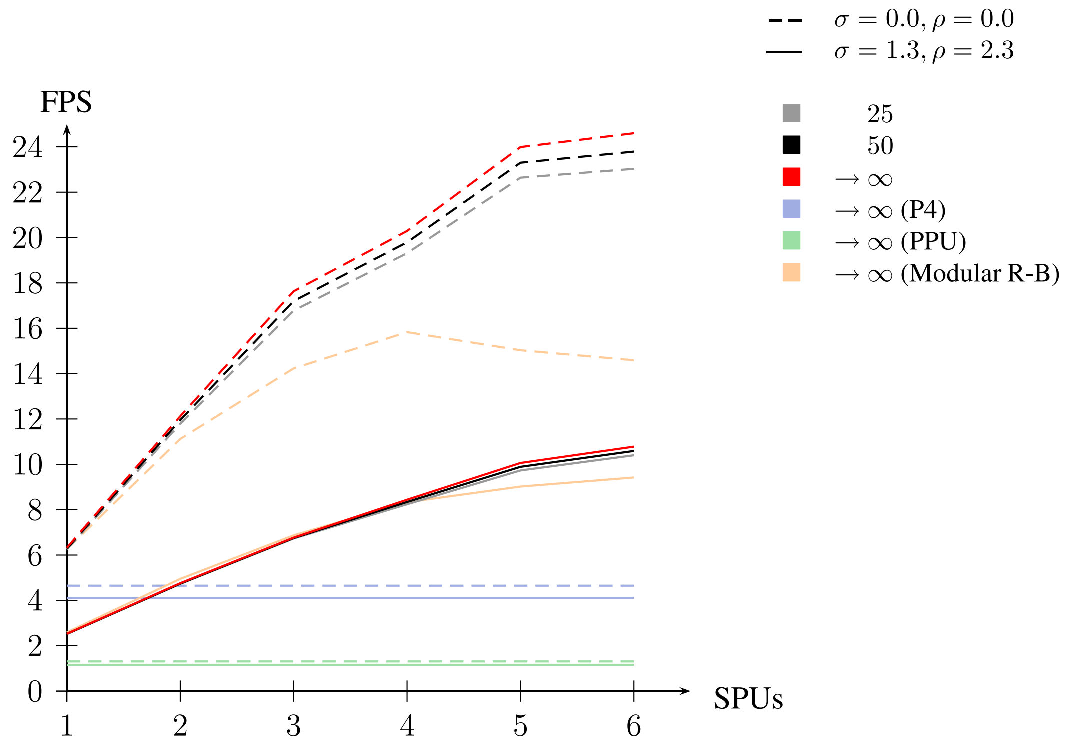

Figure 6.22: Performance (FPS) of 100 Red-Black SOR iteration steps using joint kernels on different SPU counts with equation system setup over images of size 316×252 pixels. The reference benchmarks on P4 and PPU are again plotted in blue and green, respectively, and the modular Red-Black result is drawn in light orange.

point, which introduces a certain latency responsible for the loss of time. Noticeably though, this problem can be reproduced for varying test intervals and is hence not related to measurement outliers.

Performance on Different Image Sizes

One interpretation of this problem proposed earlier is an introduced overhead by means of synchronization messages. In this case, the method would benefit from larger image sizes, while smaller images should perform inordinately worse.

Astonishingly, Table 6.12 and Figure 6.23 even prove the opposite. While the method behaves comparably for the two larger formats, it is superior to the modular Red-Black SOR approach when really small images are considered.

This observation reveals one major insight: Seemingly, the attenuation for higher SPU numbers detected with both Red-Black SOR variants is predominantly related to bus problems, or bottlenecks at the Memory Interface Controller. One logical consequence for future implementations is hence to desynchronize the single SPU kernels best possible while keeping the result stable, which is of course hardly feasible without reconsidering the main concept.

Performance on Different Iteration Counts

To gain insights about the behaviour under different iteration numbers, the method has been probed for 1, 100 and 200 iterations, like this has already been done before for the modular Red-Black SOR approach.

Altogether, the joint approach performs noticeably better, as Table 6.13 and Figure 6.24 show, even though the advantage only marginally pays off for the smoothing variant. Especially for few iterations and no smoothing however, the program runs way faster than previously presented methods, and does in particular converge a lot faster towards a constant time needed to perform one iteration. Both observations can best be explained with constant expenses spent at each run, which are composed of equation system setup phase, and optional presmoothing. Can this operation be accelerated, for instance by disregarding expensive convolutions, the dominant operation is represented by the actual solver, which scales brilliantly. However, as soon as too much time is spent in the beginning, different solver runtimes play only subordinated roles, which is expressed in generally lower frame rates and a slower declining thereof, taking different iterations into account.

Anyway, in the experiments performed with the Red-Black solver with joint kernels, another interesting observation can be made. Since the SPU kernels are started once per test sequence, the cost for loading the binaries via the EIB onto the SPUs and dispatching them there is a constant over the whole duration of the experiment. The longer the test interval is chosen, the more converges the accumulated impact per frame towards zero. This is expressed in the strikingly different measurements for 25 and 50 frames, as well as the computed difference. Going back to the modular Red-Black measurements (cf. Paragraph 6.5.8), this difference can already be spotted, but it is less pronounced than here, since constant expenses

Table 6.12: Performance (FPS) of 100 Red-Black SOR iteration steps with joint kernels on 6 SPUs with equation system setup over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
79 $\times$ **63**	146.82	153.67	161.20	121.01	124.92	129.09
158 $\times$ **126**	87.05	87.26	87.46	50.29	49.68	49.09
316 $\times$ **252**	23.03	23.79	24.60	10.40	10.59	10.78
632 $\times$ **504**	5.69	5.85	6.02	1.87	1.89	1.90

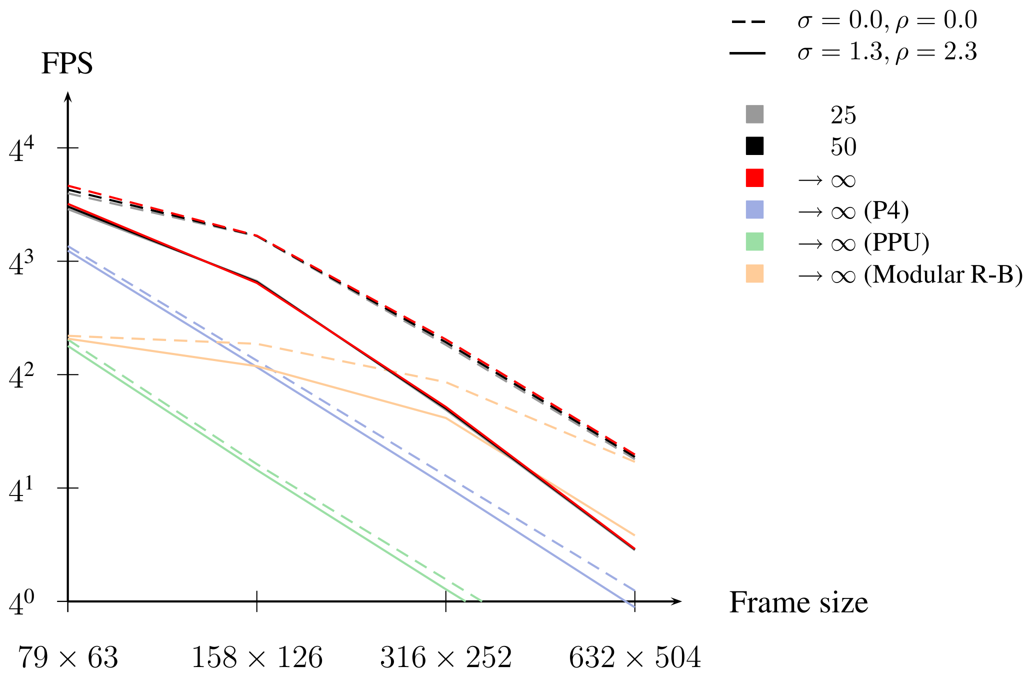

Figure 6.23: Performance (FPS) of 100 Red-Black SOR iteration steps with joint kernels on 6 SPUs with equation system setup over different image sizes. Reference measurements for P4 and PPU are plotted in blue and green, respectively, and the modular Red-Black result is drawn in light orange.

Table 6.13: Performance of different Red-Black SOR iteration steps with joint kernels on 6 SPUs with equation system setup over images of size 316×252 pixels. Columns denote test intervals averaged over. Rows display the time per frame, the time per iterations, the frames per second (FPS), and the iterations per second (IPS), respectively, for 1, 100 and 200 iterations, each.

		No Smoothing			**Smoothing**		
	it	**25**	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
	1	7.43	5.86	4.30	59.67	58.46	57.26
t [ms]	100	43.42	42.03	40.64	96.15	94.44	92.74
	200	80.24	78.29	76.34	132.42	130.80	129.19
	1	7.43	5.86	4.30	59.67	58.46	57.26
$^t/_{\text{it}}$ [ms]	100	0.43	0.42	0.41	0.96	0.94	0.93
	200	0.40	0.39	0.38	0.66	0.65	0.65
	1	134.64	170.53	232.50	16.76	17.10	17.47
FPS [s^{-1}]	100	23.03	23.79	24.60	10.40	10.59	10.78
	200	12.46	12.77	13.10	7.55	7.65	7.74
	1	134.64	170.53	232.50	16.76	17.10	17.47
IPS [s^{-1}]	100	2303.30	2379.24	2460.37	1040.02	1058.83	1078.33
	200	2492.54	2554.67	2619.98	1510.37	1529.03	1548.16

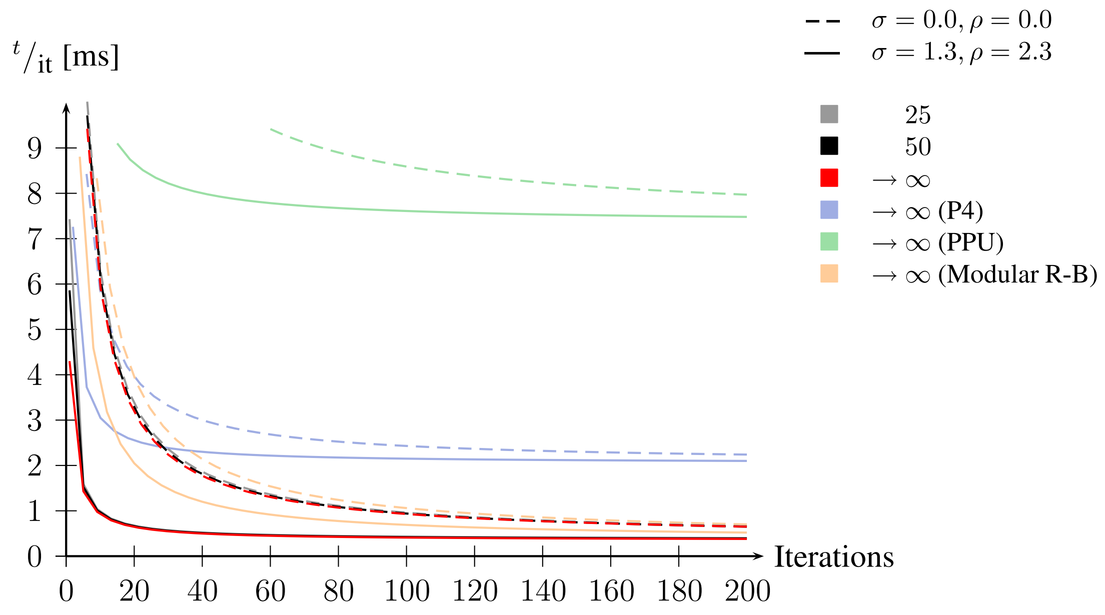

Figure 6.24: Time per iteration for different Red-Black SOR iteration steps with joint kernels on 6 SPUs with equation system setup over images of size 316×252 pixels. P4 and PPU reference values are plotted in light blue and green, respectively, and the modular Red-Black result is drawn light orange.

actually came down to simple memory allocations from the PPU, and these operations seem to be quite cheap.

6.6.4 Discussion

In course of the Red-Black SOR approaches, the traditional concept of the SOR solver has been reinterpreted to offer the opportunity for an efficient parallelization attempt. However, these experiments reveal several drawbacks related to the algorithm, but also to hardware restrictions associated to the alienation of the Playstation 3 video console as a device for scientific computing.

One insight these experiments reveal is that kernels should at best never be reloaded to the SPUs during the course of a program, but monolithic SPU kernels should be designed, being able to switch between different task modes, if necessary. Furthermore, when working on huge amounts of data, these kernels should at best be run asynchronously to reduce bursty access to both ring bus, and the Memory Flow Controller.

From an algorithmic side, several improvements can still be made, especially regarding the formulation of DMA operations. Since these instructions often interleave arithmetics, entire memory management need to be spread over the whole code, being charged with a sufficiently precise prediction of the next vector needed. As soon as such prediction is adequate, waiting times can be reduced to a minimum. In the next chapter, flexible memory management routines will hence be introduced overcoming these problems best possible.

Nevertheless, the results achieved so far are already remarkable, comparing them to the Pentium 4 implementation. With a peak performance gain of a factor of 5.2 finally yielding 24.6 FPS, these approaches are indeed worthwhile for realtime applications. Regarding the Red-Black performance achieved on one single SPU closer, SIMD-based instruction level parallelism can be identified to only compensate for architecture-specific issues, while the overall performance gain of the method can chiefly be attributed to the parallelism due to multiple cores.

Chapter 7

Full Multigrid

7.1 Theory

7.1.1 Multigrid Methods

Classical iterative solvers like the Gauss-Seidel or SOR methods presented in Paragraphs 6.1.4 and 6.1.5 suffer from one important drawback. Because they attempt to improve the quality of a solution by reducing a local error measure, i.e. by improving the solution at some point only by evaluating the accordance within the immediate neighborhood, the global error function consisting of deviations between the current and the optimal solution is more and more spatially smoothed. While high frequencies within this function are hence immediately eliminated, low-frequent errors require many iteration steps until they finally vanish [65].

Multigrid methods suggest a sophisticated remedy to this problem, by compensating for the error term on coarser grids [10, 64]: To the beginning, few iterations are performed with a classical solver for the equation system $A\,x = b$, which leads to a preliminary solution x'. This stage is also referred to as the pre-smoothing phase.

Since the iterative solver has not converged to the solution x yet, there is still a nontrivial low frequent error term $e = x - x'$ present. Even though e cannot be explicitly computed, it can be formulated as the solution of a new equation system

$$A\,e = r,$$

whereof A is the known system matrix, and the residual r can be immediately computed as

$$\begin{aligned} r &= A\,e \\ &= A\,(x - x') \\ &= A\,x - A\,x' \\ &= b - A\,x'. \end{aligned}$$

The error term e is known to be low frequent, and this allows it to be computed on a coarser grid: e is hence given by a prolongation $e = P\tilde{e}$ of the solution $\tilde{e}$ of the equation system

$$RA\,\tilde{e} = Rr,$$

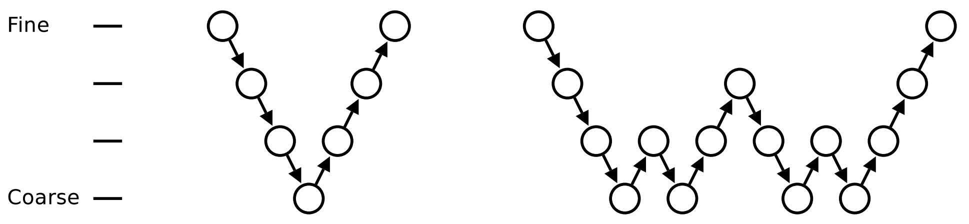

Figure 7.1: Schematic processing order in V (left) and W (right) cycles within a Multigrid setting.

where R and P are the restriction and prolongation parameters, respectively. Finally, the preliminary solution x' can be corrected by e to yield $x = x' + e$, and several iteration steps of the classical solver are applied to eliminate potential high frequent errors introduced by the previous correction step. In contrast to pre-smoothing, these operations are called the post-smoothing steps.

Since the equation on the coarser grid can again be solved by this method, a simple recursion rule is obtained: An equation is first processed by some pre-smoothing steps, the residual is then being downsampled, and the algorithm recurses. On return, the error term is upsampled, added to the intermediate solution, and finally some post-smoothing steps are performed. Since the visualization of the algorithm over different grid levels (cf. Figure 7.1) looks like the letter V, this type of recursion is referred to as V cycles.

