A Parametrizable Dataflow Implementation of Optical Flow

Reinier van Kampenhout, Sander Stuijk and Kees Goossens
Eindhoven University of Technology, the Netherlands
{j.r.v.kampenhout,s.stuijk,k.g.w.goossens}@tue.nl

Keywords: dataflow, image recognition, parallel computing, real-time systems

An increasing number of products feature complex functionality through the use of embedded computers. By collecting large amounts of data from multiple sensors and interpreting and responding to complex scenarios in realtime, such devices can become "smart." For a control algorithm to make a decision, data from the sensors must be processed before a certain deadline. These constraints may be hard- or soft-real-time, and a combination of SRT or HRT applications may be executed on one platform [2].

In streaming applications such as multimedia and signal processing, the dataflow model of computation (MoC) is a good fit because these applications are inherently data-driven [3]. A dataflow program consists of a graph of actors that communicate tokens through channels. An actor can only fire (execute) if all of its input tokens are available, and if there is enough space on its outgoing channels to produce all tokens. Some dataflow flavours such as mode-controlled dataflow (MCDF) and scenario-aware dataflow (SADF) can be used to deal with dynamic behaviour within one application [4, 5].

In this research we consider the processing of image data to obtain direction vectors of moving objects, on the basis of which control algorithms can make decisions. In particular, we use the optical flow algorithm which is commonly used to track features in a sequence of moving images. To execute this algorithm on a contemporary multi-core streaming platform with high performance and while meeting timing constraints, it is useful have a dataflow implementation. Our implementation is based on the algorithm described in [1]. After refactoring in the *C* language and removal of any dependencies on the external libraries, we ported the algorithm to our CompSOC platform [2]. To distribute the functionality over multiple dataflow actors we recognized three main stages: the loading of frames into local memory, the initial detection of features in a frame, and the optical flow that detects identical features in two consecutive frames.

After splitting the algorithm in these three stages it was parallelized and parametrized. The implementation thus obtained allows to exploit the benefits of the aforementioned mode-control and scenario-aware dataflow MoC, as the parameters can be used to set different modes or use-cases. Parallelization is straightforward as frames are split up in blocks and the calculation on each block is completely independent of other blocks. Parameters that can be adjusted are the frame size, the number of cores that will process a fraction of a frame in parallel, and optical flow parameters, namely: threshold, maximum allowed error and window size. The resulting graph for an implementation with two cores is depicted in Figure 1. Actor a is added to determine or receive the parameters, which are stored in its self-edge. These are forwarded to actor c, f and b.

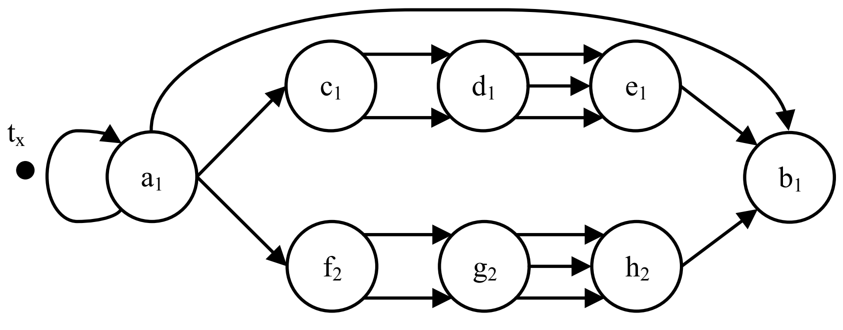

Figure 1: Dataflow graph of the optical flow algorithm, parallelized over two cores. The core onto which an actor is mapped is indicated by the subscript.

C and f both load into local memory the part of the frame that should be processed by their chain, and forward this frame information as well as the parameters to actors d and g. These detect the feature points in the first frame and forward all incoming tokens as well as a list of features to actors e and h. In turn, these actors execute the actual optical flow algorithm. The results are consolidated in actor b, in which basic sanity checks can be performed and the results are either written to memory or passed on to e.g. a control algorithm.

The result of this research is a parallelized dataflow implementation of the optical flow algorithm, which can be parametrized in terms of number of parallel workers, frame size and detection sensitivity. Runtime tests show that if only one core is used, the speed is comparable to that of the non-dataflow sequential version. Adding more cores yields an almost linear speedup, as there are no data dependencies between blocks.

References

[1] M. Birk. Implementierung eines Verfahrens zur Berechnung des optischen Flusses auf einem FPGA. Master's thesis, Universität Karlsruhe, 2009.

[2] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow. *SIGBED Rev.*, 10(3):23–34, Oct. 2013.

[3] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24–35, Jan 1987.

[4] O. Moreira and H. Corporaal. *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor.* Springer, 2014.

[5] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, pages 185–194, July 2006.