High Performance Parallel Optical Flow Algorithms on the Sony Playstation 3

Pascal Gwosdek, Andrés Bruhn, and Joachim Weickert

Mathematical Image Analysis Group,
Faculty of Mathematics and Computer Science, Building E1.1,
Saarland University, 66041 Saarbrücken, Germany
E-mail: {gwosdek,bruhn,weickert}@mia.uni-saarland.de

Abstract

In this paper, we evaluate the applicability of the Cell Processor built into the Sony Playstation 3 for variational optical flow computation. This is done by the example of the combined-local-global (CLG) method, a recent variational technique that requires to solve large and sparse linear systems of equations. Starting from efficient numerical algorithms for sequential architectures, we successively develop specifically adapted parallel variants for the Cell Processor. This includes the design of suitable numerical algorithms, the consideration of the specific memory and processor layout as well as the exploitation of instruction level parallelism (SIMD). The obtained results show that our parallelisation efforts pay off: With more than 170 dense flow fields of size 316×252 per second we achieve frame rates far beyond real-time.

1 Introduction

One of the main challenges in computer vision 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. Usually, the position and egomotion of the camera is unknown, such that only relative motion of objects with respect to the camera can be traced. The displacements between corresponding pixels in subsequent frames are then formulated in terms of a dense vector field, the *optical flow*.

Many qualitatively valuable algorithms to estimate these vector fields are based on variational methods. Such approaches compute the optical flow as minimiser of a global energy functional. Variational methods include classical techniques such as the ones of Horn and Schunck [13] and Nagel and Enkelmann [19] as well as recent high accuracy methods such as the combined local-global technique (CLG) of Bruhn *et al.* [7] and the level set approach of Amiaz and Kiryati [1]. While offering dense flow fields of high quality, variational methods require to solve large linear or nonlinear systems of equations. This in turn makes them less appealing for real-time applications, since the associated computational workload is high.

In the past, several numerical techniques have been investigated to accelerate such algorithms. Among them are fast iterative solvers based on the Successive Over-Relaxation (SOR) idea [23], preconditioned conjugate gradient methods [20] as well as uni- and bidirectional multigrid techniques [3, 4]. For smaller image sizes, some of these methods are already suited to compute flow fields on standard PCs in realtime [5].

However, as soon as a higher resolution is required or the algorithms are embedded into more complex application frameworks, the guaranteed performance does often not suffice to ensure real-time behaviour for the entire system as well. Modern parallel processor architectures promise a remedy to this problem: By providing many cores, they allow to distribute the workload on many shoulders and thus to improve the performance. In this context, Mitsukami and Tadamura [18], Zach *et al.* [24], as well as Grossauer and Thoman [12] have demonstrated that optical flow algorithms can even be efficiently computed in parallel using current graphics hardware. Moreover, Kalmoun *et al.* [9] have recently developed implementations of optical flow techniques for massively parallel computer clusters.

Another increasingly popular parallel computing platform is given by Sony's Playstation 3. It is nowadays one of the cheapest parallel computers available on the market, and is simultaneously prospecting impressive speedups in comparison to traditional architectures. The Cell Broadband Engine, a novel processor designed for the application in this device, has already been used to solve many scientific problems: Besides rather simple tasks such as matrix multiplications [8] and fast Fourier transformations [22], recently also more sophisticated algorithms like raytracing [2], video compression based on partial differential equations [16], and physical simulations for the Stokes equation [10] have been successfully implemented. Since they aim for the efficient solution of large systems of equations, the latter two approaches are closest in spirit to our work presented in this paper.

To the best of our knowledge the computation of dense optical flow fields with variational methods on a Playstation 3 has not been considered so far in the literature. The goal of our paper is thus to evaluate the applicability of the parallel architecture of this device. To this end, we propose two approaches to adapt optical flow algorithms to the new architecture and show that these solutions are indeed capable to speed up the process significantly: For images of size 316×252, up to 170 dense optic flow fields per second can be computed.

Our paper is organised as follows. In Section 2, we give a short review of the CLG method and discuss the obtained linear system of equations that has to be solved. For this task we then present two parallel implementations in Sections 3 and 4: one based on a red-black SOR variant and another one that makes use of a highly efficient full multigrid scheme. Both algorithms are evaluated in Section 5 with respect to their performance and scalability. A summary in Section 6 concludes this paper.

2 The CLG Method

In order to investigate the applicability of Sony's Playstation 3 for variational optical flow computation, we have focused on the implementation of the linear variant of the combined local-global (CLG) method of Bruhn *et al.* [5, 7]. Combining the dense flow fields of the global Horn/Schunck method [13] with the noise robustness of the local Lucas/Kanade technique [17], this approach offers a good tradeoff between accuracy and computational complexity. Moreover, like other recent variational techniques this method requires the solution of a large sparse linear (or nonlinear) system of equations. Thus, parallel implementations of the CLG method can be expected to reliably predict potential speedups for other variational algorithms being ported to the architecture of the Cell processor as well.