To enhance the quality of the solution, the underlying recursion rule can be slightly modified. Instead of recurring once per level, a second recursion is worthwhile, since the error term can be expected to be way smaller in the second call and the yielded solution should hence be more precise. This recursion strategy is referred to as W cycles (cf. Figure 7.1), and is also well known in literature. Whereever it is not explicitly stated, this variant is used throughout the experiments issued in this thesis.

7.1.2 Full Multigrid Methods

Another intuitive approach related to different grid sizes is to pursue a simple coarse-to-fine strategy: The problem is therefore first downsampled to the coarsest grid and solved with any classical method, which is fast due to the small image size. The solution can then be upsampled to the next finer grid, where it is used as an initialization for the solver at this stage. Similar to the Multigrid method described above, low frequent errors are largely eliminated, such that the solver at the current stage is meant to converge very fast. This procedure can then be repeated until the problem is entirely solved on the finest level.

Combining both coarse-to-fine strategy and the Multigrid method, benefits from both algorithms combine to a very powerful solver, the so-called *Full Multigrid method*. Here, a standard Multigrid solver is applied at every level of the coarse-to-fine strategy, and its solution is used as an initialization for the next finer level. This setting is quite cheap compared to the quality achieved: Typically, only one relaxation step is needed before and after any recursive call to yield reliable results, which allows fairly high frame rates when being applied to

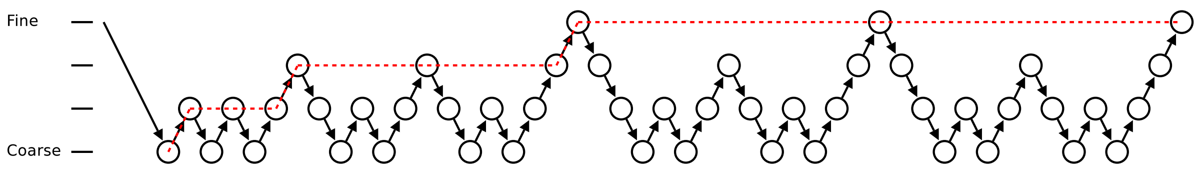

Figure 7.2: Schematic processing order within a Full Multigrid setting.

the Optical Flow problem.

Figure 7.2 shows the schematic view of such process. Like before, circles determine smoothers, while arrows describe downsampling steps of residuals, or upsampling of intermediate solutions. The course of the standard coarse-to-fine strategy is charted in red to emphasize the grid transitions related to this idea, where the solution of the previous step is actually upsampled and used as an initialization for future steps.

7.2 Reference Implementation on Pentium 4

Like for SOR, reference benchmarks performed with Bruhn and Weickert's sequential implementation [14] on the Pentium 4 are presented first to be able to judge the quality of the parallel implementation developed later in this chapter. Again, all settings are evaluated with respect to a real time measurement obtained by the C `gettimeofday` routine.

7.2.1 Impact of Different Image Sizes

In this experiment, frames 8 and 9 of the Yosemite sequence have been rescaled by factors of 0.25, 0.5, and 2 with respect to the edge lengths. Optionally, the motion tensor and the initial images have been blurred by a convolution with Gaussian kernels. As it has already been proposed for the SOR case, the variances have been chosen to yield comparable appearance, i.e. the values $\rho = 2.3$ and $\sigma = 1.3$ for the motion tensors and the frames, respectively, have been scaled by the same values as the edges of the respective frame they are applied to. Like before, the measurements rely on ten independent measurements for each value, and the test intervals have been chosen to 25 and 50 runs, as well as being subtracted to yield the theoretical performance over larger test intervals.

Figure 7.3 and Table 7.1 depict the results of these measurements. Unlike for SOR, the results achieved for Full Multigrid are significantly higher, but presmoothing is more expensive in this case, too.

7.3 Performance on the PPU

In the following, the benchmarks from the previous section are repeated on the PPU to be later consulted as a second reference value besides the original implementation on the Pentium 4. Like for SOR, the PPU related reference benchmarks are again drawn in green.

Table 7.1: Performance (FPS) of a Full Multigrid approach on the Pentium 4 3.2 GHz over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
79 × 63	331.80	325.18	318.83	259.68	258.60	257.53
158 × 126	118.65	115.32	112.18	77.48	75.53	73.68
316 × 252	33.74	32.68	31.69	17.20	16.81	16.45
632 × 504	8.37	8.42	8.48	3.22	3.21	3.21

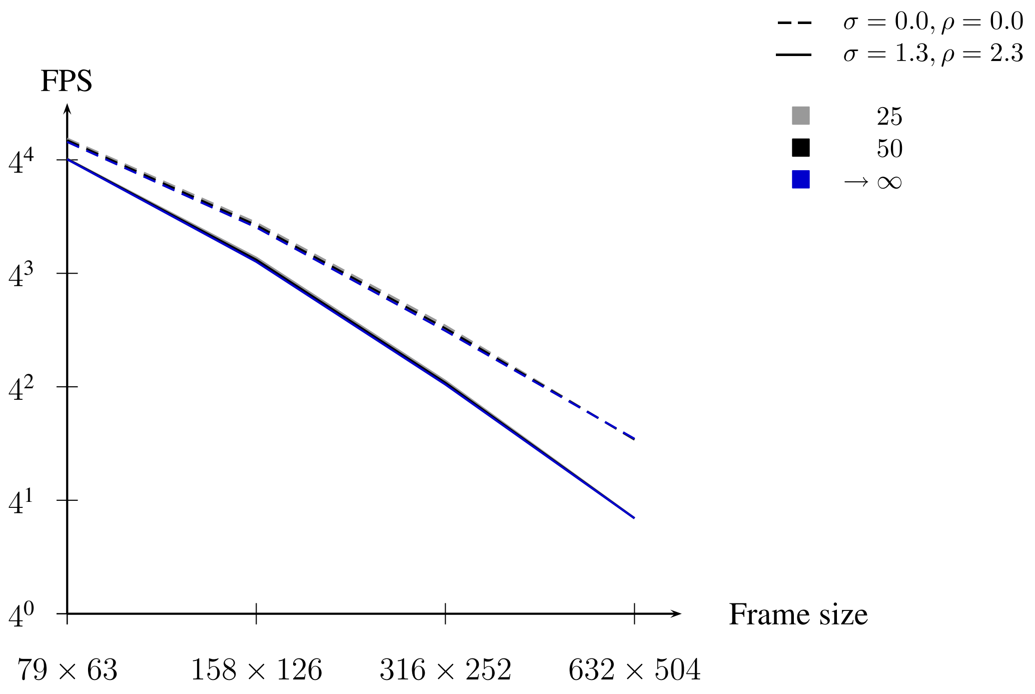

Figure 7.3: Performance (FPS) of a Full Multigrid approach on the Pentium 4 3.2 GHz over different image sizes.

7.3.1 Impact of Different Image Sizes

Here, the resampled versions of frames 8 and 9 of the Yosemite sequence are again evaluated by a Full Multigrid method. This test is entirely comparable to the respective benchmark on the Pentium 4 (cf. Paragraph 7.2.1).

Figure 7.4 and Table 7.2 show the results of this benchmark. Compared to the Pentium 4 implementation, the performance is still way lower. However, like in the SOR setting, the general appearance of the graph is still maintained.

Table 7.2: Performance (FPS) of a Full Multigrid approach on the PPU over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
79 × **63**	89.30	87.90	86.54	73.83	72.99	72.17
158 × **126**	32.63	33.16	33.72	22.56	22.72	22.89
316 × **252**	8.87	8.92	8.97	4.91	4.91	4.92
632 × **504**	1.72	1.77	1.83	0.82	0.85	0.88

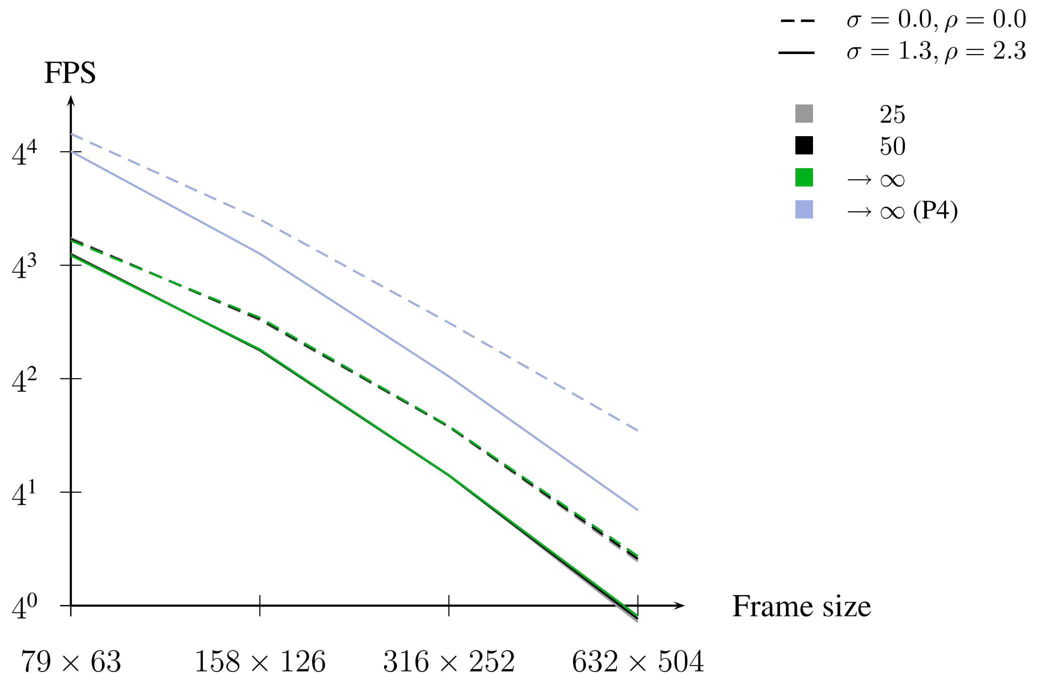

Figure 7.4: Performance (FPS) of a Full Multigrid approach on the PPU over different image sizes. The P4 reference graphs are depicted in light blue.

7.4 Towards a Parallel Variant

As previous implementations have shown, distributed problems often suffer from their synchronized SPU kernels, which is indeed a problem as soon as significant amounts of data is exchanged at once using DMA calls. Unfortunately, since operations with non-local support are involved, the synchronization can hardly be abolished completely for any of those programs. As soon as such manual synchronization is omitted, the system usually begins to self-regulate to an optimal scheduling, since the SPUs wait differently long for the first few DMA requests to be answered. This shifts their time windows slightly offset to each other, and since the kernels perform the same operations and equal access patterns reoccur quite frequently, these requests are likely to hit an idle interval at the next calls.

On the way to a multigrid approach, over-synchronization turns out to become even more severe: The main structure is no longer given by an iteration, but by multiple grid transitions, whereby resampling and Gauss-Seidel relaxation steps balance each other. Particularly the resampling steps generate a noticeable traffic on the MIC, and is all SPUs are in addition perfectly synchronized, bus contemptions and resulting stalls would be inevitable.

In a stripe-based decomposition, this problem would even be amplified: Here, not only the time spent on waiting for neighboring SPUs to finish their work would shorten the actual performance. By being dispatched synchronously again, little time remains for the system to self-regulate. But stripes also reveal another drawback: Since the stripe structure was to be propagated over all pyramid levels, the actual size of the single representations would significantly differ from each other. Especially on the coarsest grid, each SPU only had a negligible amount of data to process, which is way below the amount paying off in context of interleaved DMA operations. In particular, since DMA instructions suffer from a certain memory latency, which can be considered constant and thus independent from the vector lengths requested.

In contrast, the quite low number of available SPUs on the Cell Broadband Engine offers a convenient opportunity to bypass these problems in a convenient manner. Assuming a setting with sufficiently many consecutive frames to compute, it is considerable to charge each SPU with exactly one pair of frames, and to implement an efficient sequential solver on each of these SPUs. This approach is a special case of the *Computational acceleration model* presented by Kahle *et al.*, whereby the deployment style of SPU kernels matches rather the *Function offload model*, since one frame is always computed by one dedicated SPU [40].

This temporally decomposed implementation is expected to be faster than spatially distributed versions: In a theoretical scenario, six pairs of frames would at once be sent to the six SPUs. Thereby, the achieved frame rate is expected to benefit at least as much as in the distributed setting, since all cores are fully exhausted and the algorithm does not suffer from slow DMA operations and synchronization latencies. Unlike before however, the latency between the computation request and the final result cannot be reduced at all by using this technique, which might influence the applicability for certain real-world applications.

7.5 Local Store Based Variant

In a first implementation, Bruhn and Weickerts sequential solver has been ported to the SPU, computing the entire problem inside the cache. This approach basically comes down to an explicit modelling of the interface between PPU program and SPU kernel, but can also not be expected to be capable of reasonable image sizes. Nevertheless, it forms a good base for further development, since the program can be generalized in a stepwise manner and be extended by architectural optimizations.

To the beginning, a problem record is loaded from RAM, which includes state information such as process parameters, information about the input frames, and pointers to intermediate memory cells. The address of this record is static on a per-SPU basis and passed as a single pointer at kernel setup. Because this record is filled by the PPU while the SPU kernel is already running, a notification message by the PPU is needed to tell the SPU that the provided data is finally valid. To be later able to load and store data to the respective positions, all vectors in RAM again need to be aligned by at least 128 bits, as this has already be the case for the SOR algorithms.

Based on the information received, the SPU can then start to allocate local variables of appropriate size, which in this particular case just means enough space to contain the whole problem including intermediate results. Since DMA accesses are limited to 16 kB and this limit can indeed be exceeded in this experiment, the following pyramid-wise load instructions are designed to split large requests into such of at most the maximal transfer limit. As soon as the result is located in the respective target vectors, these need to be written back to their corresponding RAM locations again, respecting the same details as for loading.

From this point on, the environmental preconditions are comparable to those on the PPU or the Pentium 4, such that the sequential algorithm known from the PPU runs without any problems. This proves that despite of their missing cache management and of further design peculiarities, the SPUs are still closer to general purpose hardware than probably any other special purpose layout. One should however remark at this point that the actual efficiency in this experiment is rather low, because scalar operations only occupy the foremost slot in a SIMD vector, using no instruction level parallelism at all.

This is unfortunately not the only reason rendering the experiment almost incomparable to usual sequential implementations. Throughout the process, there are larger amounts of data to be worked upon, which causes the maximal image sizes allowed to range in sizes of about 20×16 pixels. By a dynamical allocation of memory on demand, the size could theoretically be extended to at most 60×48, which is already in the region of the smallest frame size examined in context of the SOR solver (cf. for instance Paragraph 6.2.1). However, since this is way below the desired dimensions, no further effort has been invested into the optimization of this particular approach.

Anyway, this first step towards an SPU kernel capable of larger images turned out to be quite helpful during later extension, not least since equation system setup and the actual solver show a minimum of cross-dependencies. This offers the opportunity to preliminary divide the program into two halves of about equal code complexity. An interface synchronizing intermediate results with the RAM at this point hence allows to reimplement both parts separately from each other, and to compare the achieved results with each other to ensure the consistency of both solutions. As soon as both parts are then successfully working, the immediate interface can be removed again and since the cache is no longer hosting the entire data set, the algorithm can be tested on larger frame sizes.

In the following, the generalized variant is described in detail.

7.6 Memory Representation

Since six independently working cores need to be supplied with enough memory space to work with, massive amounts of memory must be allocated in RAM beforehand. Since the memory address of any matrix-valued memory cell is announced in the task descriptions passed to the single SPU kernels at runtime, the actual location can be dynamically adapted to the actual application. In a realtime setting with continuously arriving frames however, ring buffers for the input frames and output flow fields, as well as dedicated areas for intermediate solutions might be the best setup.

