Parallel Implementation of Real-Time Block-Matching based Motion Estimation on Embedded Multi-Core Architectures

Nicolai Behmann, Oliver Jakob Arndt and Holger Blume
Leibniz Universität Hannover, Institute of Microelectronic Systems
Appelstr. 4, 30167 Hanover, Germany
{behmann, arndt, blume}@ims.uni-hannover.de

Abstract—Considering the strict demands of video-based advanced driver-assistance systems in terms of real-time execution, complex applications are usually realized with dedicated hardware solutions. Indeed, modern vector-accelerated multi-core processors, serving as attractive off-the-shelf components, feature increasing computational performance, while executing flexible and maintainable software code. However, as the processor characteristics (e.g., vector-instruction-set, NoC topology) vary between different chips, the parallelization and vectorization benefit depends on the target platform. In this work, we present a parallel real-time implementation of a predictive block-matching based motion estimation algorithm using parallelization methods on data and task level. We target our implementation on the Samsung Exynos 5 Octa mobile ARM big.LITTLE multi-core processor, exhibiting NEON SIMD vectorizations. While we illustrate our implementation strategies in detail, we specifically point out representative differences of the NEON instruction-set extension, compared with the SSE vector operations, common in x86 based architectures. Furthermore, to analyze and compare the resulting parallelization and vectorization benefits, as well as the scalability over threads, we also consider an Intel Core-i7 general-purpose processor as a reference platform.

I. INTRODUCTION

Motion estimation from image sequences is an essential preprocessing step in different fields, such as video processing, video coding and multimedia applications as well as ego-motion estimation in video-based driver assistance systems. Appropriate algorithms are often based on block-matching, combining complex data flows and dependencies with a huge amount of memory-intensive block similarity calculations.

The requirements concerning real-time processing, combined with low latency and high data throughput, are often fulfilled with the use of dedicated or customized hardware architectures. For prototyping purposes, these accelerators appear as inconvenient in terms of development costs and flexibility. Through the emerging smartphone market mobile multi-core processors deliver increasing computing capabilities, while featuring small power demands and product costs. Embedded multi-core SoCs profit from a bigger memory bandwidth and additional SIMD units on a single die, offering high data parallelism at minimum power consumption. Moreover, the ability of high-level programming languages reduces the time to evaluate the performance of the algorithm and offers portability and maintainability, while reducing the development costs and time-to-market. However, efficiently exploiting the SoC resources (e.g., processor, memory, SIMD units) and implementing complex parallelization methods demand experienced programmers, avoiding common concurrency errors, such as false sharing, dead locks or race conditions. Especially, the use of dedicated SIMD units, require specific programming instruments, which differ in the resulting speedup and amount of required code manipulations. As some algorithmic problems exhibit operations or data types, which do not offer a sufficient theoretical vectorization speedup, but the most efficient vectorization methods require an exaggerated coding effort, the tradeoff between invested manpower and speedup benefit should be consciously calculated before migrating the software. Therefore, this paper investigates several parallelization strategies and compares their coding effort and vectorization benefit. Combining different parallelization methods, the interferences and influences on the resulting scaling behavior across multiple processors need to be taken into account to generate the optimal tradeoff between hardware utilization and algorithmic acceleration. As the scaling behavior over threads depends on the architecture specific system interconnection network and capabilities of the particular processing units, we compare these parameters between the embedded multi-core processor Samsung Exynos 5 Octa, featuring ARM big.LITTLE architecture, and the general-purpose Intel Core-i7 quad-core processor. The latter features hyperthreading technology, enabling two independent hardware threads with separate register sets, to be executed on the same core.

