

# Acceleration of Optical Flow Computations on Tightly-Coupled Processor Arrays

Éricles Rodrigues Sousa<sup>1</sup>, Alexandru Tanase<sup>1</sup>, Vahid Lari<sup>1</sup>, Frank Hannig<sup>1</sup>, Jürgen Teich<sup>1</sup>, Johny Paul<sup>2</sup>, Walter Stechele<sup>2</sup>, Manfred Kröhnert<sup>3</sup>, and Tamin Asfour<sup>3</sup>

<sup>1</sup>Hardware/Software Co-Design, University of Erlangen-Nuremberg

<sup>2</sup>Institute for Integrated Systems, Technical University of Munich

<sup>3</sup>Institute for Anthropomatics, Karlsruhe Institute of Technology

<sup>1</sup>{ericles.sousa,alexandru-petru.tanase,vahid.lari,hannig,teich}@cs.fau.de

<sup>1</sup>{johny.paul,walter.stechele}@tum.de

<sup>3</sup>{manfred.kroehnert,tamim.asfour}@kit.edu

**Abstract:** Optical flow is widely used in many applications of portable mobile devices and automotive embedded systems for the determination of motion of objects in a visual scene. Also in robotics, it is used for motion detection, object segmentation, time-to-contact information, focus of expansion calculations, robot navigation, and automatic parking for vehicles. Similar to many other image processing algorithms, optical flow processes pixel operations repeatedly over whole image frames. Thus, it provides a high degree of fine-grained parallelism which can be efficiently exploited on massively parallel processor arrays. In this context, we propose to accelerate the computation of complex motion estimation vectors on programmable tightly-coupled processor arrays, which offer a high flexibility enabled by coarse-grained reconfiguration capabilities. Novel is also that the degree of parallelism may be adapted to the number of processors that are available to the application. Finally, we present an implementation that is 18 times faster when compared to (a) an FPGA-based soft processor implementation, and (b) may be adapted regarding different QoS requirements, hence, being more flexible than a dedicated hardware implementation.

## 1 Introduction

Computer vision algorithms such as the optical flow [HS81] can give important information about the spatial arrangement of objects in a scene and their movement. Due to the ever growing demand for autonomous systems, the importance of computing the optical flow in different embedded application domains (e. g., robotics, mobile phones, automobiles) is also increasing. For example, in [TBJ06], the importance of the optical flow for maneuvering a robot was shown. Mobile phones were transformed into pointing devices (using their video camera) with help of the optical flow in [BR05]. In [SBKJ07], Jae Kyu Suhr and others determine an automatic parking system that is able to find an appropriate free parking space based on an optical flow algorithm. Moreover, continuous technology scaling has enabled embedded computer systems to integrate multiple processors and hardware accelerators on a single chip, also known as MPSoC. It can be foreseen that in the year 2020 and beyond, the number of such computing units will exceed 1 000 processing elements on a single chip. Now, programming such a large number of processing elements poses several challenges because centralized approaches are not expected to scale [MJU<sup>09</sup>]. Such an excessive increase in the amount of hardware resources on a