2.1 Variational Model

Let $f(x,y,t)$ be a grey value image sequence, where (x,y) denotes the location within a rectangular image domain and t is the time. Moreover, let $f_\sigma(x,y,t) = K_\sigma * f(x,y,t)$ be the spatially presmoothed counterpart of f obtained by a convolution with a Gaussian K_σ of standard deviation σ.

Then the linear 2-D CLG method computes the optical flow $\mathbf{w}(x,y) = (u(x,y), v(x,y), 1)^\top$ as minimiser of the energy functional [7]

$$E(u,v) = E_{\mathrm{D}}(u,v) + \alpha\, E_{\mathrm{S}}(u,v) \tag{1}$$

with data and smoothness term

$$E_{\mathrm{D}}(u,v) = \int_\Omega \Big(\mathbf{w}^\top J_\rho(\nabla f_\sigma)\, \mathbf{w} \Big)\, dx\, dy, \tag{2}$$

$$E_{\mathrm{S}}(u,v) = \int_\Omega \left(|\nabla u|^2 + |\nabla v|^2 \right) dx\, dy. \tag{3}$$

Here, $\nabla u = (u_x, u_y)^\top$ represents the spatial flow gradient, $\nabla f_\sigma = (f_{\sigma_x}, f_{\sigma_y}, f_{\sigma_t})^\top$ stands for the spatiotemporal image gradient and

$$J_\rho(\nabla f_\sigma) := K_\rho * \Big(\nabla f_\sigma\ \nabla f_\sigma^\top \Big) \tag{4}$$

denotes the entry-wise spatially convolved motion tensor for the linearised grey value constancy assumption [6].

While the integration scale ρ of the motion tensor mainly affects the robustness under noise, the weight $\alpha > 0$ serves as a regularisation parameter that steers the degree of smoothness of the solution.

2.2 Minimisation and Discretisation

In order to minimise the energy functional in (1), we have to solve the associated *Euler-Lagrange equations* [11]. They are given by

$$0 = \Delta u - \frac{1}{\alpha}\left(J_{\rho_{11}}\, u + J_{\rho_{12}}\, v + J_{\rho_{13}} \right), \tag{5}$$

$$0 = \Delta v - \frac{1}{\alpha}\left(J_{\rho_{12}}\, u + J_{\rho_{22}}\, v + J_{\rho_{23}} \right), \tag{6}$$

with reflecting Neumann boundary conditions

$$0 = \mathbf{n}^\top \nabla u \quad \text{and} \quad 0 = \mathbf{n}^\top \nabla v \, . \tag{7}$$

Here, $\mathbf{n}$ denotes the normal vector perpendicular to the image boundary, and $J_{\rho_{nm}}$ is n, m-th component of the convolved motion tensor $J_\rho (\nabla f_\sigma)$.

Since these equations have to be solved numerically, we discretise them on a grid of size $N_x \times N_y$ with cell spacing $h_x \times h_y$. Thereby input frames and flow field components are sampled at the grid points (i, j) with $1 \le i \le N_x$ and $1 \le j \le N_y$. While the spatial derivatives for setting up the discrete motion tensor entries $[J_{\rho_{nm}}]_{i,j}$ are approximated by a fourth-order finite difference scheme evaluated between the frames, the temporal derivatives are computed using a simple two point stencil. If we denote the four-neighbourhood of a pixel (i, j) in direction of axis l by $\mathcal{N}_l(i, j)$, we are finally in the position to write down the discrete variant of the Euler-Lagrange equations in (5)-(6). They read

$$\begin{aligned} 0 \; = \; & [J_{\rho 11}]_{i,j}\, u_{i,j} + [J_{\rho 12}]_{i,j}\, v_{i,j} + [J_{\rho 13}]_{i,j} \\ & - \alpha \sum_{l \in \{x,y\}} \; \sum_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l(i,j)} \frac{u_{\tilde{i},\tilde{j}} - u_{i,j}}{h_l^2} \, , \end{aligned} \tag{8}$$

$$\begin{aligned} 0 \; = \; & [J_{\rho 12}]_{i,j}\, u_{i,j} + [J_{\rho 22}]_{i,j}\, v_{i,j} + [J_{\rho 23}]_{i,j} \\ & - \alpha \sum_{l \in \{x,y\}} \; \sum_{(\tilde{i},\tilde{j}) \in \mathcal{N}_l(i,j)} \frac{v_{\tilde{i},\tilde{j}} - v_{i,j}}{h_l^2} \, , \end{aligned} \tag{9}$$

for $i = 1, \ldots, N_x$ and $j = 1, \ldots, N_y$ which actually forms a linear system of equations with respect to the $2 N_x N_y$ unknowns $u_{i,j}$ and $v_{i,j}$.