In this work, we introduce a parallel implementation of an exemplary motion estimation pipeline, consisting of the Parallel Predictive Block-Matching (PPBM) algorithm [1] and a vector field erosion [2]. The implementation targets mobile multi-core and general-purpose platforms, satisfying real-time requirements in advanced driver assistance systems. We evaluate the platform performance of a low power SoC, featuring ARM big.LITTLE architecture, in comparison with a reference Intel Core-i7. Specifically, this work makes the following major contributions:

- Parallelized and vectorized implementation of the PPBM motion estimation algorithm and erosion
- In-depth analysis and comparison of vectorization strategies and instruction-set extensions
- Evaluation of the parallel scalability over threads, describing the influences and interferences of combined data- and task-level parallelization methods

The remainder of this paper is organized as follows: The following Section reviews the related work on algorithms and migration strategies to multi-core architectures. The general concept of the implemented motion estimation algorithm is described in Section III. Section V and IV exhibit different parallelization methods on data and task level, respectively. Results for the separate parallelization methods and their interference are given in Section VI. Finally, Section VII briefly concludes this paper.
II. RELATED WORK

Motion estimation is the task of assigning a two-dimensional vector to objects, image patches, or pixels, describing the translational movement between two successive frames. Motion estimation is applied in many application fields, ranging from motion compensated temporal interpolation and temporal noise reduction in consumer electronics to ego-motion estimation in advanced driver assistance systems. While optical flow approximates the image intensities with a Taylor polynomial, resulting in a linear equation system with additional constraints, we transfer and adapt a block based algorithm from video-coding to the use in driver assistance. Block based algorithms divide the input image into equally sized image patches (blocks) and generate distinct motion vectors for each. Leading to a two-dimensional matching problem between blocks of two successive frames, such block-based algorithms vary in their search space and the set of tested matching candidates. Traversing each candidate inside a predefined maximum search space, the Full Search Block-Matching (FSBM) features low control flow and many independent calculations, predestinated for concurrent execution. Thus, the search area dimensions define the maximum possible detectable motion vector, the computational expenses increase due to the huge amount of candidates in driver assistance systems with increasing car speed. Predictive algorithms (e.g., proposed by de Haan [3], Blume [1] and Wang [4]), considering temporal and spatial dependencies, generate smoother vector fields with higher relations to the true motion, reducing the error on periodic structures and sensor noise. While decreasing the computational effort by consciously selecting matching-candidates, dependencies in data and control-flow involve more complex implementation challenges. As these algorithms, however, exhibit high computational complexities, previous implementations make use of dedicated or customized hardware. [17], resulting in low flexibility, thereafter. To refine the motion vector field from block to pixel granular coordinates, de Haan [2] proposed a vector field erosion filter.

In the past, programmers profited from higher performance through increasing clock rates without code modifications. Due to the exponential growth of power dissipation with raising clock frequencies, processors shift from single to multi-core architectures. As the so-called Free Lunch, described by Sutter [6], is over, the migration of sequential code to concurrent architectures is mandatory to harvest the added processing performance of multiple computation engines. As programs nowadays have to be written in an explicit parallel manner to exploit all processor resources, evaluations of programming instruments in terms of simplicity and flexibility and the resulting performance benefits are important. Asanovic et. al. [7] proposed different methods, beside parallel programming models and auto-tuners, increasing the algorithm performance, in case the software design is adapted to specific computation and communication patterns. Lee [11] argues, that threads represent a step in automatic migration from sequential code execution and therefore proposed essentially deterministic library components. Encapsulating domain-specific knowledge in reusable components, integrating concurrency with languages, and supporting explicit declarations to help compilers and operating system schedulers, can help effectively exploiting multi-core architectures through software in the opinion of Manferdelli et. al. [8]. Höfler et. al. [9] proposed to use performance modeling methods from the development to the production stage, enriching the application design process. A market survey by Hebsch et. al. [10] accentuates highly used programming tools in the tool driven multi-core software development. Beside sequential code optimizations, Arndt et. al. [5] presented an efficient software migration workflow from single- to multi- and many-core architectures.

Present and future architectures feature different SIMD hardware units with increasing register width, ideally suited for parallel processing on data level. Related programming instruments range from the automatic vectorization on loop level by the compiler to the use of hardware specific assembler instructions or intrinsics. Language extensions, such as pragmas and array slicing notations, introduced to theystem language with Intel Cilk Plus, allow the programmer to provide the compiler informations (e.g., data dependencies, minimum loop lengths, branching probabilities, for optimized code generation).