In a smaller scale, these matrices are again aligned in a column-before-row manner, like this has already been done in course of the Red-Black setting (cf. Paragraph 6.5.3). Because several differently scaled grids are used and transitions between these grids are frequent throughout the algorithm, reordering schemes are however very expensive and hence infeasible in this case: For any resampling operation, this coding would first need to be reverted, and afterwards re-established again, or some complex algorithm is needed to immediately translate between different representations. Latter algorithm can still be formulated quite easily if for instance the scaling factor equals two. At least for arbitrary scaling factors as they are inevitable if the image boundaries are no power of 2 however, the case distinctions needed would soon annihilate the benefit achieved in the relaxation routines.

The Full Multigrid SPU kernel expects columns to be minimally aligned in RAM, i.e. it reads and writes in 128 bits multiples, but often works on higher alignments internally if this yields an advantage. For instance, loop unrolling schemes processing 512 bits in one iteration can be formulated more sloppy this way, if they are run on a multiple of this length without any special boundary treatment. This additional padding can either be performed at the beginning of a column, at the end, or both. In the current implementation, the boundaries are tied to zero, because this allows for algorithmic simplifications during the solver step, where boundary conditions do not need to be considered explicitly again. To keep this state consistent in sloppy loop unrolling schemes, the boundary pixels can easily be reestablished by one single assignment operation after the processing of each column, which is still a lot cheaper than handling the boundaries as special cases. When the data set is then written back to RAM respecting its actual size, the consistency to the ‘clean’ notation is preserved, since all superfluous values depending on uninitialized values are either belonging to the additional spare values which are not written back at all, or represent bogus bytes used for minimal padding anyway. When describing the basic concepts, these approaches are be discussed more detailed (cf. Section 7.7.9).

Like for Red-Black SOR, maintaining pointer hierarchies over the whole RAM data in cache is not only dispensable, but can also become very expensive in Local Store consumption. Fetching such structure from RAM each time access to a certain column is needed usually also does not pay off, since most applications process a matrix-valued structure from left to right anyway and a recomputation of a pointer can hence be realized by an addition of a column length constant. Different to before however, several pyramid layers need to be addressed this time and since the base pointers to the single levels are nonlinearly distributed, keeping them in fast lookup tables is indeed a good idea. For typical maximal recursion depths of ten levels, this comes down to 40 bytes of storage needed per pyramid, which is well affordable.

7.7 Implementation of the SPU Kernel

In the following paragraphs, the changes applied to transfer the sequential solution proposed by Bruhn and Weickert to the SPU are explained in detail. Besides SIMD related modifications, sophisticated memory management routines are presented, since previous experiments have shown that deliberate considerations on this issue make up a significant part of the resulting performance.

7.7.1 General Structure

When the kernel is booted up, it is initially idle and waits for a new task notification given by `SIG_TASK`, or a shutdown signal `SIG_REL` to arrive (cf. Figure 7.5). A third option is to receive the dummy instruction `SIG_FIN`, which immediately returns with `STAT_RDY`(cf. Figure 7.6). For some applications latter method can indeed be easier to implement on PPU side, since mailboxes can still be read out round robin, even though the computing power of one core was not needed because frames supplied from an external device do for instance not arrive in time.

As soon as a task notification waits in the mailbox and the kernel is idle, it reads out its previously defined and now valid problem record, which contains all necessary parameters and pointers to data pyramids. It then computes the pointers to all pyramid levels, feeds each pointer into the previously mentioned lookup vector, and begins with the equation system setup.

If a presmoothing of one or both input frames is desired, this operation is issued first, whereby the original image is overwritten with the smoothed version. Details about the smoothing routines implemented are discussed in Paragraph 7.7.9.

In the problem definition record, an additional synchronization step can be requested at this point, which consists of a single `STAT_PAU` token sent to the PPU, and a following listening for a `SIG_CLR` token to continue with the computation. This step is not needed if the SPUs process independent pairs of frames, but is worthwhile for a continuous image sequence: Here, every SPU only has to convolve the later frame of the assigned pair, since the earlier one can be taken from the previous SPU in the queue, which has it already smoothed

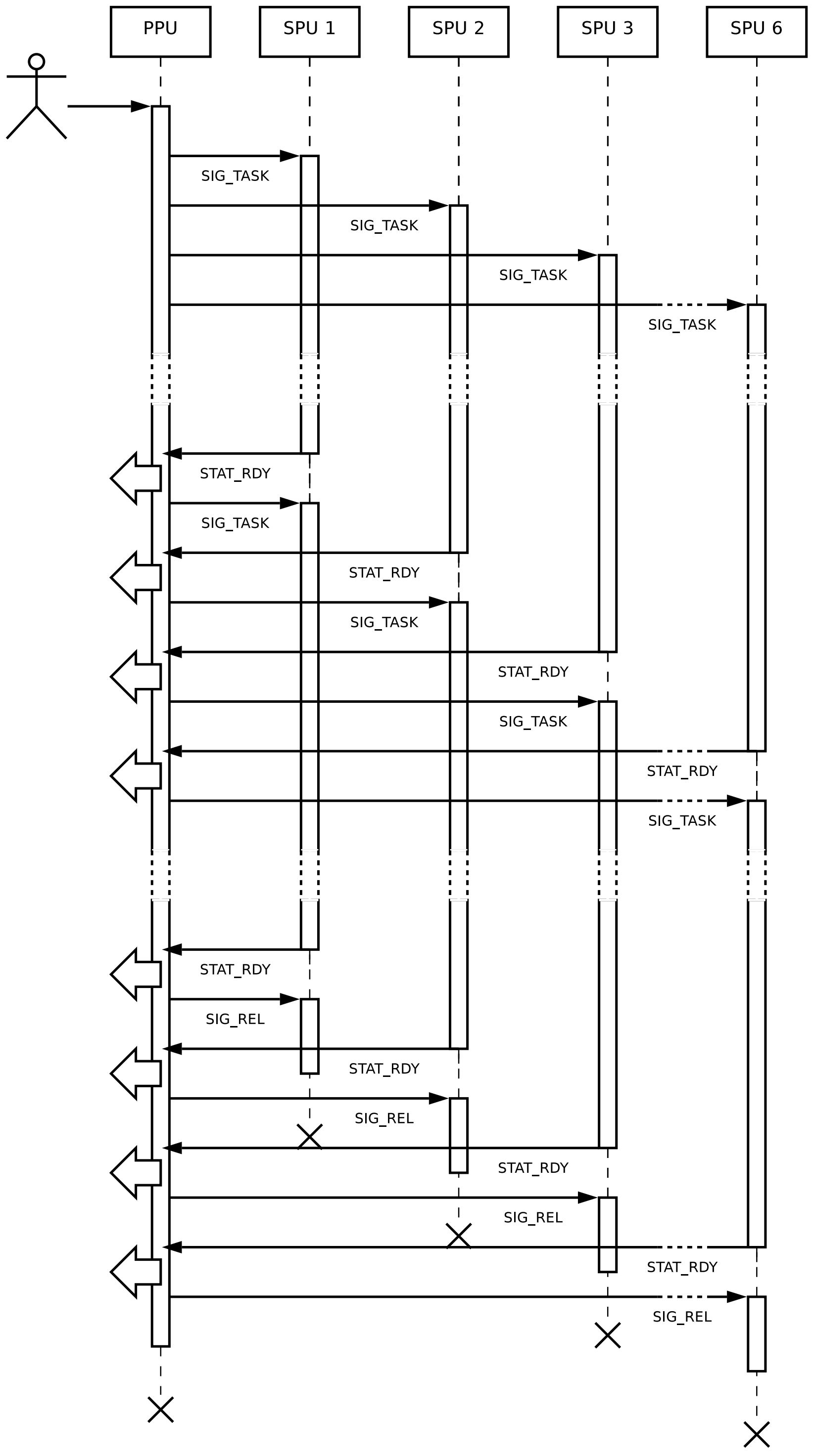

Figure 7.5: Bus protocol for the temporal decomposition of the problem. All SPUs are used.

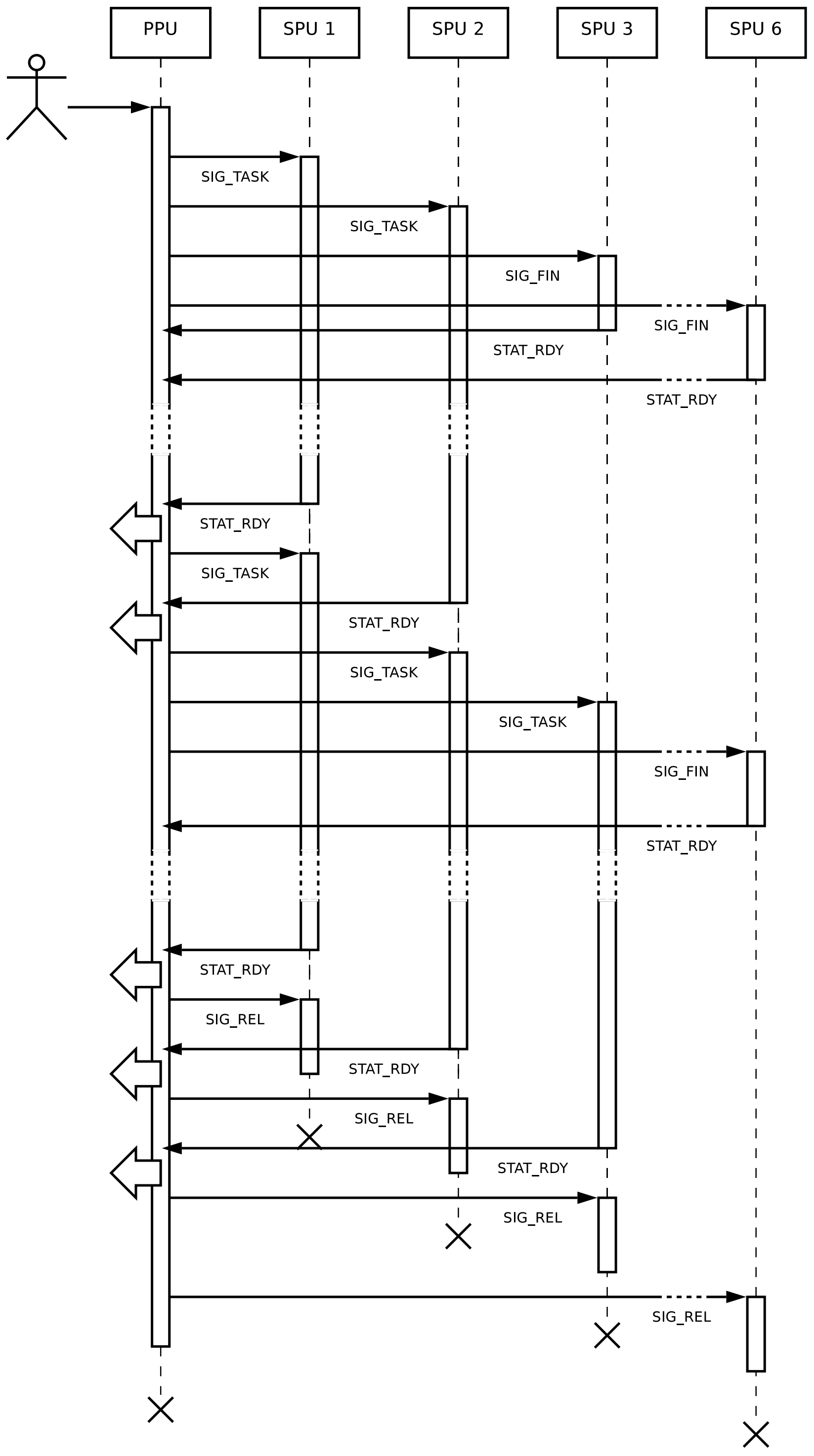

Figure 7.6: Bus protocol for the temporal decomposition of the problem. Here, only two SPUs are fully used, and one is episodically occupied.

in course of its own computations. When the validity of this earlier frame cannot be ensured by other means, it hence needs to be explicitly attested by the charged SPU beforehand.

When both input frames are available in the requested format, spatial and temporal derivatives are computed, and these derivatives are multiplied to yield the motion tensor entries (cf. Paragraph 7.7.7). Optionally, the motion tensor entries are smoothed (cf. Paragraph 7.7.9), and downsampled (cf. Paragraph 7.7.2) on all grids. On each layer, the equation system coefficients are eventually assembled (cf. Paragraph 7.7.8).

The following solver stage then first starts with a recursive descent describing the coarse-to-fine strategy, and upsamples the solution as soon as a full V or W cycle has been computed. One such cycle first starts with a certain number of relaxation steps. However, for the Cell setting, the first relaxation step is handled differently: Since the first relaxation on a newly approached grid should work on a zero-initialization, there is no need for the solution vector to be fetched from RAM. Instead, the memory management routine can be instructed to leave this DMA access out and to return a reinitialized field instead of the fetched solution. Details about this technique are described in Paragraph 7.7.3. Based on the result w' of this step, the residual $r' = A^i w' - b^i$ is computed (cf. Paragraph 7.7.5), and resampled to the next coarser grid, where the V or W cycle is computed recursively. On return, the coarse solution is upsampled, and added to the preliminary solution yielded by the presmoothing relaxation (cf. Paragraph 7.7.6). A number of post-relaxations concludes the cycle computation.

7.7.2 Resampling

The Full Multigrid solver in principle relies on two main components, namely per-level operations, like the essential relaxation steps, and resampling between the pyramid layers. Resizing a signal is hereby a dangerous operation, since erroneous interpolations or restrictions can be misinterpreted as errors, and numerical errors in coarser scale may amplify when going back to finer scales. Such bad results then cause a higher amount of postsmoothing, which significantly reduces the performance gain achievable. On the other hand, resampling algorithms must not become too expensive by themselves, as this entirely deteriorates the benefit achieved by them.

The resampling technique used in this context is a bi-linear integration over areas [14]. This practically means the two involved grids are projected to the same area, and each new cell is assigned the sum of all covered old cells, weighted by the percentage of coverage. Of course there are far more sophisticated interpolation schemes like bi-cubic splines interpolation [53], but since these methods are more expensive to compute and rely on larger neighborhoods, which is in particular problematic on the Cell processor when resampling is performed across the cache direction, they hardly pay of for this approach.

Fortunately, resampling is separable, by using a misproportioned grid as an intermediate solution. Resampling in x and y direction can hence not only be performed one after the other, but the execution order can also be changed, which makes sense from a computational sense: Choosing the preferred direction such that the intermediate grid is the smaller of both possibilities, the time needed per step is minimized. In the following, both directions are discussed separately, since the actual implementations differ due to different cache directions.

Resampling in y Direction

Like already described in Section 7.6, matrix-valued data is allocated in a column-before-row manner, which means that values from one column reside in adjacent memory cells. The resampling process in y direction is local with respect to single columns, which means there are no dependencies between columns. This observation immediately motivates to process several columns in parallel, thereby not only reducing branching, but also allowing a more flexible scheduling with respect to instruction latencies (cf. Paragraph 3.5.4).

Within a column, the instructions are however structurally heterogeneous, because a new cell can either overlap the left boundary of an old cell, its right boundary, both, or none. This case distinction unfortunately renders any instruction level parallelism infeasible, such that operations need to be written in scalar notation, anyway.

The algorithm allocates four two-vector ring buffers for input and output, each, whereby the length is chosen to equal the respective padded column heights. To simplify work on these buffers, pointers to each of the sixteen single vectors are created, while those pointing to the first half are from now on called ‘current’, while the remaining are named ‘spare’.

In a loop over all chunks of four columns, the algorithm first waits for the running read operation to be finished, then swaps equally numbered ‘current’ and ‘spare’ pointers which hence rotates the buffers, and finally starts a new DMA request on the new ‘spare’ input pointers. The ‘current’ pointers however contain valid data again and can hence be processed in parallel.

This fact also explains the main benefit of such transparent memory management functions: The algorithm can be written as if it would work on a static set of variables, independent from which column is currently processed. Whenever a transition between columns becomes necessary, a single call to the memory management routine suffices, and the pointers acting upon are henceforth referencing the new problem. This way, no more cache related instructions need to be interleaved to the code and since the function is designed blocking, data validity is always guaranteed to the calling function.

As soon as all arithmetic operations on this bunch are finished and potentially running writing operations are finalized, the ‘spare’ and ‘current’ output pointers are exchanged to create a fresh dataset for the next operation, and data in the ‘spare’ pointers is synchronized to RAM. Leaving the loop, only the last operation needs to be finalized.