2.3 Implementation

For solving this linear system of equations, we developped two parallel implementations that are discussed in detail in the following two sections. They are specifically tailored for the Cell Broadband Engine processor built into the Playstation 3 video console. This multi-core processor features, besides a general-purpose core, eight 'synergistic' special-purpose units, the *SPUs*, which are natively operating on a bandwidth of 128 bits. They own combined data and instruction caches, each 256 kB of size, which can be used for fast working copies of the problem residing in RAM. These *Local Stores* need to be explicitly filled and synchronised to the shared memory using dedicated DMA operations [15].

This new hardware setup is challenging: It requires a proper synchronisation among the cores, the design of an efficient distribution strategy of algorithmic needs to the SPUs, an evolved DMA scheduling, and an optimisation that exploits the instruction level parallelism in terms of SIMD vectors. Only then the full potential of the Cell processor can be exploited.

3 Red-Black SOR Solver

As a first algorithm for parallelisation, we consider the Successive Over-Relaxation (SOR) method [23]. It is an extrapolation variant of the classical Gauß-Seidel solver that significantly accelerates the convergence. The corresponding update rule for pixel (i, j) from iteration step k to $k+1$ is given by equations (10)-(11). Here, $\omega \in (0, 2)$ is the overrelaxation weight that strongly influences the convergence speed, and $\mathcal{N}_l^-(i, j)$ and $\mathcal{N}_l^+(i, j)$ are the neighbours of pixel (i, j) in direction of axis l that have already been updated or must still be updated in the current iteration step, respectively.

To distribute the computational load to different cores, the domain needs to be suitably decomposed. One common approach well-known in the literature is the so-called red-black reordering scheme: The image domain Ω is hereby split into a checkerboard pattern of 'red' and 'black' pixels, which are then processed in a colour-wise manner. In the case of discretisations that involve only the four direct neighbours – such as in our case – this strategy offers a particular advantage: 'Red' elements only depend on a 'black' neighbourhood and vice versa. As a consequence, each colour can be handled fully in parallel. However, after one colour is entirely processed involving multiple cores, those need to be synchronised with each other to re-establish dependencies between differently coloured elements.

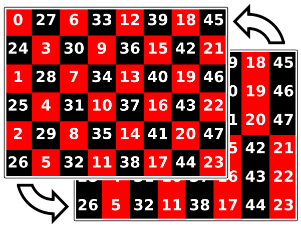

Figure 1: Red-black decomposition and reordering.

$$u_{i,j}^{k+1} = (1-\omega)\, u_{i,j}^{k} + \omega \frac{[J_{13}]_{i,j} + [J_{12}]_{i,j}\, v_{i,j}^{k} - \alpha \left(\sum\limits_{l\in\{x,y\}} \sum\limits_{(\tilde{i},\tilde{j})\in\mathcal{N}_l^-(i,j)} \frac{u_{\tilde{i},\tilde{j}}^{k+1}}{h_l^2} + \sum\limits_{l\in\{x,y\}} \sum\limits_{(\tilde{i},\tilde{j})\in\mathcal{N}_l^+(i,j)} \frac{u_{\tilde{i},\tilde{j}}^{k}}{h_l^2} \right)}{-[J_{11}]_{i,j} - \alpha \sum\limits_{l\in\{x,y\}} \sum\limits_{(\tilde{i},\tilde{j})\in\mathcal{N}_l(i,j)} \frac{1}{h_l^2}} \tag{10}$$

$$v_{i,j}^{k+1} = (1-\omega)\, v_{i,j}^{k} + \omega \frac{[J_{23}]_{i,j} + [J_{12}]_{i,j}\, u_{i,j}^{k+1} - \alpha \left(\sum\limits_{l\in\{x,y\}} \sum\limits_{(\tilde{i},\tilde{j})\in\mathcal{N}_l^-(i,j)} \frac{v_{\tilde{i},\tilde{j}}^{k+1}}{h_l^2} + \sum\limits_{l\in\{x,y\}} \sum\limits_{(\tilde{i},\tilde{j})\in\mathcal{N}_l^+(i,j)} \frac{v_{\tilde{i},\tilde{j}}^{k}}{h_l^2} \right)}{-[J_{22}]_{i,j} - \alpha \sum\limits_{l\in\{x,y\}} \sum\limits_{(\tilde{i},\tilde{j})\in\mathcal{N}_l(i,j)} \frac{1}{h_l^2}} \tag{11}$$

3.1 Data Structures and Caching

On the Cell processor, we particularly split the rectangular image domain into stripes of comparable width, and each of these stripes is processed by exactly one SPU. A region thereby owns a composition of inner and outer boundaries, whereof the first class refers to transitions between stripes, and latter are just those coinciding with image boundaries. Both types need a different treatment, namely are inner ones to be protected with one synchronisation step for any neighboring stripe pair, colour and iteration step, and outer boundaries are just to be updated according to their role in the global problem.