chip raises questions regarding the power consumption, specially in case of mobile devices where the power budget is highly limited. Hence, there exists a trend toward specializing the functionality in MPSoCs by employing *application-specific* or *programmable hardware accelerators*. Programming different parts of such heterogeneous MPSoCs, even more exacerbates the situation. Old school parallelization approaches do not match the dynamic behavior of applications and the variable status and availability of resources in large chip multiprocessors. These drawbacks can be seen in some of the architectures available today, i.e., Tilera’s 64-core processor [WGH<sup>07</sup>], 48-core Intel SCC processor [MRL<sup>10</sup>], Xetal-II [AKC<sup>08</sup>] (a SIMD processor with 320 processing elements), and Ambric’s discontinued massively parallel processors with 336 RISC cores [NWC<sup>09</sup>]. B. Hutchings et al. [HNW<sup>09</sup>] described an optical flow streaming implementation on Ambric AM204. However, due to the limitations in the I/O interfaces, the processor elements (PE) are stalled for more than 70% of the time, indicating under utilization. X. Lin et al. [LHY<sup>11</sup>] demonstrated the parallelization of a motion JPEG decoder on Tilera’s processors by using threads with shared memory communication, which scales only partly. In [DPFT11], the authors investigated the scalability of the Intel SCC [MRL<sup>10</sup>]. They demonstrated that applications which benefit from the massively parallel architecture must have a high ratio of computations versus data volume. Image processing applications usually are on the other end of the spectrum, where only a few operations are performed on each pixel and the same set of operations is needed to be repeated on millions of pixels, generating a high data volume. Hence, image processing applications, when implemented in a conventional manner (using multi-threading), cannot efficiently utilize the high parallelism that is available in massively parallel processor arrays. [MJCP08] shows that visual computing applications, like face detection and H.264 motion estimation also do not benefit from multi-threading, as it leads to load imbalances. This work also demonstrates that applications might fail to attain performance benefits on processor arrays if the available memory bandwidth is not utilized efficiently. The applications running on the PEs exchange data through message passing or shared memory. But, due to the difficulties in synchronizing data accesses through a shared memory, the most efficient way would be message passing. Also, it is important to mention that massively parallel processor arrays typically do not support cache coherency and the programmer has to handle it through various locking mechanisms. However, if used extensively, locks may create a significant overhead and the PEs will be stalled by waiting for locks for a significant part of actual execution time. As a remedy, in this paper, we present an alternative implementation to accelerate computationally intensive image processing algorithms, i.e., the optical flow algorithm, on massively parallel processor arrays such as *tightly-coupled processor arrays* (TCPAs) [THH<sup>11</sup>].

Frequently, TCPAs are used as an accelerator in an MPSoC to speedup digital media and signal processing applications. They consist of an  $N \times M$  mesh array of programmable PEs, each having a VLIW structure. A simplified design of a TCPA with 24 processor elements (PEs) is sketched in Figure 1. Only a limited amount of instruction memory and a small register file is available in each PE, and the control overhead is kept as small as possible for a given application domain. For example, there is no support for handling interrupts or multi-threading. Also, the PEs have no direct access to the main memory but data is streaming from the surrounding buffers through the array and is processed from PE to PE. These arrays, however, may achieve a very high energy efficiency [KGS<sup>11</sup>, LMB<sup>12</sup>]

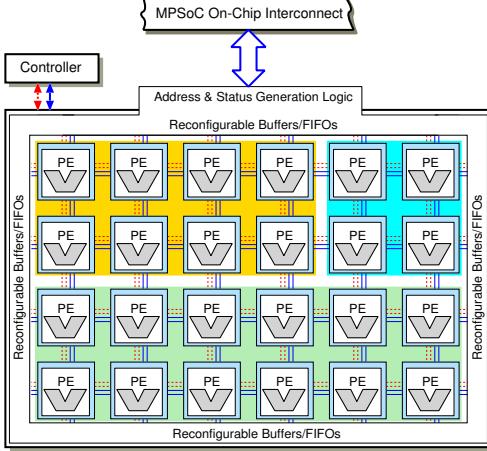


Figure 1: Tightly-coupled processor array that may be used as an accelerator in an MPSoC. The highlighted rectangular areas are occupied by three applications that may run simultaneously.

and thus are well suited for streaming applications in embedded portable devices. TCPAs rely on fine-grained scheduling of loop iterations and do not require each tile to be executed atomically, thus, lifting the corresponding constraint that is imposed by other available massively parallel architectures (e.g., GPUs, multicore architectures), and giving more room for optimized mapping and code generation. This fact is the major difference when compared to the execution of multi-threaded applications in multicore architectures or of microthreads in graphics processors, where threads are executing atomically the dispatched tiles. Moreover, no thread initialization and synchronization overhead is introduced, which directly leads to smaller overall execution latencies. Instead, tightly-coupled processor arrays may exploit the direct PE to PE communication, where the data is streaming from the surrounding buffers through the array, thus, lifting the above mentioned drawbacks while benefiting from both instruction as well as loop-level parallelism and enabling zero-overhead program control flow results in ultra-fast execution of applications [KSHT09]. To achieve both the necessary processing speed as well as low power consumption, algorithms or parts of them need to be accelerated on dedicated or programmable massively parallel architectures. This paper considers TCPA architectures for accelerating image processing applications with 2D sliding windows such as optical flow and performs an evaluation by comparing the results with other state-of-the-art implementations. Novel to our approach is that the application may be implemented such that the execution speed may adapt to a varying number of available and employed processing elements, respectively. On the other hand, in order to obtain a certain processing speed, a certain number of processing elements must be available. This flexibility as well as self-organizing properties are exploited, stemming from a new resource-aware computing paradigm called *Invasive Computing* [THH<sup>+</sup>11].