On the right boundary, the algorithm just blocks DMA requests on nonexistent memory cells, since this would introduce inconsistencies. However, arithmetics are still performed on bogus entries, since the cycles needed at this point are wasted by dependency related latencies anyway.

Since all columns containing nontrivial elements are touched during this process, the boundaries are instantly initialized with zero. This needs to be done once anyway, because it makes little sense to fetch output vectors beforehand only to keep correct boundaries in the local copy. Additionally, this step saves a dedicated initialization process at the beginning of the program run, which are very expensive on the Cell architecture: Due to the contrary cache direction, it would need to fetch top and bottom boundary pixels of each affected pyramid on all levels within 128 bit vectors one after the other, just to set one particular value, and then

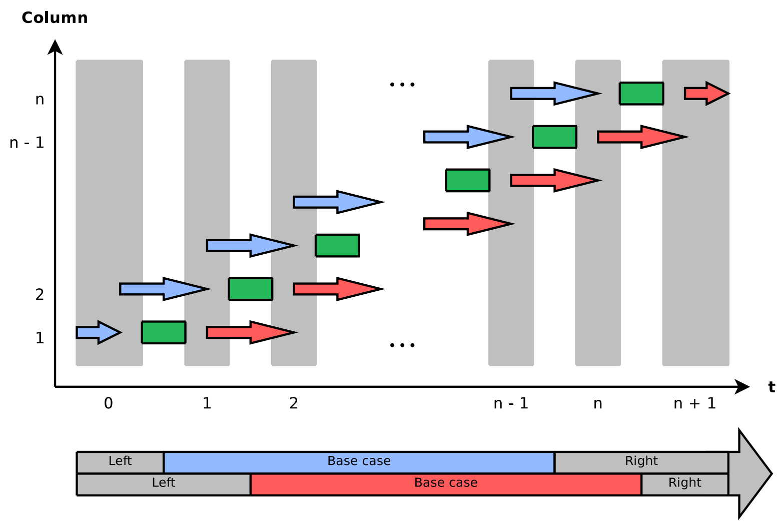

Figure 7.7: Schedule of the memory management routine for resampling in x direction. Here, both reading (blue) and writing (red) case are combined in one scheme.

write them back to RAM. By saving this process, at least two DMA operations per column can be deleted without subsitution.

Resampling in x Direction

In the perpendicular direction, the problem is slightly more complicated. Here, no dependencies exist between the single rows and hence in particular not between entries of a SIMD vector, which allows to conveniently use the full bandwidth for all arithmetic operations, and also offers the opportunity to process whole columns at once. The downside is however that the two ring buffers for input and output need to be advanced differently fast, depending on the resampling factor, since each of both ring buffers follows the spacing of one of the two grids.

For this purpose, a memory management routine for reading and writing access to columns has been designed. It takes pointers to the 'current' and 'spare' components of the working copy, which are passed in a call-by-reference style, a reference to an internal service variable indicating the column last loaded or stored, information about the total number of columns to manage and the length of each column, a switch indicating if reading or writing access is requested, a pointer to the RAM representation, and finally a DMA flag to identify related memory operations (cf. Paragraph 3.5.2). When being called, it ensures the next vector is valid, swaps the pointers, and immediately issues the next DMA operation.

Figure 7.7 shows the schedules for both modes of this routine, whereby reading is displayed in blue and writing is marked red. Active time spans of the routine are visualized by a grey block, and in between, a green bar denotes arithmetic operations on valid 'current' vector, while arrows symbolize pending DMA requests issued on the 'spare' partition.

For n columns to handle, the routine needs to be called exactly $n + 1$ times, independent of whether its mode is set to write or read. To ease this issue on the calling side, the function returns 1, if less than $n + 1$ calls have been made, and 0, if the limit has been reached. It can hence be conveniently inserted as a condition for the loop iterating over columns, such that the stopping criterion does no longer need to be explicitly modelled from the arithmetics side, but is implicitly given by the memory routine.

On the left and right boundaries, special cases need to be considered. In addition to the regular case, the very first reading step needs to execute a whole DMA operation beforehand to provide the first valid data set already when it returns. The first writing access, in contrast, is only a dummy operation immediately returning when having modified the internal column counter, since the first call does not have any data to write, yet. The second call then only starts such RAM interaction, but does not have any call to finish yet.

On the right boundary, this situation can be found in a mirrored sense: The second but last reading call only finalizes a running request, but does not start any new one, while the last call is only a dummy instruction to stay consistent to the writing case. Consequently, the last writing call does not only finalize the pending operation, but also executes a whole interaction to store the very last column to RAM. In Figure 7.7, special and regular cases are visualized in the arrow at the bottom.

This way, the routine needs equally many calls independent from whether it is used for writing or for reading. Both versions hence provide an equal interface in both a structural, as well as logical sense and no special cases need to be explicitly handled on the calling side. So, cache management and arithmetics have again been perfectly separated from each other, which has proven to be both high performant, since the function can be declared `inline`, and easy to write and debug.

Using this routine, the concrete implementation of resampling alongside the x axis is quite easy to design. The iteration over the new grid can just be modelled by a loop coupled to the result of the writing memory management function. Once the reading function is initialized, it can in contrast been called whenever this is required throughout the iteration.

However, different than for the corresponding action in y direction, columns are always processed as an entity. This means, when the overlap fraction representing the weight for the addition is computed, it is duplicated into a all four slots of a SIMD variable. A local iteration with four times 128 bit loop unrolling is then used to add the whole weighted column onto the target vector. Since this rather optimistic approach would run into trouble if the column with is no multiple of 512 bits, the Local Store representations have just been designed sufficiently large, whereby several bytes at the end of each column could then be unmeaningful. However, like it has already been observed for resampling in y direction, this is no problem at all. The four SIMD computations regarded in each step can be reordered such that one operation uses existing latencies of another, and the consistency is not put at risk, since a linear combination on valid values yields the desired result, a linear combination on the zero boundary again results in zeroes, and any operations on bogus values again yield undefined entries.

7.7.3 Relaxation Steps

A second essential component of the Full Multigrid solver is the routine for pre or post-relaxation steps using several pointwise-coupled Gauss-Seidel steps. Like mentioned earlier in this thesis, the ordered layout of the underlying data is herein indeed a problem, since it entirely forbids any form of instruction level parallelism. However this way, management of columns in the Local Store is a little bit easier, because there are no differently 'colored' vectors to be treated separately, but the processing order is quite straightforward.

Anyway, an analyse of the data dependencies reveals demands for more sophisticated scheduling in this setting, however. Two different types of variables can be distinguished. On the one hand, there are linear system coefficients and right hand side entries, of which only one is needed per operation, each. This means, the operation has local support with respect to those values and it is hence enough to hold as many input vectors in cache as lines are processed in parallel. On the other, there are the flow field entries, which serve both as input and output. More importantly however, the reading scope covers three horizontally adjacent cells, which in terms of a column-wise view comes down to at least three columns per operation that need to be held in the Local Store simultaneously.

The first type of variables is modelled as before, by a ring buffer consisting of two vectors of which one is always valid for computations, while the remaining one is meanwhile being filled with the next vector. The second type is designed as a four-vector ring buffer, consisting of three 'current', as well as one 'spare' element. Within the three valid columns, the one in the middle is currently processed. Its left neighbor is both continuously read as input for this computation, and simultaneously written back to RAM. This is feasible, since these two operations are non-conflicting and the values of this row do not change anymore. The right neighbor in contrast is only used as an input to the computation and will become the new computation target when the buffer has been rotated once further. Like before, the 'spare' element is designed to do backgrounded loads of the next column. Since pointers to the respective column bases are passed in a call-by-reference manner, arithmetics can always rely on valid data residing in the 'current' vectors.

Figure 7.8 shows the scheduling for this function, splitted into the parts for single and the triple vectors. Like for resampling, grey blocks specify the time this function has the focus, blue again denotes backgrounded loads, green symbolizes valid data currently processed both reading and writing, and red denotes dedicated store operations. Differently to before, yellow has been introduced to denote read-only values residing in cache, the white arrows mean the whole boundary is initialized with zero values instead of issuing a data load. Latter is performed by an iteration processing one SIMD vector per step.

The dashed arrows are unmeaningful during a normal run, since boundary values are not touched at all and hence do not need to be stored to RAM. Optionally, a flag can however be passed to the routine detailing that the initial value of the flow field should be assumed to equal zero, instead of loading the actual contents from RAM. This is helpful to save dedicated initialization steps at each descending edge in the schematic Full Multigrid graph (cf. Figure 7.2). As soon as this flag is set, all blue arrows in the bottom graph need to be replaced by zero-initializations, and the dashed arrows at the left and right boundary are now actually

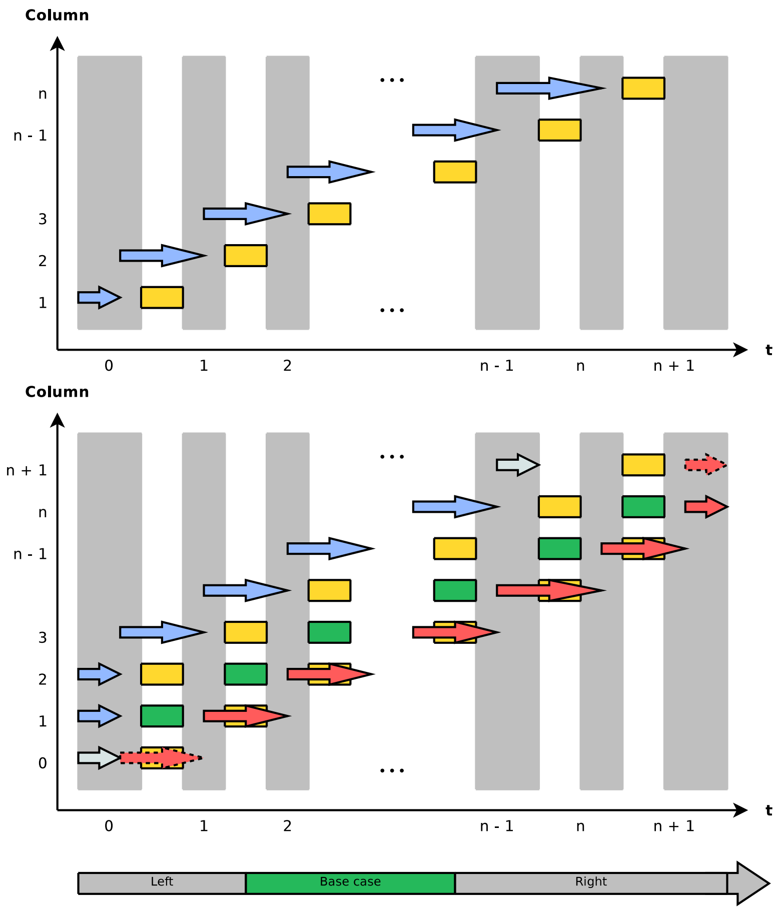

Figure 7.8: Schedule of the memory management routine for a pointwise coupled Gauss-Seidel relaxation step, regarding read-only single vectors (top) and read-write triple vectors (bottom).

describing store operations, since the boundary is meant to be initialized in RAM as well.

Using this transparent routine, the actual computation of the Gauss-Seidel relaxation step only slightly differs from a sequential implementation, since the return value of the memory management routine indicating whether there are still lines to process or not can comfortably be used as a loop criterion over columns. Per column, the implementation then equals the standard sequential solution. Furthermore, it does not make any difference to the way the routine is written whether the field is initialized with zeroes, or whether the program works on actual data.

Because this routine combines the two modes of the method described in context of resampling, it also must distinguish two special cases on the left and three on the right boundary, since reading and writing are three calls apart and the special cases involved in a single read or write operation (cf. Paragraph 7.7.2) are now merged. Furthermore, the zero initialization on the right is designed to replace the load operation elsewise issued, this is why the second but last call to the routine already describes a special setting.

Since branch mispredictions are quite expensive on the Cell Broadband Engine and it lacks a branch prediction routine (cf. Paragraph 3.5.4), it is worthwhile to hint the compiler to favour the base case, since the number of mispredictions can then be capped at the total number of special cases. Such branch hinting can be included by the compiler builtin `__builtin_expect`, which annotates a boolean evaluation with a value the compiler is meant to expect this term evaluates to when generating the binary code [1]. This method is from now on used in all memory management routines, where one case can explicitly be marked as standard case.

7.7.4 Initialization

The way of boundary handling in Paragraph 7.7.3 has another positive side effect: On the one hand, initializing a column with zeroes is typically a lot more faster than fetching a zero boundary from RAM. Additionally however, it also offers a fast alternative to a dedicated boundary initialization routine at the beginning of the computation:

Left and right boundaries are still quite cheap to set, since the DMA requests cover whole columns. For top and bottom boundaries however, the effective density would be way lower: For two pixels per column to set, either two dedicated requests over 128 bits of minimal alignment need to be established, whole columns need to be loaded and stored only to set these two values, or a sufficiently large block wrapping around the boundaries needs to be handled at once which would then contain the bottom boundary of one and the top boundary of the following column. Neither of these options is really feasible to realize.

Because of this observation, top and bottom boundary initializations have been assigned to the Gauss-Seidel steps, as well as to the residual computation described below, such that explicit routines initializing boundaries can be disregarded at all, which hence saves about 1-2 milliseconds per frame and SPU, depending on the image dimensions.

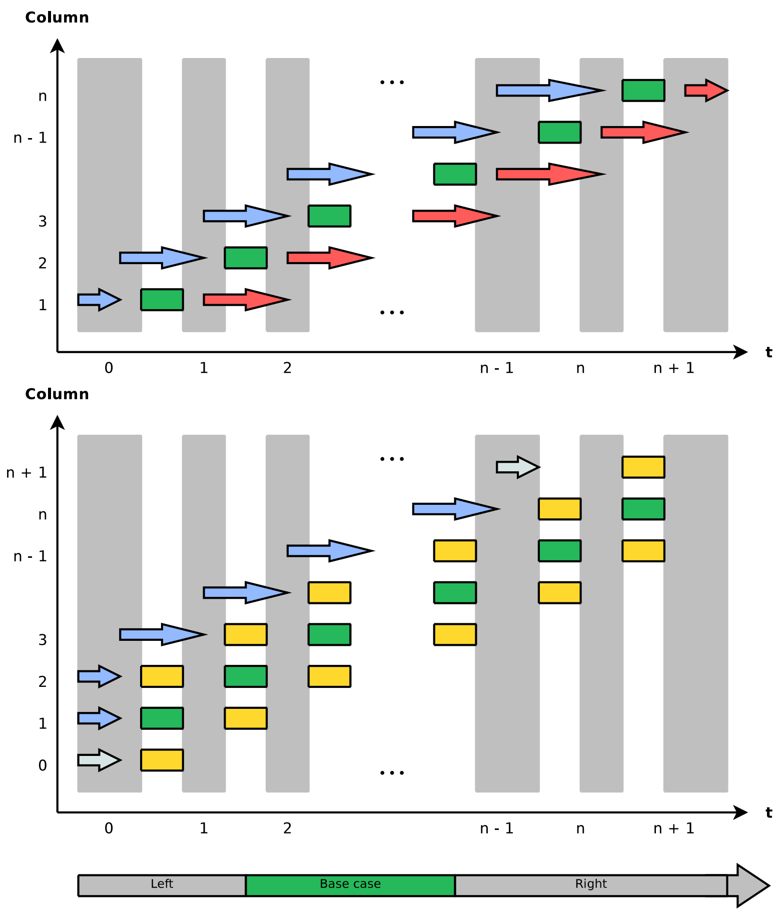

Figure 7.9: Schedule of the memory management routine for the residual computation step, regarding read-only and write-only single vectors (merged in the top diagram) and read-only triple vectors (bottom).

7.7.5 Residual Computation

To descent recursively within a V or W cycle, the low-frequent residual $r' = A^i w' - b^i$ must be computed, which can then be downsampled to yield the right hand side b^{i+1} for the error compensating problem on the next coarser grid (cf. Section 7.1). For this step, the whole linear system including the current intermediate solution must progressively be fetched for reading, and the vectors holding the residuals need to be written back to RAM in the same frequency.

Except for the preliminary flow field, which needs to be represented by three columns to provide a full four-neighborhood and is thus handled in a four-vector ring buffer, all other values are only column-based and can hence reside in double column ring buffers either designed for reading or for writing.