We furthermore make the red-black reordering explicit by agglomerating equally coloured elements to large memory blocks, thereby conforming the data to SIMD operations (cf. Figure 1). Since many solver iterations are usually needed to let the solution converge and scalar operations are much more expensive than vector-valued ones, this step amortises well. Meanwhile, working on reordered data turns out to be rather hindering during the setup of the equation system: Derivative approximations and presmoothing convolutions are by concept best suited for ordered data, though only little optimisation with respect to instruction level parallelism can be applied this way. Therefore, the equation system is first set up based on a canonical ordering, then shuffled with respect to the red-black scheme, processed by several solver iterations, and finally back-transformed into the initial representation. DMA transfers between RAM and Local Store can thus be performed on larger data chunks, and thereby benefit from faster memory accesses.

An additional concept to hide the memory latency is a technique called *double buffering* [14], which is also intensively used in our algorithm: DMA operations are primarily handled by the memory flow controller, while the SPU itself can still be working on independent data sets. Instead of working on a single instance of a local copy, one hence considers two versions, whereof one is synchronised with the RAM while the other is held valid for the algorithm and vice versa.

By using ring buffers with more than two vector-valued elements, this concept can be generalised for applications involving a whole neighbourhood to be supplied to the algorithm: Several adjacent vectors are hereby kept valid, while one or more vectors in the background are involved in DMA operations. Whenever the algorithm needs to access one new vector, the buffer is rotated. This concept often represents a good compromise between a low cache load and less redundant RAM accesses, and is thus frequently applied in our algorithm.

3.2 Algorithmic Optimisations

Due to their native bandwidth of 128 bits, SPUs are particularly suited to process SIMD vectors, and scalar code fragments are comparably expensive. Thanks to the reordered data structure, instruction level parallelism can easily be established for the solver iterations. In fact, even the reordering steps themselves can be formulated in a SIMD based manner, thereby gaining an additional factor of 1.78 compared to a variant only involving memory level parallelism.

In contrast, the setup of the equation system as well as the optional presmoothing and integration steps require more attention, since they are dominated by convolutions. Those are separable, and the instruction level parallelisation options vary depending on whether the direction of the convolution matches the direction of the elements in memory or not: Along the memory direction, a high number of expensive misaligned vector loads would be necessary, which are in general much slower than a simple scalar notation. In the perpendicular direction, however, blocks of adjacent memory are computationally independent and thus fully parallelisable.

Latter approach requires a special treatment of outer boundaries: Since branching is very expensive, it often pays off to pad the major memory direction sufficiently, to process all elements in parallel regardless of their role, and to later correct boundary values on a scalar basis. This way, no branching is introduced at the boundaries at all, which accelerates the program noticeably: The branch misprediction penalty is with 18 to 19 cycles rather high and dynamic branch prediction is not supported on the hardware [14]. Hence, about as much latency would in the average case be expected for every processed line, instead of about three cycles in our implementation.

Furthermore, a high padding of the inner matrix dimension in the cached working copy is worthwhile to significantly reduce pipeline latencies: The SPUs own two pipelines, one of which is exclusively used for load and store operations, and one for arithmetics [14]. By explicitly encoding several operations in parallel performing a loop unrolling strategy, bubbles in the pipeline can be reduced to a minimum. On padded datasets, we can eventually again formulate this algorithm without any introduction of branching.

Since only one kernel can be run on an SPU at a time, and loading a new kernel to the device is expensive, we created one monolithic SPU kernel per core. It is started once at the beginning of the computations, and is instantly charged with new problem sets whenever a new pair of frames is available.

4 Full Multigrid Solver

The second algorithm that we consider for parallelisation is the so called *Full Multigrid* algorithm. Representing one of the most powerful classes of numerical solvers for linear and nonlinear systems of equations, these methods enjoy great popularity in computer vision. In [5, 6], Bruhn *et al.* have demonstrated that such techniques allow to compute the optical flow of an image sequence in realtime, while recently Grossauer and Thoman ported these methods on a graphics processing unit to accelerate them even further [12]. In the following, we present our own Full Multigrid implementation that is specifically tailored for the Cell processor. It is based on the sequential algorithm in [5].