Section 2 presents our approach, i.e., how to map the different parts of optical flow computation on TCPAs. Results are discussed in Section 3, and finally, Section 4 concludes.

## 2 Optical Flow Computation on TCPA

A TCPA as shown Figure 1 consists of a two-dimensional set of processing elements (PE) that may be customized at synthesis-time with respect to many parameters such as types and number of functional units for arithmetic, logic, branching and logical operations. The interconnection network consists of point-to-point connections between neighbor PEs that are also parameterizable using interconnection wrappers, each of them surrounds a PE. Using this wrapper, different interconnection topologies can be configured at run-time by changing the value of select registers in the interconnection wrapper multiplexers. As the processing elements are tightly-coupled, they do not need direct access to the global memory. The data transfers to the array is performed through the PEs at the borders, which are connected to the banks of surrounding memory buffers. These buffers could be implemented as simple FIFO buffers or RAM-based addressable memory banks. The memory architecture and technique used to fill the pixel data into the I/O buffers in Figure 1 is beyond the scope of this paper and it is explained in detail in [LHT09].

Due to the aforementioned arguments, massively parallel processor arrays are well suited for accelerating compute-intensive vision algorithms such as optical flow. In the following, three subsequent compute stages of the optical flow algorithm are outlined. In the first stage, an *image smoothing* is performed as described in Section 2.1. The second stage is the *census signature generation* (see Section 2.2) where a unique signature is computed for each pixel in a frame based on its neighboring pixels. The third stage operates as follows: When two consecutive signature images are available, the signature of one pixel in frame  $t$  is compared with the signatures of all pixels in a certain neighborhood of frame  $t + 1$ , i. e., a possible *matching* is determined, which computes a corresponding flow vector (see Section 2.3). Since such a pixel-to-pixel signature comparison is a computationally intensive task, the search region is limited to a small region in frame  $t + 1$  located around a candidate pixel from frame  $t$ . The wider the search region, the longer the flow vector might be, and the faster the motion that can be detected. A wider search region also helps to reduce the number of false vectors. But, this also leads to higher computing requirements as more signatures need to be compared.

Based on the above application requirements, we outline next how each stage may be mapped to a  $w \times d$  region of a TCPA array, where  $w$  is the window size and  $d$  is called *depth* and denotes the number of consecutive pixel values and moving window operations, respectively. We have the possibility to allocate more PEs and increase the value of  $d$  in order to decrease the execution time. In case of insufficient resources, the application can decide whether to reduce the window size  $w$  so to avoid a frame drop. The upcoming sections explain this scalable application mapping for all three stages of the optical flow and demonstrate the variation in computation time with varying resources on a TCPA.

### 2.1 Image Filtering

The optical flow computation starts with a simple image filtering in order to remove noises in the input image. In the algorithm a 2D window of size  $w \times w$  slides over the complete image. The window is placed centered at each pixel in the image and each pixel is modified based on its neighboring pixels and the weights as in Eq. (1), where  $P(x, y)$  is the pixel intensity at location  $(x, y)$  in a frame of size  $U \times V$ ,  $W(x_1, y_1)$  is the corresponding coefficient weight and  $w$  is the window size.

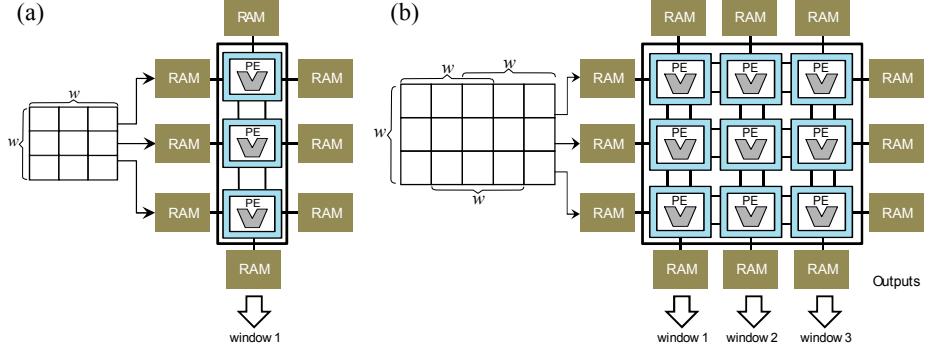


Figure 2: Basic image filtering in a TCPA for a  $w \times w$  window, where  $w = 3$  in (a). Processing multiple 3 windows simultaneously in (b)