III. ALGORITHM BASICS

Motion estimation algorithms assign an individual vector \( \vec{V}(x, k-1) \) to each pixel \( I(x, k-1) \) of the reference image, describing the local motion between two successive frames \( k-1 \) and \( k \). Each vector consists of the pixel-accurate displacement, describing the position of the candidate block in the following frame with the maximum similarity. The basic FSBM traverses the reference block within a predefined search area in the following frame and selects the candidate vector with the maximum similarity, on the basis of the block-wise sum of absolute differences (SAD). Large block displacements, common in driver assistance systems, require a speed proportional adaption of the search area, increasing the computation time, while becoming more noise sensitive.

Assuming spatial and temporal relations within the image, the number of candidate vectors, influencing the necessary execution time, can be significantly reduced, while the vector field homogeneity is enhanced. In respect to predominant motion in horizontal and vertical directions, the Parallel Predictive Motion Estimation (PPBM) [11] algorithm divides the search space into the same independent convergence directions (horizontal and vertical), each converging to the true object movement via the amount of frames and number of blocks within an object. Each convergence direction includes one spatial, one temporal and eight spatial update candidate vectors per block, as illustrated in Figure 1. Based on the hypothesis, that an object is larger than a block, spatial predictors represent an already calculated motion vector of the current vector field. Consequently, the motion vector is taken from a block left or above of the current block for the horizontal or vertical convergence direction respectively. The temporal candidate takes the motion from the previous frame.

Fig. 1. Common predictor configuration of the PPBM, and possible horizontal candidate vectors, extracted from the spatial and temporal predictor.
into account, wherefore its position is independent. The fixed update vector set, added to the spatial predictor, ensure the convergence of the predicted motion vectors against the true object movement and correct small motion changes. Within each convergence direction, the candidate with the maximum similarity, evaluated by the sum of absolute differences (SAD), is chosen. Finally, the independent convergence directions are combined using a decision scheme, based on the possible convergence against the true motion within each convergence direction and motion vector length. Additionally, the zero vector is evaluated, resulting in 21 candidate block positions for each block. A typical predictor configuration for the current block at the position $\vec{c}$ is illustrated in Figure 1 with an block size of $[4 \times 4]$. The spatial and temporal predictors, $\vec{v}_{h/v}$ and $\vec{V}_{h/v}$, are extracted from the current or previous vector field $\vec{V}_{h/v}(\vec{x} + \vec{t}, k - 1)$ at the position $\vec{V}_{h/v}(\vec{x} + \vec{t}, k - 2)$ respectively for the separate horizontal and vertical search directions. For overview purposes only the candidate vectors of the horizontal search direction are illustrated.

The implemented motion estimation algorithm uses a cascade of Parallel Predictive Block-Matching (PPBM) stages with decreasing block sizes from stage to stage, illustrated in Figure 2. An initial block-matching $PPBM_1$ with bigger blocks omits errors through periodic structures, in combination with a reduced calculation time. The following smaller block sizes, taking the results of previous stages as temporal predictors into account, improve the coarsely granular vector field to the pixel plane through iterative dividing each block into four equally sized sub-blocks and assigning each sub-block the median vector of its neighborhood.

**IV. PARALLELIZATION**

In this section, we introduce multi-core parallelization methods on task level, which we applied to the previously introduced motion estimation pipeline, consisting of the Parallel Predictive Block-Matching and a subsequent erosion. While considering internal control- and dataflow dependencies of the algorithm, we present the associated software structure and work distribution strategy. Based on the heterogeneous ARM big.LITTLE [19] multi-core architecture, the Samsung Exynos 5 Octa 5422 SoC [18] represents the target platform. This multi-core processor is divided into two quad-core clusters, which communicate via a coherent interconnection bus, managing shared memory accesses. The big and LITTLE clusters feature each four ARM Cortex A15 or A7 respectively, running at 2.0 GHz or 1.4 GHz. As each big core features a 128 bit ARM NEONv2 and VFPv4 SIMD unit, we compare the performance of these accelerators with the vector units of the Intel Core-i7 4771 quad-core general-purpose processor. This architecture provides a 256 bit AVX2 vector unit on each core, backward compatible to the MMX and SSE instruction-set extensions. Due to dual hyperthreading, the processor entirely delivers eight threads with hardware-managed context switching in the single quad-core cluster, running at 3.5 GHz.