4.1 Algorithmic Sketch

In order to understand the arithmetic operations involved, let us start by a short sketch of the main strategy of multigrid algorithms. These methods try to overcome the slow attenuation of low-frequent errors by classical iterative solvers, by using a sophisticated error correction strategy. Its main idea can be summarised in four steps [4, 21]:

1) Perform η_1 presmoothing relaxation steps with a basic iterative solver (e.g. Gauß-Seidel).
2) Solve residual equation system on coarser grid to obtain correction. This requires intergrid transfer operators (restriction/prolongation).
3) Correct fine grid solution by the computed coarse grid error.
4) Perform η_2 presmoothing relaxation steps with a basic iterative solver (e.g. Gauß-Seidel).

In general, such a two-grid cycle is applied in a hierarchical/recursive way to improve the performance. The Full Multigrid solver extends this strategy by an additional coarse-to-fine strategy. Starting from a very coarse representation, the original problem is successively refined. Thereby solutions from coarser grids serve as initialisations for finer ones. For a more detailed algorithmic description of this numerical scheme applied to the CLG method we refer to [5]. For our purpose, it is sufficient to note that we use the Gauß-Seidel technique with $\eta_1 = \eta_2 = 1$ as basic solver, make one recursive call per level (V-Cycle) and apply area-based resampling to transfer the data between the grids. In the following we will denote this method by $V(1,1)$.

4.2 Parallel Setup

In this context, a decomposition of the image domain like pursued for the Red-Black SOR method (cf. Subsection 3.1) is nontrivial, since several grids of different dimensions need to be considered. However, a similar spatial distribution on all grids is likely to fail, because the synchronisation related overhead amortises well on fine grids, but becomes serious on coarser scales. On the other hand, it is also not advisable to reduce the number of cores processing the problem while working on coarser grids, since computing resources are meanwhile dissipated.

Instead, we can turn the rather low number of parallel cores the Playstation 3 provides to an advantage: Since typically, longer input streams are provided to the system, consecutive pairs of frames can be sent to single SPUs. Those process them individually, but locally optimised for the hardware.

At the cost of a higher latency, this parallelisation technique performs at least as well as a potential decomposition scheme, since over arbitrary time intervals, the same number of frames can be processed. Moreover, SPU kernels are allowed to run asynchronously, which reduces access bursts on the Memory Interface Controller and the Element Interconnect Bus and thus enhances the performance significantly.

4.3 Data Structures and Caching

Though an explicit memory reordering like pursued in the SOR setting would still be beneficial to accelerate Gauß-Seidel pre- and postrelaxations, the structure of the algorithm counteracts this scheme: Such reordered datasets cannot be immediately transferred to different grid sizes, since potential mappings like the one proposed by Trottenberg *et al.* [21] are not colour-preserving. Indeed, it turns out that optimisations with respect to the relaxation steps instantly cause a higher effort to perform grid transitions, such that a canonically ordered grid seems to be the best solution.

By a strict separation of cache-related and arithmetics-oriented instructions, we yield smaller kernel binaries and thus a higher flexibility with respect to the data partition in the shared Local Store, and are furthermore able to easily design more complex and thus efficient cache management algorithms: Predicting the next requests of the respective modules of the algorithm, dedicated functions fetch vectors early enough to have a valid copy cached when the requests actually occur, and make hence active use of ring buffers not to lose time due to memory latencies. Thereby, they automatically reuse cached vectors when they are needed a second time, and treat boundaries correctly while still minimising the amount of data requested from RAM.

4.4 Algorithmic Efficiency

Based on the canonically ordered grid, Gauß-Seidel steps, like they are used for pre- and postrelaxation and for the computation of the residual, are most efficiently written in a fully-sequential manner: The memory access patterns generated this way are better suited to the way the Local Store is organised, and thereby reduce DMA synchronisations to RAM to a minimum.

On the other hand, the remaining modules of the algorithm allow for a higher degree of instruction level parallelism: Restriction and prolongation operators are realised by area-based resampling [5] and thus separable. The same holds for the convolutions involved in the image presmoothing as well as in the local integration of the motion tensor. In particular, this means that perpendicular to the direction of operation, the elements involved in these schemes are pairwise independent.

This observation allows not only for loop unrolling approaches to minimise pipeline latencies, but coinciding with the major memory direction, independent values are residing in adjacent memory cells, such that at least in this direction, the full SIMD width can be used. Like for the SOR solver, a sufficiently high padding of the image domain eliminates branches in the algorithm, and thus allows for highly efficient code.