$$P_s(x, y) = \frac{\sum_{x_1=-\lfloor w/2 \rfloor}^{\lfloor w/2 \rfloor} \sum_{y_1=-\lfloor w/2 \rfloor}^{\lfloor w/2 \rfloor} P(x+x_1, y+y_1) \cdot W(x_1, y_1)}{\sum_{x_1=-\lfloor w/2 \rfloor}^{\lfloor w/2 \rfloor} \sum_{y_1=-\lfloor w/2 \rfloor}^{\lfloor w/2 \rfloor} W(x_1, y_1)} \quad (1)$$

forall input pixels  $P(x, y) : \lfloor w/2 \rfloor \leq x < U - \lfloor w/2 \rfloor \wedge \lfloor w/2 \rfloor \leq y < V - \lfloor w/2 \rfloor$ . A careful inspection shows that for a sliding window approach each window shares its pixels with the neighboring window. Thus, if multiple neighboring windows/pixels are processed simultaneously, this may lead to a) much better utilization of memory bandwidth. To process one pixel for a window size of  $3 \times 3$ , we map the above computation onto a set of 3 PEs as shown in Figure 2(a). Each PE processes one row of the window, so together they can process one complete window in  $w$  time steps. Moreover, to exploit the parallelism of TCPAs, multiple windows (corresponding to multiple pixels) can be processed simultaneously. For example, another set of 3 PEs, if available, can process the neighboring window by reusing the data read by the first set of PEs, through continuous data exchange over the dedicated interconnect as shown in Figure 2(b). This helps to reduce the overall execution time without increasing the memory bandwidth requirements. For example, for an image filtering application processing a  $5 \times 5$  window, the application can be optimally mapped to a  $5 \times 5$  PE region. If a sufficient number of PEs is not available, the application can decide to choose the biggest available configuration, e. g., in case only an array of size  $3 \times 2$  is available, the application can decide to filter only using a  $3 \times 3$  window instead and process only 2 windows simultaneously. This gives the application an opportunity to scale its requirements according to the amount of available resources. Eq. (2) can be used to compute the execution time in cycles/pixel (CPP) for any array configuration of size  $w \times d$ :

$$CPP(w, d) = \frac{(C_p \cdot w) + (d - 1)C_d + C_{const}}{d} \quad (2)$$

In Eq. (2),  $C_p$  is the number of clock cycles to process the pixels of one row,  $d$  denotes the number of windows (pixels) that are processed simultaneously.  $C_d$  denotes the number of cycles needed for one PE to skip the pixel which does not belong to its window, because each PE of one row receives the same input values, and  $C_{const}$  is the time to reinitialize  $P_s(x, y)$  after processing one window and move on to the next. Note that the presented application mapping can be applied to many other sliding window image filters like edge detection, Harris Corner detection, morphological operations (e. g., dilation, erosion).