Minimizing synchronizations, work imbalances and communication latencies, when distributing the entire work load over multiple cores, requires a detailed knowledge of the algorithm’s data dependencies, which are caused by spatial predictors and the algorithm’s processing chain. Since this algorithm contains two independent and thus potentially concurrent cascades of PPBM stages, representing the horizontal and vertical convergence direction, both cascades could be processed in parallel. However, as such parallelization schemes limit the dynamic scalability of the resulting implementation to a maximum of two cores and moreover involve high memory consumptions, we decided to execute the individual processing stages sequentially. Here, task parallelism is enabled by splitting the data sequences in equally sized chunks (i.e., data decomposition [5]) within each individual processing stage, as illustrated in Figure 3. Thereby, a separate and thus private data chunk is assigned to each core, improving data locality and minimizing cache coherency overhead. Each core concurrently executes the same processing stage on an equally sized data chunk, which nearly eliminates all possible work imbalances and idle times. As horizontal spatial predictors involve horizontal data dependencies, while vertical spatial predictors involve vertical data dependencies, the data decomposition splits the work amount with respect to the individual dependency direction into horizontal or vertical data chunks respectively. Moreover, as each core processes its data chunk in column-row order, the data locality is further improved. To exploit the entire vector register capacity and further increase memory access linearity, the combination process has been split into two stages. First, the SAD of the zero vector candidate is calculated for each block in the current image and stored in a temporal array. Afterwards this temporal array is combined with candidates of both dependence direction, resulting in the final motion vector field.

**V. VECTORIZATION**

The SAD block similarity calculation, which represents the runtime bottleneck is perfectly suited for data level parallelism.
Fig. 5. Code samples for the SAD calculation of a block with size [16 \times 16] (i.e., vectorization), as it exhibits neither data dependencies nor a high control flow complexity, but features linear memory accesses within each block row. The programmable SIMD accelerators on both considered architectures feature optimized memory access operations and specialized SAD instructions, reducing the number of memory accesses and instructions in comparison to the sequential execution. Moreover, outsourcing the workload enables hardware-managed out-of-order executions, thus the calling processor is able to execute scalar operations concurrently to vector operations. In this section we present different vectorization methods and demonstrate exemplary implementations of the SAD calculation of a single block, in order to evaluate the coding effort and difficulty. The non-vectorized SAD calculation of a single block can be expressed as a double nested loop, iterating over all block elements and accumulating the pixel-wise absolute difference, as illustrated in Figure 5a.

**Auto-vectorizer:** Compilers often include methods for auto-vectorization, representing the most common method of vectorization [14]. Modern auto-vectorizers are able to vectorize on loop-level with a compiler-specific amount of known pattern, while this set of patterns mainly influences the effectiveness and efficiency of vector operations. As auto-vectorizers can be used for regular high-level languages, this instrument provides the highest portability without the need of any code changes. In order to improve the efficiency of the vectorization, the loop body should be as slim as possible to fit the compiler’s known loop-pattern most optimal and the loop length should be known to the compiler at code generation time. Additional performance improvements and optimized load and store instructions can be enabled through compiler hints, which accentuate aligned memory or explicitly exclude loop carried dependencies, caused by multiple pointers pointing to the same memory block. Therefore, OpenMP [16] offers comfortable SIMD pragmas. Considering the SAD calculation of one block, compiler hints concerning properly aligned and linear memory access with a summating reduction can improve the auto-vectorization result and are illustrated in Figure 5c.

**Array Slicings:** An alternative abstract vectorization instrument is the notation of array operations in slicings, known from MATLAB. Intel Cilk Plus [15] introduces this method in the current Intel compilers and the GCC starting from version 4.9. Due to the explicitly parallel syntax expression, as illustrated in code 5b, efficient vectorizations on multiple architectures become possible. However, Cilk Plus array slicing operations are currently not generally supported on ARM.