5 Results

In the following, we evaluate the runtime performance of the two implemented methods with respect to different criteria, and compare them to a sequential implementation on a Pentium 4 with 3.2GHz. The measurements given for the following experiments represent the achieved number of *frames per second* (FPS). This provides an intuitive measure for real world applications. The times are

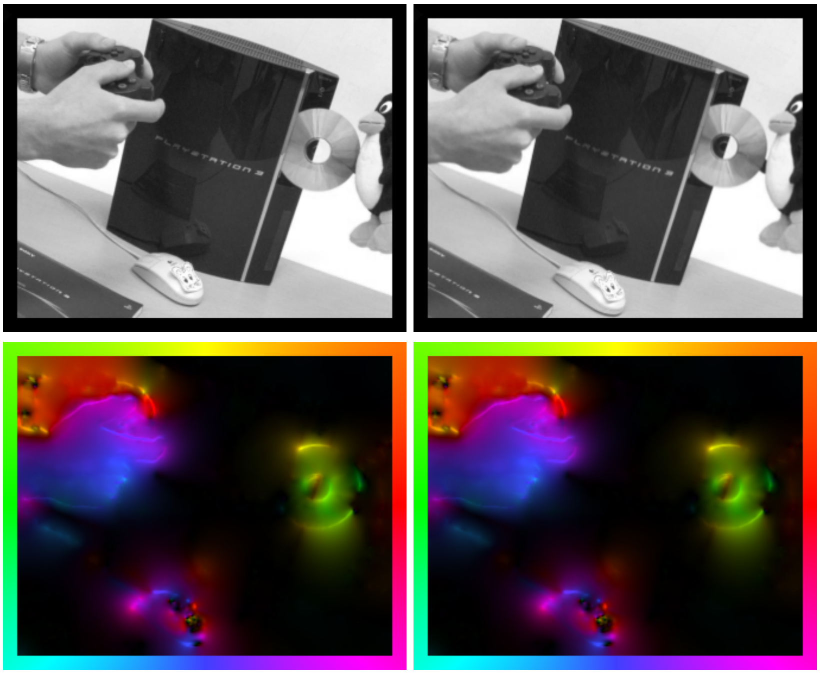

Figure 2: Visual quality of computed flow fields. **Top Row:** Frames 1 and 2 of the *Flying Penguin* sequence (316 $\times$ 252). **Bottom Row**: Flow field for the CLG method ($\sigma = 0.30$, $\rho = 0.55$, $\alpha = 1000$) using 100 Red-Black SOR iterations, and Full Multigrid with one V(1,1)-cycle per level, respectively. The magnitude of each displacement is encoded by brightness, the direction by colour (as shown at the boundaries).

always measured over the whole algorithm, including the equation system setup, optional presmoothing, parallelisation and hardware related expenses as well as the iterative solver. Because only six out of eight SPUs are available on the Playstation 3, we consider this setup as a maximal configuration. However, the Cell processor is also marketed on special boards on which all eight cores are accessable, such that even higher speedups are to be expected in such scenarios.

5.1 Red-Black SOR

In our first experiment, we evaluate the runtime of the parallel Red-Black SOR implementation against a sequential reference setup on the Pentium 4 with 3.2 GHz, and analyse the scaling of the algorithm over varying SPU counts. In an ideal setting, the processed number of frames should be linearly correlated to the number of cores involved, such that the total performance only depends on local optimisations on the single cores.

Table 1 shows the obtained scaling behaviour. With six SPUs, and frames rates of up to 25 dense flow fields per second for images of size 315×252, we are able to achieve realtime performance, and beat the sequential implementation by a factor of 5.2. Meanwhile, this experiment also reveals an attenuation of the performance for a higher number of cores as can be seen from Figure 3. This attenuation is related to the relatively small partitions of data processed by one SPU in between two consecutive synchronisation steps, and to bus contentions on the memory flow controller.

In our second experiment, we are interested in the performance of the Red-Black SOR algorithm over varying frame sizes. To make justified statements about this behaviour when applying presmoothing, we scale the variances according to the edge length ratio of the considered images, such that we use $(\sigma, \rho) = (0.075, 0.1375), (0.15, 0.275), (0.3, 0.5)$ and $(0.6, 1.1)$ for frames of 79×63, 158×126,

Table 1: Scaling behaviour of our Red-Black SOR implementation with 100 iterations for an increasing number of SPUs. Performance is measured in flow fields per second and refers to images of size 316×252.

	Number of SPUs					
Red-Black SOR	**1**	**2**	**3**	**4**	**5**	**6**
CLG ($\sigma = 0.00$, $\rho = 0.00$)	6.29	12.06	17.57	20.19	24.07	24.76
CLG ($\sigma = 0.30$, $\rho = 0.55$)	4.08	7.71	10.98	13.04	15.36	15.61

Table 2: Performance of the Red-Black SOR implementation with six SPUs for different frame sizes.