Indeed, the memory management routine turns out to be quite similar to its counterpart for the Gauss-Seidel step, despite of the fact that the written vectors do not happen to coincide with the quadruple ring buffers involved in the setting, such that explicit writing of boundary columns does not need to be regarded at all. Again, zero-boundaries on the left and the right are however computed rather than loaded and the remaining bounds are set in between the respective column is loaded and processed.

This renders the yielded solution partly inconsistent to its counterpart in Bruhn and Weickert's implementation, since the leftmost and rightmost columns in RAM still contain undefined contents rather than zeroes. However, this restriction does not hurt as long as it is appropriately handled on SPU side, because the following resampling step only works on nontrivial, i.e. inner, columns anyway.

7.7.6 Matrix Addition

The addition of two matrices, as it can be found for the correction of a finer error by a coarser adjustment field, is an independent operation on a per-element basis. This allows for a high level of parallelism, both on instruction, as well as on memory level. Because this addition occurs always for two matrices in parallel, these two steps have been unified in one routine to generate more compact utilization of the SPU's arithmetic pipeline.

Additionally, a 512 bits alignment has been chosen, which means that the length of column representations in Local Store and RAM can vary by up to 384 bits. This offers the opportunity to perform loop unrolling over as much as four independent SIMD operations per matrix being processed, which comes down to eight independent operations taking place inside the loop body. This is considered to be enough for the compiler to reschedule and to interleave the instructions to achieve dense code.

For the memory management, a composition of two writing and four reading double ring buffer methods like they are described in Paragraph 7.7.2 has been used. It is indeed necessary to merge these routines rather than keeping them modular, since waiting times for all DMA operations to finish can be overlaid this way. In the naive setting, all DMA requests would be issued time shifted to each other, which in this case causes up to six times as long latencies.

This routine already concludes the part covering the pure solver. Everything now following is used for the setup of the equation system, which is actually no longer method specific, but specific to the time decomposed layout, and to the CLG model used. Anyway, as the respective considerations in context of the spatially distributed preparation phase for the Red-Black SOR solver (cf. 6.5.7) have shown, little synchronisation was necessary in these cases as well, such that many of the improvements described in the following sections can be applied to speed up the previous SOR experiments as well.

7.7.7 Motion Tensor Computation

The derivative approximation and motion tensor setup follows closely the implementation pursued in the Red-Black SOR setting (cf. Paragraph 6.5.5). Since the stencil for the Taylor expansion covers five elements, five plus one spare column of a linearly interpolated intermediate frame of both input images are simultaneously kept in cache. Within this stencil, the x derivative in the central column can immediately be computed. After establishing Neumann

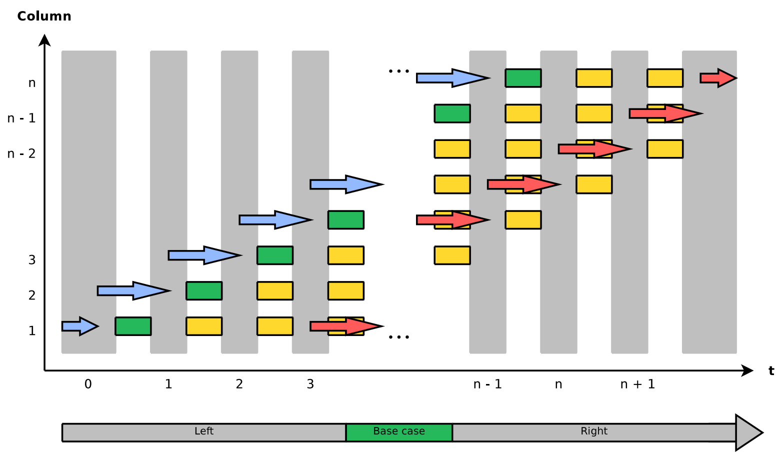

Figure 7.10: Schedule of the memory management routine for the motion tensor computation.

boundary conditions for the interpolation in this column, the same can be done for the y direction as well. For this reason, the Local Store representations of this data have again be chosen 128 bits wider on each side.

Because the temporal derivative under application of a forward difference needs the original frame contents instead of the interpolation and their load is two iteration steps ahead to create the five column stencil, the temporal derivative is cached in a three column ring buffer.

Since the two ring buffers used are both not involved in DMA operations, but only hold temporary values, they are rotated in the actual routine rather then in the memory management function. The local copies of the input frames and the motion tensor entries to output are all modelled as double ring buffers.

Figure 7.10 displays the scheduling of the memory management routine deployed. The symbols are again chosen similarly to before, whereby yellow now denotes dependent values of this column still held in cache. Even though the general structure is not too different to the single routines used in context of resampling, the extraordinary large displacement between reading and writing introduces eight special cases in total, which can in fact be reduced to six different cases.

7.7.8 Equation System Setup

After the motion tensor is in place, the last step in creating the linear system can be executed, namely to precompute three henceforth constant parts of the equation system. Like for SOR, these are the main diagonals concerning the flow field components u and v, as well as the common determinant, of which the inverse is stored such that later only multiplications need to be applied instead of a division.

Though this step would comfortably hook up with the motion tensor computation and columns would not need to be re-fetched this way, this concept is annihilated by the optional presmoothing step of the motion tensor entries, which can optionally be issued in between both parts. So, three vectors are again to be fetched, namely the motion tensor entries $[J_{11}]$ and $[J_{22}]$ for the respective diagonal, as well as $[J_{12}]$ for the determinant.

In this context, all operations are performed in full SIMD width. For this purpose, the boundary conditions need to be considered separately: In x direction, this is no problem, since SIMD vectors can be regarded homogeneously. Here, one 128 bit sample vector representing boundary characteristics can be created once per column, and then easily be included in the iteration over all chunks of 128 bit the column provides. In y direction in contrast, boundary conditions are constant across columns, but vary locally. This is why one complete column representing special cases is allocated to the begin of the routine, and is then reused for all single columns processed.

7.7.9 Presmoothing

Despite of the presmoothing steps for both the input frames and the motion tensor, the Full Multigrid SPU kernel is already comprehensively described. Luckily, the algorithms for both operations do not differ at all, if the input data is processed on a per-element basis. This means that for the input frames, both images need to be regarded one after the other, which is beneficial anyway (cf. Paragraph 7.7.1), and the same holds for the elements of the motion tensor, which however does not hurt at all.

Since convolutions are separable, the different smoothing directions can again be processed in a decoupled manner. Before however, the program creates a convolution mask containing one side of the symmetric Gaussian kernel of a variance previously chosen by the user, and stores it into a mask vector. This can then be used by the presmoothing routines and since the kernel is symmetric, both sides are entirely described by this present representation. Depending on whether the masks to be computed will be the same, based on equal grid spacing and variance, the routine is called once or twice, by allocating one joint mask for both direction, or dedicated ones if these are required.

Presmoothing in y Direction

Like for resampling, the presmoothing step alongside the data orientation in memory is less expensive, since there are no dependencies across columns. This means not only that columns can be fetched one after the other without the necessity to keep some of them longer in cache, it also motivates to use a loop unrolling approach over blocks of columns. This produces more independent instructions at a time and hence allows the compiler to reschedule these instructions to reduce latencies to a minimum. Hence, an approach comparable to the one described with respect to resampling in y direction (cf. Paragraph 7.7.2) has been applied.

Four two-vector ring buffers are allocated for input and output, each, while the length of each container is chosen to hold three times the height of a column. The central third of the input vectors are initially filled with the first four columns after the left boundary, and the next block is already requested into the inner third of the spare sections when the first is given into the arithmetic part of the routine. Latter then first mirrors sufficiently many values from the inner into the outer thirds such that the actual convolution does not need to consider special cases at all. The number of pixels set this way is given by the cutoff width of the Gaussian kernel applied. Naturally, since all elements computed upon are residing inside the Local Store and SIMD-based optimizations are hardly to achieve due to the inhomogeneity of the convolution mask used, the routine does not differ too much from the sequential implementation. However, in contrast to latter program, everything is explicitly coded four times in parallel.

Eventually, the results of this computation are transferred to the RAM and depending on the number of remaining columns, the running reading request is finalized and the next chunk being demanded. The algorithm then continues on the next block of four columns, while DMA operations both reading and writing are still running.

Presmoothing in x Direction

In the other direction, a much more sophisticated approach is necessary. As this has already been the case for other separable operations like resampling, the columns can be handled as entities, since there are no dependencies across rows. However, because the Local Store capacity is restricted and the Gaussian kernel can in principle be cut off at an image width, it is infeasible to keep the whole area of incidence in there.

Instead, two different operations need to be distinguished in this context: The target vector can be advanced in a single-step left to right manner, as this has already been done for other vectors. However, in a reading sense, all columns in a certain neighborhood of the respective column need to be fetched one after the other, which renders this pointer to show an irregular random access behaviour.

Unlike the quite comparable concept pursued in context of the Red-Black SOR solver (cf. Paragraph 6.5.4), no further latency should be introduced this time. Therefore, the return values of the corresponding memory management function have been extended to a triple state value, either denoting that a usual integration step should be performed on the data provided, that the incoming data belongs to a new target vector, this is why the resulting value should be assigned rather than added, or detailing that the process has come to an end. Latter case has again been represented by zero, such that the return value of the function can still be used as a pseudo-boolean loop condition.

To maintain this special behaviour, the function is given two state identifiers at hand, one of which denotes the position of the target vector currently valid, and the other specifying which of its neighbors, relative to its own position, is represented in the related vector in the Local Store representation. Since the convolution mask is symmetric, the neighborhood can be loaded pairwise, i.e. starting with the single center column in the beginning. All further loads always give the pair of columns encapsulating the already processed block, and hence offer the opportunity to explicitly process those in parallel to reduce scheduling latency. Fortunately, the distinction whether one single column or a pair thereof is currently valid is already modelled by the two positive cases of the triple state return value. Both the writing, as well as the two reading vectors are designed as two-way ring buffers and already filled or written back backgrounded to arithmetic operations.

To simplify the treatment at boundaries, which could in the worst case have an impact on the whole image, if the kernel is not cut into a smaller scale than the image width, mirroring is

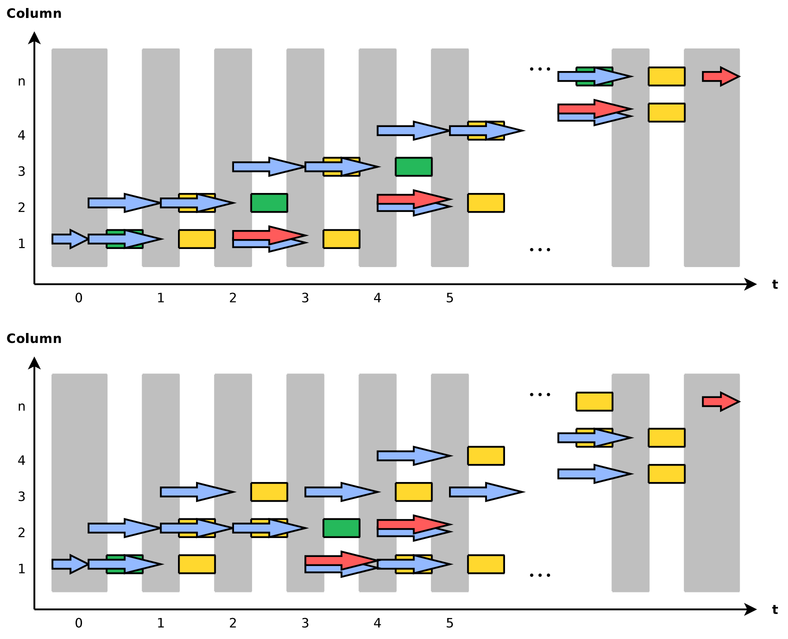

Figure 7.11: Sample schedule of the memory management routine for presmoothing, with a Gaussian convolution mask cut off at three (top) and five (bottom) elements.

natively implemented in the memory routine, which mirrors the incident pointer for fetching a neighborhood element if it exceeds a boundary.

As a further optimization, loop unrolling over four times 128 bit SIMD width has been applied. Since the vectors are large enough anyway to allow mirroring, the only problem remaining are the boundary values accidentally overwritten thereby. This issue is handled by the memory management routine, which masks these values to zero just before starting the DMA request to write them back. Because convolution mask entries are constant on a per-column basis, the current weight is once propagated into all four slots of a SIMD vector and then used as such.

Figure 7.11 shows the scheduling during two sample runs, one using a mask consisting of a discrete Gaussian cut off at a width of three, and the other at five. Green blocks hereby denote central columns, while yellow ones symbolize the neighborhood which is being added in a weighted manner. Like before, blue arrows denote reading DMA operations, while red ones describe writing ones.

The larger the stencil becomes, the more columns are mirrored at the boundaries, which introduces some suboptimal situations. For instance, in the run sketched in the bottom half of Figure 7.11, the second column is fetched three times in succession. Because its value has not changed in the meantime, one would like to reuse existing representations. The problem at this point is however where the line should be drawn, since these considerations are always a compromise between time and memory efficiency. Buffering of columns would require a more sophisticated cache strategy, like a least recently used (LSR) strategy in addition to the lookahead structure described above. Considering sufficiently small convolution masks, this idea could indeed be connected to a speedup. However, since additional administrative vectors are needed in this case and the implementation is rather nontrivial, it is left for future work.

7.7.10 Debugging

So far, all components of the actual program have been described. However, debugging within this framework is quite complicated. Like already sketched with respect to the development platform established on the Playstation 3 (cf. Section 4.4), common debugging techniques known from sequential hardware are often unlikely to yield suitable results, or are not available at all.

A quite helpful approach is hence to output debug information using simple `printf` statements, but this method is also quite restricted: Data to be output to the screen necessarily needs to reside in the Local Store, but since many problems occurring in the program actually referred to local vectors being written back to wrong RAM locations, changes of such memory cells cannot be traced naively. Instead, at least 16 surrounding bytes need to be loaded into the Local Store, whereby a correct alignment needs to be ensured and since this special part has no claim to realtime performance, the DMA operation should be finalized immediately.

To simplify these calls, a dedicated inspection routine has been designed, which accepts a potentially misaligned pointer to a scalar value, and a string for the screen output. It then first computes the associated base pointer to the 128 bit vector this vector resides in, requests this vector from the RAM to a temporary location, waits for it to become valid, and dereferences the actual location requested by the programmer. The value at this point is then concatenated with the string the programmer chooses and flushed to the screen.

Even though this technique is rather unsophisticated and also sometimes quite tedious, it turned out to be the most convenient method for many memory-related issues, such as buffer overflows or underflows, DMA operations to wrong RAM locations, or undefined variables related to wrong image boundary treatment. One quite severe class of bugs is in this context related to the special cache functionality: Since the Local Store serves both as instruction and data cache, it does not explicitly forbid self-modifying code. Due to this fact, pointer miscomputations can cause data to be written into some arbitrary location in code, which typically does not express in a crash of the actually failing routine, but in some other function called later in the program flow. Very frequently, the instructions accidentally created are found to be interpreted as branch instructions, even though according to the SPU ISA documentation, the related OpCodes involving the pattern `00110` in the five most significant bits should be quite unlikely to match by coincidence [33].

This might be due to an improper handling of illegal instructions on SPU side, but unfortunately no documentation could be found about this issue. If this interpretation proved true however, it would mean that the SPU hardware only checks every incoming instruction for a certain OpCode pattern to occur until it can doubtlessly be attributed to a certain assembly token, but does not check the remaining bits of the OpCode to be compliant to this assumption. On most modern hardware, this second step is indeed implemented and in case that the full OpCode does not match any mnemonic, an illegal instruction interrupt is thrown.

7.8 Binary Optimization

The GNU C Compiler offers the opportunity to apply so-called inlining of ‘simple’ functions. Using this technique, functions are not called internally by extending the stack by one function frame and then copying the parameter values to their new position. Instead, the function code is completely embedded into the calling function, while parameters are already matched at compile time, thus avoiding both memory operations and branching. Which functions are considered to be ’simple’ and which are not usually depends on the interpretation of the compiler designers.