**Hardware specific intrinsics and inline assembly instructions** offer low-level access to SIMD engines [13] on the costs of portability and flexibility losses [12], as well as increasing programming effort. Hardware specific intrinsics, available in modern compilers, allow the compiler to further optimize the software by hiding explicit register allocation from the user in comparison to inlining blocks of assembly-language. As this intrinsics are directly mapped to the equivalent assembler instructions, we will focus on this vectorization method. As such vector instruction-set extensions become more and more standardized and are typically valid for whole architecture families, the portability increases. For instance Intel released a wrapper for the execution of NEON specific intrinsics on AVX/SSE SIMD units.

Using data level parallelization methods, the inner loop of the SAD calculation of one block can be partially executed in parallel. Within each iteration of the loop over all block rows, first one entire line of the reference and candidate block are loaded into two separate vector registers and their row-wise SAD value is accumulated over all rows. The AVX2 and SSE instruction-set extension of the Intel architecture feature a single instruction, computing the element-wise absolute difference of two vector registers and internally reducing each vector of eight consecutive absolute difference values to one 16 bit element. Finally, the accumulated 16 bit SAD value, which is generated for each row of the current block, needs to be extracted from the vector register into a scalar register. In case of a block width including more than eight elements, the two SAD of the upper and lower register half need to be added. The resulting code is illustrated in Figure 5d and reveals the huge modifications during the migration workflow, in order to exploit the SIMD units. As the maximum considered block width of the PPBM is 16 pixel (each 8 bit grayscale) and thereof derived maximum register length is 128 bit, we will focus on the SSE instruction-set extension in the remainder of this paper. Implementations, considering ARM NEON compatible vector units, include SAD calculations of two vectors is realized using a vertical accumulation of the element-wise absolute difference of two registers in one accumulating register in one single instruction. Finally, the accumulating register, including the column-wise absolute difference over all rows of each column,
needs to be reduced to one block-wise SAD. This is realized by a staged summing reduction, horizontal pair-wise adding the values of the vector register in each iteration. In order to prohibit range overflows of the vertical accumulation, the element-wise absolute difference is widened to 16 bit elements prior to the accumulation. Thus, the NEON vectorization demands twice as much registers, compared with SSE units, for block widths above eight pixels (128 bit accumulation registers). Considering a particular convergence direction, each reference block is compared with ten candidates. In order to reduce load operations, in each iteration of the loop over the lines of the compared blocks, the reference line is loaded once and compared with each corresponding line of the candidate blocks. The definition of the update-vector set at runtime prohibits the further combination of multiple SAD calculations within one vector register, for instance register shifting or simultaneous calculation in the lower and upper half of the register.

As, no vector-valued median filter is defined, the motion vector field erosion from block to pixel coordinates divides into an element-wise median filtering and a following verification. This postprocessing step ensures, that both components result from one input vector, prohibiting new motion vectors without a previous block-matching. The SIMD units enable the calculation of the erosion for all four sub blocks in one vector register. The unstructured register assignment of the neighboring vectors prevents optimized load operations and result in a scalar motion vector loading, followed by an unoptimized stuffing of the SIMD registers.

VI. Evaluation

In this section, we present our performance evaluation results. First, we compare the different presented vectorization methods in terms of performance and resource exploitation. Additionally, we give a performance comparison between the SSE and NEON SIMD units on the different examined platforms. Finally, we evaluate the scaling behavior of the vectorized motion estimation algorithm on multiple cores.