	Frame Size			
Red-Black SOR	**79 × 63**	**158 × 126**	**316 × 252**	**632 × 504**
CLG ($\sigma = 0.00$, $\rho = 0.00$)	160.84	85.66	24.76	5.97
CLG ($\sigma = 0.30$, $\rho = 0.55$)	133.91	62.34	15.61	3.16

316×252, and 632×504 pixels, respectively.

The obtained frame rates are listed in Table 2. Though the maximal computable problem size is restricted by the rather poor RAM equipment of the Playstation 3 and related swapping processes for larger problems, our method seems to scale very well over different frame sizes and does not show preferences towards particularly large or small problems.

Figure 2 (c) shows the computed flow field for the Playstation 3 sequence of size 316 × 252 with parameters $\sigma = 0.3$ and $\rho = 0.55$. As one can see, the estimated motion looks reasonable. In this context, one should note that since the present parallel implementation entirely follows the linear 2-D CLG model, its results are equivalent to those achieved in the sequential algorithm from [5]. For the popular Yosemite sequence without clouds we achieve for

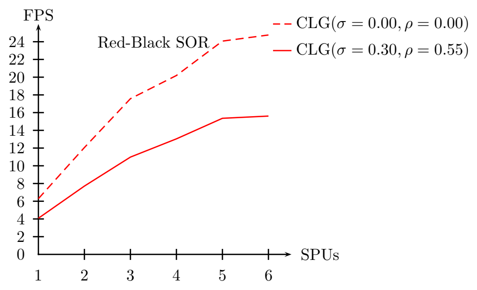

Figure 3: Scaling behaviour of the Red-Black-SOR implementation with 100 iterations for an increasing number of SPUs (images of size 316 × 252).

instance the same average angular error of 2.63°.

5.2 Full Multigrid

For the Full Multigrid method, we repeat the experiments performed with the SOR solver, and again compare the achieved results to a sequential solution. As before, our first experiment is dedicated to the scaling behaviour of the algorithm for an increasing number of SPUs. As one can see from Table 3 as well as from Figure 4, the novel concept indeed pays off, and our method scales almost perfectly linear over an increased number of cores. This is, because the algorithm is both no longer affected by synchronisation-related stalls, and moreover related to the lax timing of memory requests, which automatically schedule in the least conflicting way. Compared to the sequential solution, we achieve a speedup by a factor of 5.3. In particular are bus contentions diminished as well, since the single SPU kernels can run asynchronous to each other, such that DMA requests are issued sequentially and thus do not interfere.

In our second experiment, we again evaluate the development of the frame rate for different image sizes. This time, the maximal image size to fit entirely into RAM is even more constrained than for Red-Black SOR, since six independent problems need to be simultaneously held in the memory. However, for the maximum frame size of 316×252, we achieve an excellent average performance of up to 170.15 FPS, which is far beyond realtime. This way, we are even able to outperform fast algorithms

Table 3: Scaling behaviour of our Full Multigrid algorithm with 1 V(1,1)-cycle per level for an increasing number of SPUs. Performance is measured in flow fields per second and refers to images of size 316×252.

	Number of SPUs					
Full Multigrid	**1**	**2**	**3**	**4**	**5**	**6**
CLG ($\sigma = 0.00$, $\rho = 0.00$)	30.46	60.29	89.65	118.65	144.85	170.15
CLG ($\sigma = 0.30$, $\rho = 0.55$)	19.89	39.59	58.12	76.17	93.99	112.71

Table 4: Performance of the Full Multigrid implementation with six SPUs for different frame sizes.

	Frame Size		
Full Multigrid	**79×63**	**158×126**	**316×252**
CLG ($\sigma = 0.00$, $\rho = 0.00$)	1572.14	495.48	170.15
CLG ($\sigma = 0.30$, $\rho = 0.55$)	1198.06	417.52	112.71

running on graphics cards, like the one proposed by Zach *et al.* [24], by more than a factor of five. Since our algorithm is not constrained to be run on a Playstation 3, an even higher performance can be achieved when it is executed in setups with all eight SPUs available. The scaling behaviour over different frame sizes even promises realtime performance for frames of about 1024×768 pixels, if enough memory space can be granted to the algorithm.

The computed flow field for the Playstation 3 sequence is depicted in Figure 2 (d). Despite the much higher frame rate, there are no visual differences to the result of the Red-Black SOR method. This is confirmed by checking our algorithm once again with the Yosemite sequence without clouds. Not surprisingly, we achieve also this time an average angular error of 2.63° as stated in [5].