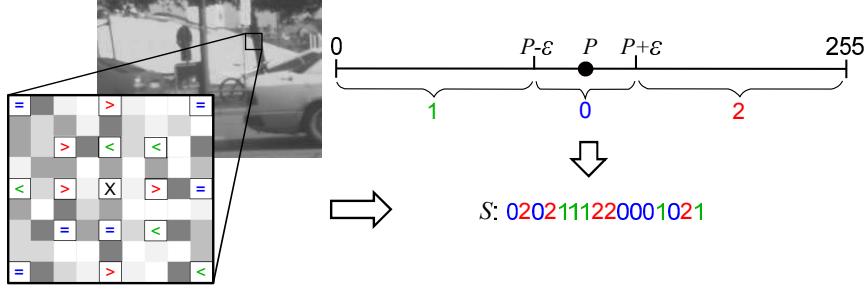


Figure 3: Signature generation technique

## 2.2 Census Signature Generation

After image filtering, the next step is to generate census signatures from the filtered image. The technique described in [CLJS09] is briefly explained in Figure 3. A unique signature  $\xi(i, j)$  is calculated for each pixel in the frame by comparing its value (marked by 'x') with its neighboring pixels located within its window region. The result of each comparison can be presented by two bits and is calculated using a deviation  $\epsilon$  as shown in Figure 3. Unlike window-based image filtering, only a subset of the neighboring pixels is required for signature generation. Those pixels are marked by the symbols ( $<$ ,  $=$ ,  $>$ ) that indicate the relation of the pixel  $(x, y)$  with its neighboring at a particular location  $(x + i, y + j)$  as visually described by the 16-stencil in Figure 3. As described by Eq. (3), when a neighbor pixel is bigger than  $P(x, y) + \epsilon$ , 2 is output. However, if the neighbor is less than  $P(x, y) - \epsilon$ , an 1 as output is assumed. Otherwise, when the neighbor pixel is between both values, a 0 is output. Next, the 16 computations of  $\xi$  are concatenated to one signature  $S(x, y)$  (see Eq. (4)).

$$\xi_k = \begin{cases} 2, & \text{if } P(x + i_k, y + j_k) > P(x, y) + \epsilon \\ 1, & \text{if } P(x + i_k, y + j_k) < P(x, y) - \epsilon \\ 0, & \text{else} \end{cases} \quad \forall 1 \leq k \leq 16 \quad (3)$$

$$S(x, y) = (\xi_1, \xi_2, \dots, \xi_{16}) \quad (4)$$

From Figure 3, it becomes clear that in total five columns have to process pixels. Hence, in the TCPA implementation, we allocate five PEs per column for computing the signature. Moreover, the signature generation stage of the algorithm also involves a sliding window and hence, the processing can be accelerated using multiple columns of PEs as shown in Figure 4(a). Similar to the approach adopted for image filtering, the results are propagated through the PEs to the buffers located at the borders of the array. A unique signature is thus generated for each smoothed pixel in the input frame, forming a signature image and this is used for computing the flow vectors as explained in the next section. The computation time in cycles/pixel can be calculated using Eq. (5), where  $C_s$  is number of cycles required to move a result from one PE to its neighbor over the dedicated interconnect and  $d$ , as mentioned previously, is the number of windows that are processed simultaneously.

$$CPP(w, d) = \frac{(C_p \cdot w) + (w - 1)C_s + C_{const}}{d} \quad (5)$$

## 2.3 Flow Vector Generation

The final stage of the optical flow computation generates flow vectors representing the relative motion between the camera and the objects in a frame. For vector generation, two

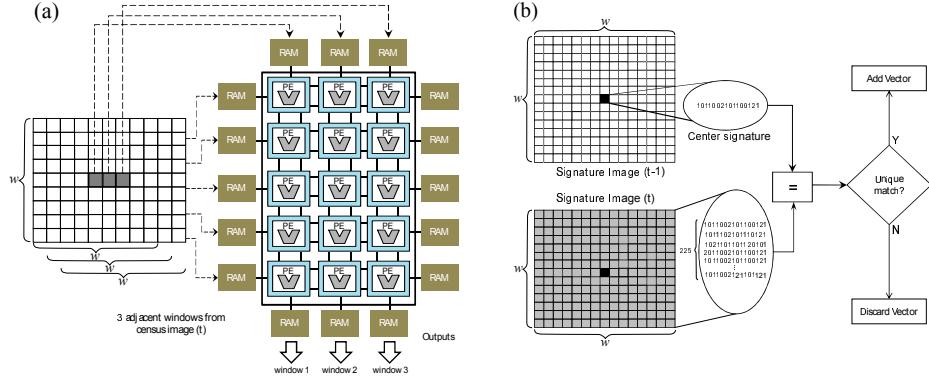


Figure 4: Mapping scheme for signature generation with a considered  $w \times w$  window ( $w = 9$ ) in (a). Vector generation for a  $w \times w$  window ( $w = 15$ ) is shown in (b)

consecutive signature images are required, e. g., from frames  $t - 1$  and  $t$ . As explained before, a signature represents the relation of the current pixel with its neighborhood. Therefore, if the signature of a pixel  $S_{t-1}(x, y)$  matches with the signature of the same or another pixel  $S_t(x', y')$  in the next consecutive frame, then it is assumed that the pixel has moved from its location  $(x, y)$  in frame  $t - 1$  to the new location  $(x', y')$  in frame  $t$ . To understand this process, consider a window of size  $15 \times 15$  centered at coordinates  $(x, y)$  in the signature image, i. e.,  $S_t(x, y)$ . Now, the signature at location  $(x, y)$  in frame  $t - 1$  is compared with each signature within the considered window in the next frame  $t$ . This is explained using Figure 4(b) and the following pseudo algorithm.