Because branching is quite expensive on SPUs (cf. paragraph 3.5.4), it is indeed often worthwile considering whether hinting the compiler to inline certain functions might speed up computation significantly. In terms of the C programming language, this hint is indicated by either the `inline` keyword, or the `__inline__` function attribute. However, both typically still underlie size restrictions for the function to be inlined and cannot be ensured just including these hints. In the negative case, the compiler warning flag **`-Winline`** makes the compiler report those deviations.

For the Full Multigrid solver framework, the compiler refuses to inline the DMA functions due to massive code size, but due to them being run in style of a frequent background task, inlining is nevertheless desireable. Therefore, the default behaviour has been overridden by adding the flags

```
-finline-limit=10000 --param large-function-growth=1000
```

to **`gcc`**’s command line. While **`-finline-limit`** increases the actual inlining code size limit, the **`large-function-growth`** parameter allows inlining in the case that the calling function is significantly blown up in size. Hereby, $n = 1000$ sets the limit to 1000% of the original code size:

$$\frac{i+c}{c} \overset{!}{<} \frac{n}{100} + 1,$$

where i determines the size of the inlined functions, i.e. the part extended by inlining, c the code size of the calling function without any inlining performed, and n the parameter as denoted above.

With both values chosen empirically to 10000 and 1000, respectively, a speedup of approximately 7.8 ms per frame could be achieved. The benchmark is based on the Yosemite sequence with parameter settings $\sigma = 1.3$ and $\rho = 2.3$.

Buttari *et al.* suggest to try the compiler optimization flag **`-Os`** minimizing the resulting binary size rather than optimizing the program for speed [18]. Own experiments however proved this approach to be bad for the programs developed in context of this thesis, since the saved time due to a faster kernel transmission is already absorbed during computation of the first frame. For larger numbers of frames computed, a loss down to about a third of the performance achieved with **`-O3`** has been measured.

In the following, the **`-O3`** compiled version using inlining of all DMA functions is tested for its runtime based on different environment configurations.

7.9 Benchmarks

7.9.1 Impact of Different Image Sizes

In the first experiment with this method, the performance over different frame sizes has been evaluated. For this test, frames 8 and 9 of the Yosemite sequence scaled to halved and quartered edge length have again been measured over intervals of 25 and 50 runs, and finally the difference was regarded as an extrapolation towards infinitely large test intervals. Optionally, the frames and motion tensor entries have been convolved with Gaussian kernels, such that the appearance corresponds to variances of $\sigma = 1.3$ and $\rho = 2.3$ for the 316×252 pixels version, respectively. The optional intermediate synchronization step is not used this time.

This time, a problem already known from SOR implementations emerges even more severe than before: Because six complete data sets need to be allocated in RAM when all SPUs are occupied, the physical memory does not even suffice for images sized 632×504 pixels. Of course, the Linux kernel handles this problem by swapping data sets out on the harddisk, but the results achieved are not nearly comparable to the results entirely computed on RAM, since the swapping largely annihilates the speedup due to the faster processor.

Figure 7.12 and Table 7.3 visualize the results of this benchmark. On the first view, the general appearance of the graphs already reveals a strong similarity between the behaviours of the Pentium 4 implementations and the SPU based variant. Different to the distributed Red-Black solver (cf. Paragraphs 6.5.8 and 6.6.3), the algorithm performs particularly good on smaller image sizes, where it obviously does not suffer from scheduling overhead.

However, one can furthermore observe a speed gain by a factor of about 4 to 5.3, depending on whether presmoothing is requested, or not. This is quite remarkable, since this speedup factor is entirely comparable to the respective result in the joint Red-Black setting (cf. Paragraph 6.6.3), although the absolute number of frames per second and thus also the amount of data involved is unlike higher.

Assuming the measured peak performance of 171.03 FPS for the solver without smoothing and two frames per core to be loaded, 103.91 MB of input data is processed per second, and the total bus load also covering all intermediate results and helper variables way exceeds the 1 GB mark.

The actual workload involved in such process can also expressed from a different perspective: For frames of 316×252 pixels, 171.03 FPS and two flow field components to be solved in each point, 27.24 millions of unknowns are determined in every second, which is quite an astonishing value and a significant speedup compared to 5.05 millions on the Intel Pentium 4 (cf. Section 7.2).

Another interesting observation can be made when one compares the distance between the results for varying test intervals. While constant parts played a subordinated role in the SOR settings, they are now indeed existent. Computing the runtime of the constant portion, it is with 61.8 ms in this case even lower than for Red-Black SOR with joint kernels, which needs 69.3 ms of initialization. Because the general throughput of data achieved with Full Multigrid is a lot higher than for SOR however, the relative impact of the initialization phase is a lot higher.

Table 7.3: Performance (FPS) of the Full Multigrid solver on 6 SPUs over different image sizes. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\rightarrow \infty$	**25**	**50**	$\rightarrow \infty$
79 $\times$ **63**	1162.13	1290.98	1451.97	894.20	951.81	1017.36
158 $\times$ **126**	402.16	446.09	500.78	256.54	280.99	310.58
316 $\times$ **252**	120.21	141.19	171.03	58.88	62.77	67.21

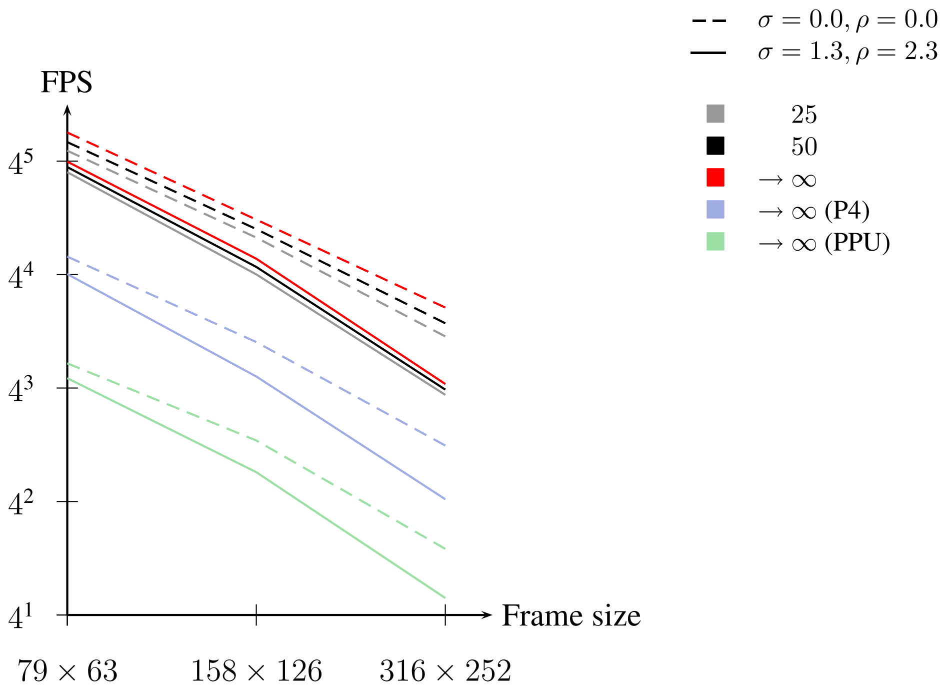

Figure 7.12: Performance (FPS) of the Full Multigrid solver on 6 SPUs over different image sizes. The P4 and PPU reference graphs are depicted in light blue and green, respectively.

Extrapolating the results of this experiment to larger image sizes, which is indeed feasible considering an application of the Cell processor in a device different to the Playstation 3, where more physical memory is available, about 19 pairs of frames sized 1280×1024 pixels could still be computed in every second, which can considered almost realtime. Since in these cases, the total number of available SPUs is typically not artificially restricted, the additional two SPUs usable in this case should suffice to achieve realtime performance.

7.9.2 Scaling with the Number of SPUs

Like for SOR, the next experiment shall give hints about how good the algorithm scales over a varying number of SPUs concerned with the problem. Here, no synchronization latencies should occur at all, and since the SPU kernels will soon self-regulate to run with a certain offset to each other, bus contemptions should automatically minimize, such that the overall acceleration with an increased number of SPUs should be even higher.

These assumptions can be confirmed by measurements, as Figure 7.13 and Table 7.4 show. For sufficiently large measurement intervals, the measurements converge towards an almost-linear scaling behaviour. This means that at least within the range of up to six SPUs, doubling of the number of cores involved also doubles the FPS count. Even though bus contemptions are very hard to predict, because little perturbations in the environment characteristics can already cause severe changes in the outcoming performance, there is evidence that this assumption holds as well for slightly higher numbers like the eight SPUs available on the IBM Cell blades: Because the SPUs can be assumed to be running in less conflicting time slots after quite some while and the RAM bus is with two times 12.8 GB/s still rather optimistically dimensioned to handle such unbursty requests (cf. Section 3.8), two more SPUs should indeed scale up to about 225 FPS.

7.9.3 Scaling with the Number of SPUs for Small Intervals

The distance between the measurements over 25 and 50 frames motivates for an interesting question: If the chosen test interval really has such a large impact, then what happens if only one frame should be computed per SPU?

Figure 7.14 and Table 7.5 show that small numbers of frames indeed deteriorate the achievable results of this parallelization attempt. Unfortunately, using more than two SPUs does not pay off anymore, but keeps the ratio between computed frames and the time needed for this operation almost constant. On the transition from four to five SPUs however, a little reproducible anomaly can be observed, which is probably related to a bus contemption on a time critical operation.

7.10 Discussion

The last experiment not only visualizes a problem restricted to a certain setting, but reveals a general issue with this architecture: Until the kernels are uploaded to the SPUs and they

Table 7.4: Performance (FPS) of the Full Multigrid solver on different SPU counts. Columns denote test intervals averaged over.

	No Smoothing			**Smoothing**		
	25	**50**	$\to \infty$	**25**	**50**	$\to \infty$
1 SPU	27.80	28.88	30.06	11.81	12.01	12.20
2 SPUs	53.57	56.62	60.03	23.28	23.75	24.24
3 SPUs	74.11	81.13	89.62	33.78	34.75	35.78
4 SPUs	92.66	103.43	117.02	43.10	45.29	47.70
5 SPUs	106.92	123.35	145.75	50.65	54.26	58.43
6 SPUs	120.21	141.19	171.03	58.88	62.77	67.21

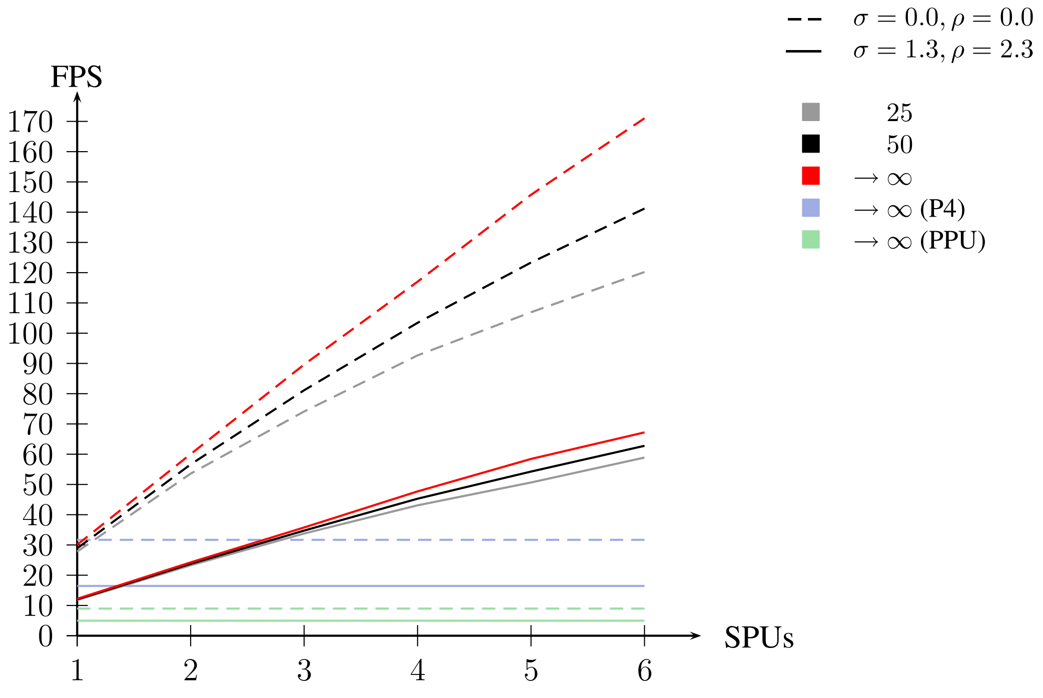

Figure 7.13: Performance (FPS) of the Full Multigrid solver on different SPU counts. The reference benchmarks on P4 and PPU are again plotted in blue and green, respectively.

Table 7.5: Performance (FPS) of the Full Multigrid solver on different SPU counts, whereby every available SPU computes only one frame.

	No Smoothing	**Smoothing**
1 SPU	10.00	6.45
2 SPUs	14.85	11.27
3 SPUs	14.96	12.92
4 SPUs	15.29	14.28
5 SPUs	15.16	12.91
6 SPUs	15.21	13.29

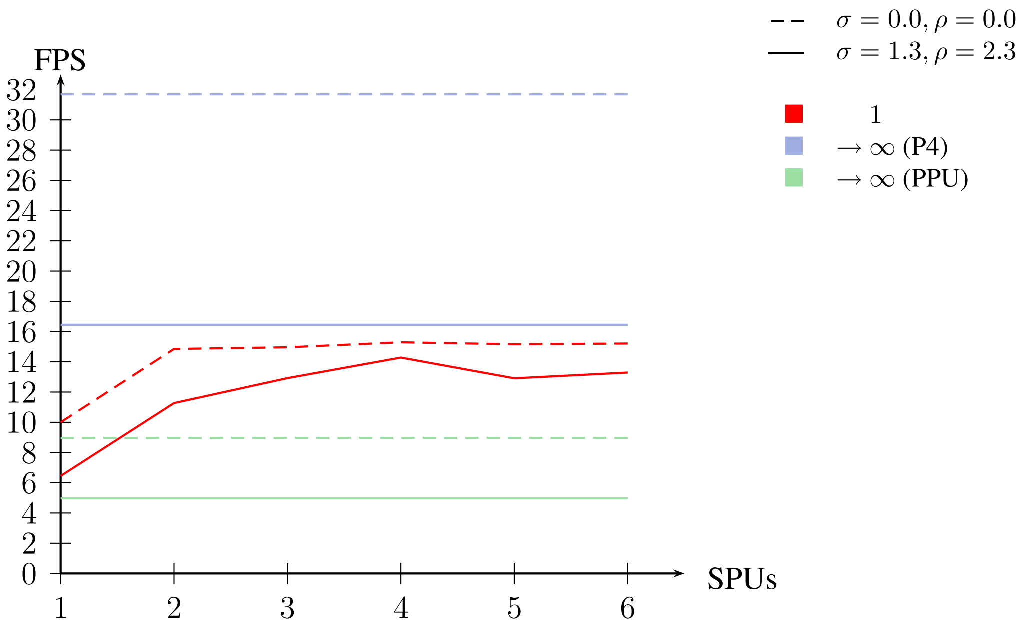

Figure 7.14: Performance (FPS) of the Full Multigrid solver on different SPU counts, whereby every available SPU computes only one frame. The reference benchmarks on P4 and PPU are again plotted in blue and green, respectively.

are dispatched, several milliseconds just pass without any visual effect. Afterwards however, these cores can become really fast and if their operations are hand-optimized, speed gains of about a factor of 5 in comparison to a single core implementation on a Pentium 4 are well achievable.

Because of this, the Cell Broadband Engine is indeed brilliantly suited, if larger amounts of data are to be processed. For smaller problems however, or if the general structure of the algorithm applied changes too rapidly over time, the contrary effect can arise and the time needed can even exceed implementations on traditional sequential hardware.

In addition, the results yielded by the Full Multigrid implementation presented in this chapter even numerically differ from the sequential implementation. This is due to the restricted IEEE compliance of single precision floating point operations: Most modern processors support round-to-nearest as the standard rounding mode whenever an arithmetic result cannot be entirely displayed with the available precision, and so does the PPU. The SPUs for single precision operations however only apply round-to-zero, which means that values are truncated after the last digit of the mantissa [31].

Assuming values in the range $[-100, 100]$ and taking a deviation by the least meaningful bit of the mantissa, the fifth decimal place is already a subject to changes. In particular when this error is upsampled in course of the Full Multigrid solver, variations in the fourth decimal place are the order of the day.