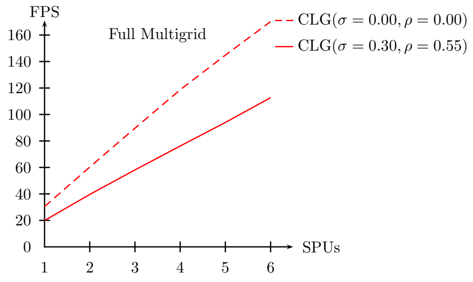

Figure 4: Scaling behaviour of our Full Multigrid implementation with 1 V(1,1)-cycle per level for an increasing number of SPUs (image size 316×252).

6 Summary

In this paper, we have presented two parallel high performance algorithms for the optical flow problem running on the Cell processor of a Playstation 3. Combining smart memory layouts, highly efficient numerics and optimisations to the hardware, we yield speedups of about a factor of 5.3 compared to traditional hardware. In terms of frame rates, this equals an astonishing performance of up to 170 dense flow fields per second on image sequences of size 316×252.

This shows that optical flow methods relying on highly efficient numerics can even be accelerated further, if recent hardware is effectively exploited for parallelisation. Significant speedups can also be expected for models with nonquadratic penalisation. This however is subject of ongoing work. We hope that our paper will motivate more computer vision researchers to exploit the fascinating potential of the Cell processor.

References

[1] T. Amiaz and N. Kiryati. Piecewise-smooth dense optical flow via level sets. *International Journal of Computer Vision*, 68(2):111–124, 2006.

[2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the CELL processor. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pages 25–23, Salt Lake City, UT, 2006. IEEE Computer Society Press.

[3] F. Bornemann and P. Deuflhard. The cascadic multigrid method for elliptic problems. *Numerische Mathematik*, 75:135–152, 1996.

[4] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, Apr. 1977.

[5] 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.

[6] A. Bruhn, J. Weickert, T. Kohlberger, and C. Schnörr. A multigrid platform for real-time motion computation with discontinuity-preserving variational methods. *International Journal of Computer Vision*, 70(3):257–277, Dec. 2006.

[7] A. Bruhn, J. Weickert, and C. Schnörr. Lucas/Kanade meets Horn/Schunck: Combining local and global optic flow methods. *International Journal of Computer Vision*, 61(3):211–231, 2005.

[8] A. Buttari, P. Luszczek, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the Playstation 3. Technical Report UT-CS-07-595, University of Tennessee Knoxville, May 2007.

[9] M. El Kalmoun, H. Köstler, and U. Rüde. 3D optical flow computation using a parallel variational multigrid scheme with application to cardiac C-arm CT motion. *Image and Vision Computing*, 25(9):1482–1494, 2007.

[10] M. R. Elgersma, D. A. Yuen, and S. G. Pratt. Adaptive multigrid solution of Stokes' equation on CELL procesor. *Eos Transactions AGU*, 87(52), Dec. 2006. Fall Meet. Suppl., Abstract IN53B-0821.

[11] L. E. Elsgolc. *Calculus of Variations*. Pergamon, Oxford, 1961.

[12] H. Grossauer and P. Thoman. GPU-based multigrid: Real-time performance in high resolution nonlinear image processing. In A. Gasteratos, M. Vincze, and J. K. Tsotsos, editors, *Computer Vision Systems*, volume 5008 of *Lecture Notes in Computer Science*, pages 141–150. Springer, Berlin, 2008.

[13] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

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

[15] 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):589–604, 2005.

[16] H. Köstler, M. Stürmer, C. Freundl, and U. Rüde. PDE based video compression in real time. Technical Report 07-11, Institut für Informatik, Univ. Erlangen-Nürnberg, Germany, 2007.

[17] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proc. Seventh International Joint Conference on Artificial Intelligence*, pages 674–679, Vancouver, Canada, Aug. 1981.

[18] Y. Mitsukami and K. Tadamura. Optical flow computation on Compute Unified Device Architecture. In *Proc. 14th International Conference on Image Analysis and Processing*, pages 179–184. IEEE Computer Society Press, 2007.

[19] H.-H. Nagel and W. Enkelmann. An investigation of smoothness constraints for the estimation of displacement vector fields from image sequences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:565–593, 1986.

[20] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, second edition, 2003.

[21] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, San Diego, 2001.

[22] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *CF '06: Proc. ACM Conference on Computing Frontiers*, pages 9–20, Ischia, Italy, 2006.

[23] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.

[24] C. Zach, T. Pock, and H. Bischof. A duality based approach for realtime TV-L^1 optical flow. In F. A. Hamprecht, C. Schnörr, and B. Jähne, editors, *Pattern Recognition*, volume 4713 of *Lecture Notes in Computer Science*, pages 214–223. Springer, Berlin, 2007.