If a unique match ( $m(x, y) = 1$ ) was found,  $F(x, y)$  represents the flow vector (i. e., movement of pixel  $P(x, y)$ ). This operation is repeated for each signature in frame  $t - 1$ , using a sliding window.

```

 $m(x, y) = 0$ 
 $F(x, y) = (0, 0)$ 
for  $u = -7 \dots 7$ 
  for  $v = -7 \dots 7$ 
    if  $(S_{t-1}(x, y) == S_t(x+u, y+v))$  then
       $m(x, y) = m(x, y) + 1$ 
       $F(x, y) = (u, v)$ 
    endif
  endfor
endfor

```

As one PE may only process a part of the complete image, the results are not final and have to be compared with results of other PEs for multiple match condition. Each PE performs a signature matching between the central signature and the signatures from the corresponding row within the window and outputs the result in the form of  $(x, y)$  coordinates and match count to the neighboring PE. The individual results are then verified if there was a unique, a multiple or no-match correspondence. This stage uses a mapping scheme similar to the signature generation and hence, the computation time (cycles-per-pixel) can be calculated equally using Eq. (5). In case of insufficient resources, the application may decide

	Image Filter	Signature Generation	Vector Generation
Configuration size (bits)	1 344	3 360	3 232
Configuration latency (ns)	840	2 100	2 020

Table 1: Configuration bit stream sizes (bits) for the three parts of the optical flow as well as the configuration latency  $C_{conf}$ , where a 32-bit bus is used to load each configuration to the PEs (50 MHz clock frequency).

Architecture	Hardware cost			Performance (s)
	slice regs	LUTs	BRAMs	
FPGA	26 114	32 683	81	0.048
5×5 TCPA	35 707	105 799	85	0.427
LEON3	28 355	42 210	65	7.589

Table 2: Hardware cost and performance comparison (time in seconds for fully processing one image frame) for the optical flow application implementation on a LEON3, 5×5 TCPA, and a custom FPGA implementation. All architectures operate at 50 MHz clock frequency and have been prototyped on a Xilinx Virtex-5 FPGA.

again to reduce the window size  $w$  so to avoid any frame drops.

### 3 Results

Assuming the computation according to a VGA image resolution (640×480 pixels), this section presents results as overall performance and scalability of the optical flow application when mapped onto TCPAs of different sizes. Here, the metric cycles per pixel ( $CPP$ ) is considered for comparison representing the number of clock cycles needed to compute one output pixel. Let  $C_{conf}$  denotes the time to configure the TCPA interconnect structure and to load the executable code into the PEs. The values in Table 1 were obtained using a cycle accurate simulator published in [KKHT07] and the graphs in Figure 5 were plotted using Eqs. (2) and (5) for a fixed window size of  $w = 3$  in the case of image filtering,  $w = 5$  for signature generation, and  $w = 15$  for vector generation.

Figure 5 indicates the variation of the  $CPP$  value based on the degree of parallelism ( $d$ ) for the image filtering, and signature and vector generation stages of the optical flow algorithm. According to our explanation in Section 2,  $d$  denotes the number of PE columns that are employed to compute the application (the more PE columns we use, higher the degree of parallelism). The graph clearly shows that the  $CPP$  decreases with increasing  $d$ .