A. Vectorization

The duration of the vectorized SAD calculation for one block in clock-cycles is illustrated in Figure 6 for particular block sizes on the Samsung Exynos 5 Octa embedded processor and Intel Core-i7 general-purpose architecture. The proportional execution time increases linearly with the block size for the non-vectorized execution on both considered architectures comparable. The auto-vectorizer of the GCC Compiler (-O3) generates a comparable scaling behavior over the cores on both architectures, but delivers a speedup of 3.18 at a block size of $[16 \times 16]$ on the Intel GPP architecture, in comparison to 7.21 on the ARM big.LITTLE architecture. The performance of the vectorization using additional OpenMP compiler hints, nearly matches the performance of the auto-vectorizer on the general-purpose processor for block sizes up to $[8 \times 8]$ ($S = 1.56$). Further block size enlargements enable a further acceleration in comparison to the optimized auto-vectorizer of factor 2. Considering the ARM mobile multiprocessor architecture, the number of clock cycles of the SAD calculation for one block using OpenMP compiler pragmas, follow the auto-vectorized implementation, but slowing down the calculation for blocks of size $[2 \times 2]$. The usage of array slicing notations on the GPP processor, introduced by Intel Cilk Plus, results in a speedup between the highly optimized usage of SSE hardware intrinsics and the auto-vectorizer, achieving an acceleration of 4.75 for the SAD calculation of blocks with size $[16 \times 16]$, which is factor 3.45 better than the auto-vectorizer.

The SAD calculation implementation, using hardware specific and highly optimized SSE and NEON intrinsics on the Intel general-purpose and mobile multi-core architecture respectively, outperforms all auto-vectorized implementations for all considered block sizes. The number of clock-cycles is nearly constant for block sizes up to $[8 \times 8]$ utilizing the SSE instruction-set extension intrinsics, due to unoptimized memory accesses for underused vectors. Data of blocks with row lengths smaller than 8, needs to be loaded in the scalar units and then stuffed into vector registers, resulting in additional overhead, undoing the advantage of a smaller row count. NEON instruction-set extensions are able to handle small amounts of data more efficiently compared to SSE. Block of size $[16 \times 16]$ need double the number of registers and instructions per block line, due to the element-wise accumulation with 16 bit words, resulting in the jumping number of clock cycles. Due to a slower execution of the non-vectorized SAD calculation on the ARM architecture, in comparison to the GPP, the resulting speedup for blocks of size $[16 \times 16]$ of 91.1 (NEON) outperforms SSE intrinsics, achieving a speedup of 85.1 compared to the sequential execution.

While the auto-vectorization requires no code changes, but achieves low benefit, hardware specific intrinsics can exploit the available vector resources most effectively. In contrast to the high coding effort for intrinsics, slicing notations come with increased portability at adequate speedup.

B. Parallelization

The execution time of the vectorized implementation of the motion estimation pipeline, using NEON or SSE instruction-set extension intrinsics on ARM or Intel architecture respec-

![Fig. 6. Evaluation of clock cycles for the SAD calculation of one block with variable block size on the general-purpose and mobile ARM architecture.](image-url)

(a) SAD cycles on Intel Core-i7 (3.5 GHz).

(b) SAD cycles on Samsung Exynos 5 Octa (2.0 GHz).
Table 1 summarizes the execution times and resulting speedups of the realized motion estimation pipeline with a resolution of 720p on both considered architectures. The total speedup $S_{tot}$ characterizes the ratio between the sequential, un-vectorized reference and the presented implementation utilizing data- and task-level parallelism. Using data parallelization methods for the combination and erosion stage, the memory bandwidth limits the resulting speedup. Additionally, the vectorization speedup of the erosion is lower, due to inefficient loading operations, combined with a lower number of vector operations, which influence the vectorization speedup.

VII. CONCLUSION

With the increasing computational performance of low power embedded multi-core architectures and emerging general-purpose processors, the implementation of complex real-time algorithms in software is feasible. In applications, demanding flexibility, portability and low development costs, a parallel software implementation competes against dedicated hardware architectures. In this work, we presented a parallelized and vectorized implementation of a common motion estimation algorithm on a Samsung Exynos 5 Octa, representing a heterogeneous mobile multi-core architecture. While we investigated several vectorization and parallelization strategies, the most effective implementation features 38.5 fps at VGA resolution on the embedded ARM platform. Maximum resource utilization and computational performance of the SoC is achievable by software tuning and manual vectorization, such as using instruction-set specific intrinsics, at the expense of lower architectural flexibility.

REFERENCES