Anyway, one major insight of this implementation is that the simple strategy to parallelize the program by executing temporally independent partitions on different SPUs is indeed worthwhile to consider. Although the general throughput of the Full Multigrid solver is rather high and little perturbations have a significant impact on the general performance like the last experiment has shown, this strategy is able to achieve quite remarkable results.

A drawback is however that even though the throughput of this approach can in theory freely be increased by taking more cores into the setup, the latency does not change at all. This is different to spatially distributed problems like the Red-Black SOR solver, where additional SPUs both increases the FPS rate and lowers the latency of each frame to be computed. In interactive or response time critical application fields, this can indeed become a problem, since it means that the problem definition and the result propagation are always a constant amount of time apart. However, as long the pure workload per time is the predominant criterion, this parallelization technique seems to be a powerful alternative to distributed attempts. As a positive side-effect, this method is furthermore way easier to implement, since no synchronization needs to be handled at all and optimization comes down to SIMDization and adoptions to SPU-specific characteristics.

Chapter 8

Interactive Realtime Setup

Motivated by the results of the previous chapters, this last experiment shall demonstrate the effectiveness of these approaches within an interactive setup on the Sony Playstation 3. In particular, a webcam is being connected to the video console, and the input is used for a realtime computation of the optical flow field. This field is then color-coded and output to the display, whereby the coloring proposed by Bruhn is used (cf. Section 5.1) [15].

This way, the user can immediately observe the flow fields estimated from the live pictures coming in from the camera, interpret and also interactively influence them by introducing additional motion to the captured scene. Even though this experiment might only look playful at the first glance, it indeed provides many insights about the applicability of the Playstation 3 for real-world applications: Often, optical flow estimations are used as a module within a more extensive algorithm used for some different purpose. Structure-from-Motion or flow-driven deinterlacing methods are only two examples for such process building upon these methods.

Interactive programs for all of these approaches have in common that images are acquired from some internal or external device, they are prepared for the actual computation step, processed by the algorithm, and finally output to another device like the screen. Timing information about the several steps performed on the Playstation 3 can hence be worthwhile to appraise the general cost for such setups, including device-specific characteristics.

8.1 Installation of the Webcam

The camera used in this context is a Philips PCVC840K/20 webcam, connected to the Playstation 3 by one of its front USB 2.0 ports. It can be brought up under Linux using Luc Saillard's **pwc** driver, which is not available as a precompiled module for the present kernel version and architecture [55]. However, since the Linux kernel has been recompiled in an optimized version for this thesis anyway (cf. Section 4.1), both kernel sources and development tools are already available and the compilation process works in a straightforward manner.

After the module has been loaded and set to 30 FPS mode using the command

```
$ modprobe pwc leds=0 compression=0 power_save=0 fps=30 \
  size=vga
```

the camera device is available in devfs, and typically allocated as **`/dev/video0`**. Finally, the permissions need to be set world writeable and readable, such that users can access the device:

```
$ chmod 0666 /dev/video0
```

To read the webcam image, it suffices to open a file handler for the block device, and to continuously read frames from it that are provided in a row-before-column manner. The dimensions must hereby be known *a priori*, and each pixel is stored in eight bit grey values.

8.2 Adoption of the Program

During the development process of the SPU kernels, a SDL frontend has already been developed to pursue the results achieved by the algorithms, and to aid in debugging (cf. Section 5.7). This frontend is however not suited for an interactive realtime setting yet, since it still assumes the frames to be statically available at program start and thus entirely abstracts from time-critical input handling. While this is acceptable for benchmarks over the pure optical flow algorithm, it is no longer applicable for this realtime setting, because all factors are meant to be considered this time, and the time measurements have to be adjusted accordingly. Instead of measuring the time a certain number of frames needs to compute as this has persistently been done before, the time measurement is now evaluated and restarted from within the display routine, which hence captures all steps performed throughout the program, starting with the acquisition, over the solver, the color coding (cf. Figure 5.2), and finally the output to the screen. This also allows for an explicit runtime analysis of the distinct parts, since many steps can just be manually excluded from the process and the overall performance of the program then hints to the impact of this specific module.

A second novelty in this setup is the efficient storage of the involved data sets, as well as their interaction with both the display routine and the camera. Particularly the Full Multigrid implementation presented in Chapter 7 requires a different handling than before: Due to its displaced sequential processing order of incoming frames, they need own variable sets for input and output data, as well as for intermediate values. Different to before however, the real performance of the method becomes obvious, because it is now really working on a continuous chain of input frames, and presmoothing performed by one SPU renders the same step for the next SPU for one of its two input frames obsolete. This means, that instead of two input frames per SPU, one is sufficient, since the other one can be taken from the predecessor in the chain, and does not even require any further treatment.

To ensure data validity for this case, the optional intermediate synchronization step behind the presmoothing stage can be enabled (cf. Section 7.7.1). Experiments revealed however that this step is usually superfluous, since any SPU has to render one of the two frames, and two neighboring SPUs are furthermore dispatched with a certain offset, such that it is very likely that the earlier dispatched core finishes the first stage before the second. In fact, anomalies related to this issue have never been observed, and since every synchronization costs time, it has been left away for the sake of performance.

So far, only the memory demands related to the SPUs have been respected. Additionally however, the camera and display routines implemented on the PPU also need access to the least recent input and the most recent output vector, respectively, to provide new frames or to output flow fields to the screen. This introduces race conditions, since a solution must not be overwritten by the assigned SPU while the PPU reads it into the framebuffer, and no computation must be started on invalid or only partly valid input frames.

To anticipate that only five SPUs are working while one is always in clash with the PPU, and to reduce related latencies, an alternative strategy has been developed. Comparable to SPU internal DMA operations, where ringbuffers have been frequently used, this concept has also been applied to the input and output vectors for the Full Multigrid solver in RAM.

For a total number of six available SPUs, this results in one seven-element ringbuffer for the input frames, and another one containing seven pairs of directional flow field components. Intermediate results are in this context not affected at all, and can thus remain in six statically assigned regions. After the first two frames are loaded into the first two elements of the input buffer, the first SPU kernel is dispatched upon this pair. As soon as the third element is valid, the second SPU can process the second and third frames relying on the second frame already being smoothed by the first SPU, and so on. Since one element is always unused at any time, the PPU can fill it with new data while the SPU kernels are still computing the other flow fields simultaneously. The same holds for the output vectors: Here, one buffer element is in spare as well, and can be read out unhurriedly while the SPU kernels are working.

For a better visibility towards the user, the framework has furthermore been extended by a simple rescaling functionality duplicating a pixel to be output into several neighboring cells. This enlarges the displayed region though keeping the actual resolution constant, which makes it for instance easier for the auditorium to see the results in video projector presentations. Unfortunately, such implementation immediately affects performance, since this process can unfortunately not been handled by the graphics pipeline being hidden from the programmer, even though this task could natively be performed with hardware acceleration on most modern graphics cards.

Finally, output to the console has been extensively minimized. Instead of outputting the frames per second number to the console, it has been embedded as a bitmap font into the actual frame buffer. Because SDL has no native routines for doing so, the `sdl_picofont` library developed by Fredrik Hultin has been used [23]. It works based on constant bit patterns describing all ASCII letters in a fixed-width 8×8 pixels matrix, each, whereby a binary 1 depicts that the respective pixel belongs to the letter, and a 0 represents the background. This way, any text can be rendered as a bitmap font and is returned as an SDL surface object with text in a predefined color set on transparent background. Such SDL surface can then instantly be blitted onto the screen, using a double buffering approach. By this optimization, about 13 ms can be saved per computed frame, compared to a solution outputting the frames per second count on the console using the C `printf` routine. The reason for this difference being that severe might be related to the bad multitasking performance of the PPU and the necessity to schedule both the active console window, as well as the actual program reasonably well.

Table 8.1: Expenses for different parts of the program for the computation of one frame. 'Framework' refers to the remaining operations not explicitly listed, like synchronization or graphics output.

Matter of expense	Time
Optical flow algorithm	2 - 40 ms
Image acquisition	82 ms
Color coding	11 ms
Framework	5 ms

8.3 Results

Confident of the fast solver implementation, the 30 frames per second provided by the webcam should be no problem at all, in particular since the camera only provides a resolution of 160×120 pixels, which is about comparable to the halved edge length version of the Yosemite sequence often used throughout this thesis. Counter-intuitively however, the opposite is the case, and the actual frame rate for the fastest version without rescaling does not exceed 10.5 FPS.

Furthermore, only little difference can be observed among all SPU based methods, including the Block SOR solver, both variants of the Red-Black solution and the Full Multigrid implementation, and even a Full Multigrid reference implementation running entirely on the PPU. Only a pure PPU-based SOR method performs with 4.2 FPS about the factor of two worse. Apparently, the limiting factor is hence not located in the actual optical flow estimator, but somewhere else in the framework.

Table 8.1 lists the most expensive operations involved. From earlier experiments it is known that the actual optical flow algorithm needs about two to 40 milliseconds, depending on whether a fast Full Multigrid method or a slow SOR method like Block SOR is being applied. Surprisingly however, this section is way faster than 'organizational' operations on the PPU like the image acquisition step reading the frame out of the camera's framebuffer which consumes about 82 ms per frame. Since this operation is mostly simultaneous with SPU based computations, it dominates the process and the solver iterations can just be abstracted from, since they are taking place on another core. This explains the little difference between the different SPU implementations.

Though the remaining expenses are still not to underestimate at all and the PPU based color coding step is in particular about five times as expensive as the Full Multigrid solver itself, the main problem, i.e. the poor frame rate of about 13 FPS achieved by the camera driver, cannot effectively be addressed at all. It is most likely related to the special setting on the Playstation 3, for instance caused by its virtualization layer provided to the third party operating system (cf. Section 4.1), or to the bad multitasking of the PPU. In the latter case, a similar situation like observed before in context of the text console might arise, and bad scheduling on the two available pipelines could deteriorate both the performance of the webcam driver module and the program. The USB bus does however not seem to be the problem, since the data rate achieved exceeds the USB Low-Speed limit of 1.5 MBit/s, but is still way below the next critical mark of 12.5 MBit/s for USB Full-Speed [3]. Linux can hence be expected to address the device correctly as a Full-Speed device and should hence fully exhaust the 30 FPS captured by the camera.

Another explanation for this phenomenon could indeed be related to the design of the Cell processor itself, like mentioned by Perrone [48]. By initializing a single precision floating point valued frame ring buffer element of dimensions 160×120 pixels, 75 kB of memory are modified, which for the time being takes place in the local L1 and L2 caches of the PPU. With its 256 kB of L2 cache, the PPU is however not pressurized to hurry with writing these vectors back to RAM, but it might take some time until the Least Recently Used (LRU) strategy considers a certain amount of data dispensable. In the meantime, SPU requests issued to the MIC are redirected into the L2 cache and suffer from a bottleneck at the PPU's own bus controller. Though this interpretation does hardly supply an adequate explanation for the loss of an overall factor of three, it might at least play a significant role in the whole process.

One should note that the optional rescaling of the output image is as well expensive: By resampling the solution to a four times as big framebuffer representation with respect to the edge lengths of the image, additional 40 ms per frame are required. So far however, this step is only being naively implemented by single scalar assignments of values. By remodelling the operation in x direction as `memmove` operations over whole columns, and by a SIMDization of the y direction, which the PPU is as well capable of (cf. Section 3.4), this process can certainly be accelerated noticeably.

8.4 Summary

This experiment shows that even though the asymmetric layout of the Cell processor can exhaustively be used for arithmetic accelerations, the PPU still represents a bottleneck, especially when it comes to device interaction or organization of larger data sets later to be accessed by the SPUs. This reveals practical problems for various applications relying on a fast access to external devices, or even streamed data coming in via ethernet: Even though most devices can be memory-mapped and then immediately accessed by the SPUs, the PPU running the operating system kernel is often still occupied by taking care of the various drivers needed, as well as of synchronization and general application management tasks like allocation of memory, etc.

Especially within the arrangement in Sony's Playstation 3, where devices are accessed through a virtualization layer, this fact has an immediate effect on the performance, and thus restricts the number of problems that can be regarded to be suited for this architecture. Though pure arithmetics can be perfectly optimized by means of both instruction level and memory level parallelism, data hazards regarding a fast delivery of problem definitions, or sometimes also a rapid conveying of computed solutions towards a target device are a potential risk for realtime applications.

Chapter 9

Conclusion and Future Work

9.1 Overview

To the beginning of this thesis, the Playstation 3 video console has been explored, and the Cell Broadband Engine has in particular been closer examined. Important characteristics and technical specifications have thereby been showcased.

Based on these hardware specifications, it has been shown how to set up an entire development and testing environment. For this purpose, a Linux distribution has been installed and customized, and been extended by a development framework. The debugging capabilities have been explained, and the applicability to common program errors has been critically questioned.

In the following, the optical flow problem has been depicted, whereby the CLG approach by Bruhn *et al.* has been explained in more detail [17]. To solve the this particular variational model, the corresponding Euler-Lagrange equations have been developed and discretized.

To solve the resulting equation system, the Successive Over-Relaxation method has been explained and used first. Three concrete implementations, namely a Block SOR solver, and two Red-Black SOR solvers, have been proposed. Comparing their performance to a sequential implementation of Bruhn and Weickert [16], speedup factors of 1.3, 3.4 and 5.3, respectively, could be achieved. With the second Red-Black SOR solver, it is hence possible to yield realtime performance for 100 iterations on frame sizes of 316×252 pixels, computing 24.60 frames per second.

As a second class of algorithms, a Full Multigrid solver has been presented, and a parallel variant thereof has been developed. Unlike for the SOR implementations spatially distributing the problem on several SPUs however, the algorithm has only been parallelized on instruction level and has been left sequential with respect to each single flow field frame estimated, such that memory level parallelism is introduced as soon as more than two frames are to be computed. With a peak performance of 171.03 FPS for 316×252 pixels large frames, this method turns out to be up to 5.4 times faster than the Pentium 4 implementation.

Finally, the developed algorithms have been tested in an interactive real world setting with a USB webcam connected to the Playstation 3. It has been found that the PPU, perhaps in combination with the hardware virtualization in the concrete setting, can represent a bottleneck when it comes to realtime processing of data acquired from external or internal devices, and that in this respect, the Cell processor might sometimes not be able to unveil its full potential.

9.2 Conclusion

In this thesis, several facets of the Cell Broadband Engine, but in particular also the Playstation 3, have been touched. The experiments revealed that it is indeed possible to speed up the computation process for optical flow estimations by quite an impressive factor beyond 5.0. This is in particular interesting, since the video console is cheaper than an average multimedia PC.

Unfortunately, there are on the other hand also several drawbacks to be reported. From time to time, the PPU has turned out to be indeed a major bottleneck when it comes to interactions of the fast parallel program with the operating system, and since a remedy would almost certainly imply to parallelize a number system daemons, this problem seems hard to address at this point.

Secondly, timing problems due to synchronization seem to be a major problem for some tasks. Though this problem has been addressed in the Full Multigrid setting by executing whole frame computations on different cores, the achieved solution is still not perfect for several real world applications, since the latency every result is afflicted with cannot be reduced at all.

Finally, the development process on the Cell processor has unfortunately turned out to be quite tedious and error-prone, since the tools nowadays available are still quite less powerful, both with respect to compilation, and debugging. Many steps are still dominated by manual work, and cannot be accelerated at all.

The decision whether the Cell Broadband Engine is applicable for an actual implementation using optical flow, or not, can hence not be answered without further knowledge of the actual application and depends strongly on the effort the developer is disposed to invest. Though, regardless of hurdles to be cleared during development, the prospected speed gain is indeed interesting and should still be way superior to modern symmetrical multi-core processors. In particular with respect to larger problem sizes, it is however advisable to supply an increased amount of physical memory, or to design more evolved storage concepts, such that the achieved speedup is not going to be lost in expensive swapping processes again.

9.3 Future Work

The work pursued in this thesis can be continued by many means, with respect to the underlying model, towards a better hardware configuration and device interaction, or in terms of many optimizations of various parts of the software.