Next, we compare the execution time of our approach with the FPGA-based implementation as well as a pure software implementation on a soft RISC processor (LEON3). Table 2 shows the results of this performance comparison as well as hardware cost comparison. The hardware costs are reported for a Xilinx Virtex5 synthesis and the performance is reported in terms of the overall application computation time per frame. For the TCPA architecture, we configured a 5×5 array in order to show the achievable performance of TCPAs, having a hardware cost close to one LEON3 core. In order to exploit the full processing capacity of the TCPA, each part of the application is assumed to allocate the whole array. This is enabled by fast coarse-grained reconfiguration capabilities of TCPAs, where con-

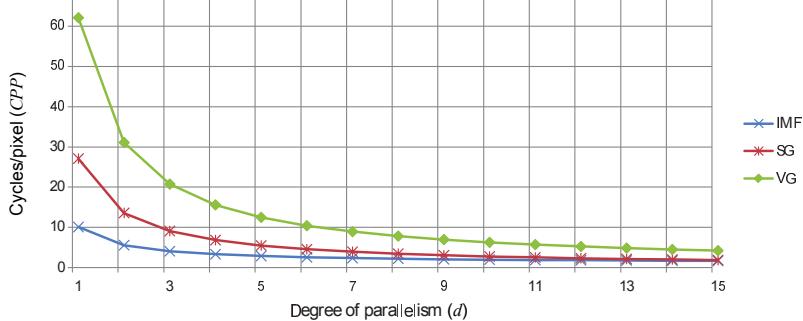


Figure 5: The obtained cycles/pixel (CPP) in dependence of the degree of parallelism  $d$  (number of simultaneous window computations) for a window size of  $w = 3$  in the case of image filtering,  $w = 5$  for signature generation, and  $w = 15$  for vector generation. (IMF: image filtering, SG: signature generation, VG: vector generation)

text switches of the array are performed at the nanosecond scale (see Table 1). Although using more resources than the FPGA-based implementation and consequently achieving lower performance, TCPA architectures have the great advantage to be programmable and able to adapt itself regarding to different QoS requirements. The LEON3 is configured as follows: 4 KB on-chip RAM, 32 KB data cache, and 16 KB instruction cache. Moreover, in our evaluation, we consider only the pure execution time of the application by assuming the input data already being loaded into the data cache of the LEON3 and input buffers of the TCPA, respectively. Therefore, the timing overhead of the communication between peripherals, bus, and the memory access are ignored in each case. In terms of execution time, the LEON3 needs 1244 clock cycles to generate one signature and to compute one matching. Consequently, it takes approximately 7.6 seconds to process one VGA frame, when running at 50 MHz. This implementation would therefore provide no more than 0.13 frames per second (fps). At the same clock frequency, the  $5 \times 5$  TCPA configuration achieves approximately 2.35 fps. However, the size of the processor array could be further increased to increase the performance according to Figure 5. The total area needed to implement the  $5 \times 5$  TCPA, i. e., 25 processing elements is practically equal to the area requirements of a single LEON3. But, the overall execution time achieved by our solution outperforms that of the LEON3 by a factor of 18, and even faster than the Ambric MPPA implementation [HNW<sup>+</sup>09], where in case of Ambric the system operates at 300 MHz frequency and processes  $320 \times 240$  images and obtains 37 fps performance. With the same system setup, the TCPA would reach a frame rate of 55.2 fps.

## 4 Conclusion

This paper presents the benefits of implementing computationally intensive image algorithms, i. e., the optical flow on programmable massively parallel processor arrays such as tightly-coupled processor arrays (TCPAs). We demonstrated that TCPAs do not only provide scalable performance for sliding window based algorithms such as the optical flow, but also benefit from very fast reconfiguration in order to compute the different algorithmic parts in a time-multiplexed manner. The results presented a linear performance improve-

ment in terms of number of available processor elements for different parts of the optical flow application. Moreover, our TCPA-based solution was compared against a software-based implementation on a LEON3 (single core) as well as a known dedicated hardware implementation. Notably, the  $5 \times 5$  TCPA variant does not need substantially more logic resources on an FPGA than an implementation based on a single LEON3 core. However, the TPCA implementation may offer an 18 times higher frame rate than the LEON3 implementation.

## Acknowledgment

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89).

## References

- [AKC<sup>+</sup>08] A.A. Abbo, R.P. Kleihorst, V. Choudhary, et al. Xetal-II: A 107 GOPS, 600 mW Massively Parallel Processor for Video Scene Analysis. *IEEE Journal of Solid-State Circuits*, 43(1):192–201, 2008.
- [BR05] Rafael Ballagas and Michael Rohs. Mobile Phones as Pointing Devices. In *Workshop on Pervasive Mobile Interaction Devices (PERMID 2005)*, pages 27–30, 2005.
- [CLJS09] Christopher Claus, Andreas Laika, Lei Jia, and Walter Stechele. High Performance FPGA-based Optical Flow Calculation Using the Census Transformation. *IEEE Intelligent Vehicle Symposium*, June 2009.
- [DPFT11] A. Diavastos, P. Petrides, G. Falcao, and P. Trancoso. Exploiting Scalability on the Intel SCC Processor. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*, pages 253–256, 2011.
- [HNW<sup>+</sup>09] B. Hutchings, B. Nelson, S. West, et al. Optical Flow on the Ambric Massively Parallel Processor Array. In *17th IEEE Symp. on Field Programmable Custom Computing Machines*. IEEE, 2009.
- [HS81] B. Horn and B. Schunk. Determining Optical Flow. *Artificial Intelligence*, 17(1):185–203, 1981.
- [KGS<sup>+</sup>11] Dmitrij Kissler, Daniel Gran, Zoran A. Salcic, Frank Hannig, and Jürgen Teich. Scalable Many-Domain Power Gating in Coarse-grained Reconfigurable Processor Arrays. *IEEE Embedded Systems Letters*, 3(2):58–61, 2011.
- [KKHT07] Alexey Kupriyanov, Dmitrij Kissler, Frank Hannig, and Jürgen Teich. Efficient Event-driven Simulation of Parallel Processor Architectures. In *Proc. of the 10th Int. Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 71–80, 2007.
- [KSHT09] Dmitrij Kissler, Andreas Strawetz, Frank Hannig, and Jürgen Teich. Power-efficient Reconfiguration Control in Coarse-grained Dynamically Reconfigurable Architectures. *Journal of Low Power Electronics*, 5:96–105, 2009.
- [LHT09] Vahid Lari, Frank Hannig, and Jürgen Teich. System Integration of Tightly-Coupled Reconfigurable Processor Arrays and Evaluation of Buffer Size Effects on Their Performance. In *Proceedings of the 4th International Symposium on Embedded Multicore Systems-on-Chip*, pages 528–534, 2009.
- [LHY<sup>+</sup>11] X.Y. Lin, C.Y. Huang, P.M. Yang, T.W. Lung, S.Y. Tseng, and Y.C. Chung. Parallelization of motion JPEG decoder on TILE64 many-core platform. *Methods and Tools of Parallel Programming Multicomputers*, pages 59–68, 2011.
- [LMB<sup>+</sup>12] Vahid Lari, Shravan Muddasani, Srinivas Boppu, Frank Hannig, Moritz Schmid, and Jürgen Teich. Hierarchical Power Management for Adaptive Tightly-Coupled Processor Arrays. *ACM Trans. on Design Automation of Electronic Systems*, 18(1), 2012.
- [MJCP08] A. Mahesri, D. Johnson, N. Crago, and S.J. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 164–175, 2008.
- [MJU<sup>+</sup>09] Mojtaba Mehrara, Thoma B. Jablin, Dan Upton, David I. August, Kim Hazelwood, and Scott Mahlke. Compilation Strategies and Challenges for Multicore Signal Processing. *IEEE Signal Processing Magazine*, 26(6):55–63, November 2009.

- [MRL<sup>+</sup>10] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, et al. The 48-core SCC processor: the programmer’s view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.
- [NWC<sup>+</sup>09] B. Nelson, S. West, R. Curtis, et al. Comparing fine-grained performance on the Ambric MPPA against an FPGA. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE, 2009.
- [SBKJ07] Jae Kyu Suhr, Kwanghyuk Bae, Jaihie Kim, and Ho Gi Jung. Free Parking Space Detection Using Optical Flow-based Euclidean 3D Reconstruction, 2007.
- [TBJ06] Valerij Tchernykh, Martin Beck, and Klaus Janschek. An Embedded Optical Flow Processor for Visual Navigation using Optical Correlator Technology. In *IEEE/RSJ Int. Conference on Intelligent Robots and Systems*, pages 67–72, October 2006.
- [THH<sup>+</sup>11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, chapter 11, Invasive Computing: An Overview, pages 241–268. Springer, 2011.
- [WGH<sup>+</sup>07] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J.F. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, September 2007.