For instance, only a very simple optical flow approach, namely the CLG model by Bruhn *et al.*, has been used so far [17]. Several more accurate, but also more complex approaches could be worthwhile to test, such as the approach by Uras *et al.* using a different data term [62], spatiotemporal approaches like proposed by Nagel [26], or highly accurate methods including nonquadratic penalization functions and multiple constancy assumptions are conceivable, like introduced by Brox *et al.* [13]. Different characteristics with respect to cache scheduling might become challenging in this context.

Another interesting option is to re-implement the Full Multigrid solver (cf. Chapter 7) in an actual spatially distributed setting, like this has been pursued for the SOR solvers (cf. Chapter 6). This would offer the opportunity to yield about the same frame rates as for the present variant (cf. Section 7.9), by reducing the latency. The pyramidal memory structure might however require a more sophisticated decomposition strategy than stripes, since enormous synchronization overheads could otherwise be introduced on coarser scales.

The second class of optimizations imaginable at this point is related to expected runtime gains, rather than referring to visual improvements. For instance, the DMA transfers described in this thesis are still way from being optimal, since all of them have been constrained to a certain image dimension, rather than to optimal chunks in terms of the underlying hardware (cf. for instance Paragraphs 6.4.1, 6.5.6, 7.7.2). A possible remedy to this problem is given by a software implemented cache management routine providing a virtual memory layer, thereby decoupling the memory management from the actual problem characteristics, while still providing foresighted DMA requests.

Emulating a cache management hardware circuit, such routines could also adopt other attributes thereof, like a Least Recently Used (LRU) strategy. Managing a larger amount of data than currently needed by the program, obsolete vectors can be kept in the Local Store for a longer period of time. This way, frequent redundant data loads on equal lines, like for the presmoothing step in x direction in terms of the Full Multigrid solver (cf. Paragraph 7.7.9), could be circumvented, which will have a measurable impact on the runtime.

To minimize the time DMA transfers require, to reduce bus contemptions at the Memory Interface Controller (MIC), and to increase the maximal frame size possible to process, it might be helpful to compress data inside the RAM by using a very cheap compression algorithm. Instantly on load or store of a vector, the data could then be decoded or encoded on the fly, such that the SPU kernels can operate on these vectors like before.

On the compilation of the Linux kernel, support for huge TLB pages has already been enabled. By use of this technique, additional speed gains of about 5% can be expected [46].

Finally, further experiments could be made using different input sources to eliminate bottlenecks responsible for the bad performance of the Playstation 3 in the last experiment (cf. Chapter 8). Videos from the internal DVD and BlueRay player are as well imaginable, as live streaming via the Gigabit ethernet device. Fast data delivery is a key issue when using the Playstation 3 for realtime applications, and can be a decisive factor for or against the applicability of this particular device for scientific computing in a realtime setup.

List of Figures

List of Tables

Bibliography

[1] Other built-in functions provided by GCC. Online: http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. Retrieved 08-03-19.

[2] Root privilege required to run certain function in cell. Online: http://www.ibm.com/developerworks/forums/message.jspa?messageID=13863171#13863171. IBM developerWorks web forum posts. Retrieved 08-02-21.

[3] Universal Serial Bus Specification, Revision 2.0. Technical report, Compaq Computer Corp., Hewlett-Packard Company, Intel Corp., Lucent Technologies, Microsoft Corp., NEC Corp., and Koninklijke Philips Electronics N.V., April 2000.

[4] FlexIO™ Processor Bus Product Brief. Online: http://www.rambus.com/assets/documents/products/FlexIO_ProcessorBus_ProductBrief.pdf, 2005. Retrieved 08-01-27.

[5] Meet the experts: David Krolak on the Cell Broadband Engine EIB bus. Online: http://www-128.ibm.com/developerworks/power/library/pa-expert9/, December 2005. Retrieved 08-01-26.

[6] Mike Acton, Jakub Kurzak, and Alfredo Buttari. HowTo: Huge TLB pages on PS3 Linux. Online: http://www.cellperformance.com/articles/2007/01/howto_huge_tlb_pages_on_ps3_li.html, January 2007. Retrieved: 08-04-07.

[7] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 25–23. IEEE Press, 2006.

[8] Arnd Bergmann. Spufs: The Cell Synergistic Processing Unit as a virtual file system. Technical report, International Business Machines Corp. (IBM), June 2005.

[9] Michael J. Black. Yosemite Sequence FAQ. Online: http://www.cs.brown.edu/people/black/Sequences/yosFAQ.html. Retrieved: 08-03-13.

[10] A. Brandt. Multi-level adaptive solution to boundary-value problems. *Mathematics of Computations*, 31:333–390, 1977.

[11] Brokensh. List of Odd/Even SPU Instructions. Online: `http://www-128.ibm.com/developerworks/forums/thread.jspa?threadID=104798`. IBM developerWorks web forum post. Retrieved: 08-03-26.

[12] Ilja N. Bronštein, Konstantin A. Semendjajew, Gerhard Musiol, and Heiner Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, Thun, 5th edition, 2001. (German).

[13] Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High Accuracy Optical Flow Estimation Based on a Theory for Warping. In T. Pajdla and J. Matas, editors, *Proc. 8th European Conference on Computer Vision, Lecture Notes in Computer Science 3024*, volume 4, pages 25–36. Springer-Verlag Berlin Heidelberg, May 2004.

[14] A. Bruhn, J. Weickert, C. Feddern, T. Kohlberger, and C. Schnörr. Variational Optical Flow Computation in Real-Time. *IEEE Transactions on Image Processing*, 14/5:608–615, May 2005.

[15] Andrés Bruhn. *Variational Optic Flow Computation - Accurate Modelling and Efficient Numerics*. PhD thesis, Saarland University, 2006.

[16] Andrés Bruhn. Lecture notes to Correspondence Problems in Computer Vision, pages 3.3,4.10-4.14. Saarland University, 2007.

[17] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr. Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods. *International Journal of Computer Vision*, 61(3):211–231, 2005.

[18] Alfredo Buttari, Piotr Luszczek, Jakub Dongarra, and George Bosilca. A Rough Guide to Scientific Computing On the PlayStation 3. Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, May 2007.

[19] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. Technical report, IBM developerWorks, November 2005. Online: `http://www.ibm.com/developerworks/power/library/pa-cellperf`, Retrieved 08-01-27.

[20] David Krolak. Unleashing the Cell Broadband Engine Processor: The Element Interconnect Bus. In *Papers from the Fall Processor Forum 2005*. IBM developerWorks, November 2005.

[21] L. E. Elsgolc. *Calculus of Variations*. Pergamon Press Ltd., 1962.

[22] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):118–135, September 1972.

[23] Fredrik Hultin. sdl_picofont. Online: `http://nurd.se/~noname/`. Retrieved: 08-04-02.

[24] Free Software Foundation, Inc. GDB: The GNU Project Debugger. GDB Documentation. Online: `http://www.gnu.org/software/gdb/documentation/`, 2007. Retrieved 08-03-03.

[25] Greg Chabala. How To Install Fedore Core 6 on your Playstation 3. Online: `http://gregchabala.com/computer/playstation3/howto-linux-on-ps3.php`, September 2007. Retrieved: 08-03-25.

[26] Hans-Hellmut Nagel. Extending the 'Oriented Smoothness Constraint' into the Temporal Domain and the Estimation of Derivatives of Optical Flow. In Olivier D. Faugeras, editor, *Computer Vision - ECCV'90, First European Conference on Computer Vision, Antibes, France, April 23-27, 1990, Proceedings*, volume 427 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 1990.

[27] Berthold K. P. Horn and Brian G. Schunck. Determining Optical Flow. *Artifical Intelligence*, 17:185–203, 1981.

[28] IBM developerWorks. *SPE Runtime Management Library Version 2.2*, CBEA JSRE Series edition, September 2007.

[29] IBM Systems and Technology Group. Developing Code for Cell, May 2006. Course notes L3T2H1-36.

[30] International Business Machines Corp. (IBM). *Cell Broadband Engine Programming Tutorial*, December 2006.

[31] International Business Machines Corp. (IBM). *SIMD Math Library Specification for Cell Broadband Engine Architecture*, CBEA JSRE Series edition, September 2007.

[32] International Business Machines Corp. (IBM). *SPU Assembly Language Specification*, CBEA JSRE Series edition, September 2007.

[33] International Business Machines Corp. (IBM). *Synergistic Processor Unit Instruction Set Architecture*, January 2007.

[34] International Business Machines Corp. (IBM), Sony Computer Entertainment Inc., and Toshiba Corp. *Cell Broadband Engine Architecture*, October 2007.

[35] International Business Machines Corp. (IBM), Sony Computer Entertainment Inc., and Toshiba Corp. *Cell Broadband Engine Registers*, April 2007.

[36] International Business Machines, Inc. (IBM). Cell Broadband Engine resource center. Online: `http://www-128.ibm.com/developerworks/power/cell/`. Official site for Cell software developers. Retrieved: 08-03-27.

[37] International Business Machines, Inc. (IBM). IBM Full-System Simulator for the Cell Broadband Engine Processor. Online: `http://www.alphaworks.ibm.com/tech/cellsystemsim`. Retrieved: 08-03-08.

[38] International Business Machines, Inc. (IBM). *Cell Broadband Engine Programming Handbook*, April 2007.

[39] International Business Machines, Inc. (IBM). *IBM Full-System Simulator User's Guide. Modeling Systems based on the Cell Broadband Engine Processor*, September 2007.

[40] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[41] Khronos Group. Khronos Promoting Member Companies. Online: `http://www.khronos.org/members/promoters`. Retrieved: 08-03-31.

[42] Michael Kistler and Sidney Manning. *Debugging Cell Broadband Engine systems. Essential tools and techniques for the Cell BE software developer*. International Business Machines, Inc. (IBM), August 2006.

[43] Jörg Knitter. Konsolen-Linux. Ubuntu für PlayStation 3. Online: `http://games.magnus.de/konsole/artikel/ubuntu-fuer-playstation-3.html`. Retrieved: 08-04-03. (German).

[44] H. Köstler, M. Stürmer, Ch. Freundl, and U. Rüde. PDE based Video Compression in Real Time. Technical Report 07-11, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2007.

[45] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems). *LAPACK Working Note*, (175), July 2006.

[46] H.J. Lu, Kshitij Doshi, Rohit Seth, and Jantz Tran. Using Hugetlbfs for Mapping Application Text Regions. In John W. Lockhart, David M. Fellows, and Kyle McMartin, editors, *Proceedings of the Linux Symposium*, volume 2, pages 83–90, July 2006.

[47] Bruce D. Lucas and Takeo Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. In *Proc. Imaging Understanding Workshop*, pages 121–130, 1981.

[48] Michael Perrone. Cell BE Programming Gotchas! -or- 'Common Rookie Mistakes'. Technical report, IBM Research, 2005. Online: `http://www.cs.utk.edu/~dongarra/cell2006/cell-slides/18-Michael-Perrone.pdf`, Retrieved 08-03-03.

[49] Mike Acton. Better Performance Through Branch Elimination. Online: `http://www.cellperformance.com/articles/2006/07/tutorial_branch_elimination_pa.html`, July 2006. Retrieved: 08-03-26.

[50] Christian Persson, editor. *c't special Playstation 3*, volume 01/07. Heise Zeitschriften Verlag, Hannover, 2007. (German).

[51] Peter Seebach. Introducing the PowerPC SIMD unit. Technical report, IBM developerWorks, March 2005.

[52] Johannes Plötner and Steffen Wendzel. *Linux, Das distributionsunabhängige Handbuch*. Galileo Computing, 2nd edition, 2008. (German).

[53] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*, pages 135–136. Cambridge University Press, Third edition, 2007.

[54] Remy Bohmer. Howto: Install Fedora 8 on a Sony PS3. Online: `http://www.bohmer.net/ps3_stuff/install-fedora-8-on-PS3.html`. Retrieved: 08-03-27.

[55] Luc Saillard. Philips USB Webcam Driver for Linux. Online: `http://www.saillard.org/linux/pwc/`. Retrieved: 08-04-02.

[56] Sam Fuller. Motorola's AltiVec™ Technology. Technical report, Freescale Semiconductor, Inc., 1998.

[57] Sony Computer Entertainment Inc. Open Platform for PLAYSTATION®3. Online: `http://www.playstation.com/ps3-openplatform/index.html`. Official site for guest system support on the Playstation 3. Retrieved: 08-03-27.

[58] Sony Computer Entertainment Inc. *SPU C/C++ Language Extensions*, CBEA JSRE Series edition, October 2005.

[59] Sony Computer Entertainment Inc. How to Enable Your Distro. Online: `http://ps3.keshi.org/dsk/20061208/doc/HowToEnableYourDistro.html`, 2006. Official tutorial. Retrieved: 08-03-27.

[60] Andreas Stiller. *Prozessorgeflüster*, page 26. Heise Zeitschriften Verlag, Hannover, 2006. (German).

[61] Tungsten Graphics, Inc. Mesa Cell Driver. Online: `http://www.mesa3d.org/cell.html`. (Project webpage). Retrieved: 08-03-27.

[62] S. Uras, F. Girosi, A. Verri, and V. Torre. A computational approach to motion perception. *Biological Cybernetics*, 60(2):79–87, December 1988.

[63] Valgrind Developers. Supported Platforms. Online: `http://valgrind.org/info/platforms.html`, February 2007. Retrieved: 08-03-23.

[64] W. Hackbusch. *Multigrid Methods and Application*. Springer Verlag, Berlin, 1985.

[65] Christian Wagner. Introduction to Algebraic Multigrid. Universität Heidelberg, 1998/99. (Course notes).

[66] Joachim Weickert. Lecture notes to Differential Equations in Image Processing and Computer Vision, pages 17.3-17.11. Saarland University, 2007.

[67] Joachim Weickert and Christoph Schnörr. Variational Optic Flow Computation with a Spatio-Temporal Smoothness Constraint. Technical Report 15/2000, Computer Vision, Graphics, and Pattern Recognition Group, University of Mannheim, July 2000.

[68] English Wikipedia. History of video game consoles (fifth generation). Online: `http://en.wikipedia.org/wiki/History_of_video_game_consoles_%28fifth_generation%29`. Retrieved: 08-03-31.

[69] English Wikipedia. History of video game consoles (seventh generation). Online: `http://en.wikipedia.org/wiki/History_of_video_game_consoles_%28seventh_generation%29`. Retrieved: 08-03-31.

[70] English Wikipedia. History of video game consoles (sixth generation). Online: `http://en.wikipedia.org/wiki/History_of_video_game_consoles_%28sixth_generation%29`. Retrieved: 08-03-31.

[71] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20. ACM, 2006.

[72] P. Yeung, A. Torres, and P. Batra. Novel Test Infrastructure and Methodology Used for Accelerated Bring-Up and In- System Characterization of the Multi-Gigahertz Interfaces on the Cell Processor. *DATE 2007*, 2007.

[73] D. M. Young. *Iterative Solvers of Large Linear Systems*. Academic Press, New York, 1971.

[74] Andreas Zeller. *Why Programs Fail: A guide to systematic debugging*. Morgan Kaufmann Publishers and dpunkt.verlag Heidelberg, 2006.

Errata to: "Realtime Optical Flow Algorithms on the Cell Processor (Master's Thesis)"

Pascal Gwosdek

February 7, 2009

1 Introduction

Despite all endeavors and inspections, errors are made. In this document, I track the errata for my Master's Thesis "Realtime Optical Flow Algorithms on the Cell Processor".

2 Overview of errata

- **Page 45, line 18**

  "~~$E_\rho(u,v) = \int_\Omega w^T \left((K_\rho * J(\nabla_3 g))\, w + \alpha \left(|\nabla u|^2 + |\nabla v|^2\right)\right) dx\ dy.$~~"

  The outermost pair of parentheses is misplaced. The correction reads

  "$E_\rho(u,v) = \int_\Omega \left(w^T (K_\rho * J(\nabla_3 g))\, w + \alpha \left(|\nabla u|^2 + |\nabla v|^2\right)\right) dx\ dy.$"

- **Page 98, lines 16–17**

  "~~Whereever it is not explicitly stated, this variant is used throughout the experiments issued in this thesis.~~"

  The benchmarks presented refer to the setup performed on V cycles. With W cycles, frame rates up to about 135 FPS for 316×252 pixels on six SPUs are achieved. The correction hence reads

  "Whereever it is not explicitly stated, V cycles are used throughout the experiments issued in this thesis."