Automatic Streamization of Image Processing Applications

Pierre Guillou(✉), Fabien Coelho, and François Irigoin

MINES ParisTech, PSL Research University, Paris, France
{pierre.guillou,fabien.coelho,francois.irigoin}@mines-paristech.fr

Abstract. New many-core architectures such as the Kalray MPPA-256 provide energy-efficiency and high performance for embedded systems. However, to take advantage of these opportunities, careful manual optimizations are required. We investigate the automatic streamization of image processing applications, implemented in C on top of a dedicated API, onto this target accessed through the ΣC dataflow language. We discuss compiler and runtime design choices and their impact on performance. Our compilation techniques are implemented as source-to-source transformations in the PIPS open-source compilation framework. Experiments show lowest energy consumption on the Kalray MPPA target compared to other hardware targets for a range of 8 test applications.

1 Introduction

As predicted by Moore's law, billions of transistors can be integrated today into a single chip, enabling multi-core or even many-core architectures, with hundreds of cores on a chip. The low energy consumption Kalray MPPA-256 processor [17], released in 2013, offers 256 computing VLIW-cores for 10 W. Task and/or data parallel approaches can be used to take advantage of such parallel processing power for a given application domain.

We show how to use this innovative hardware to run image processing applications in embedded systems such as video cameras. In order to enable fast time-to-market developments of new products, applications must be ported quickly and run efficiently on these targets, a daunting task when done manually. To alleviate this issue, we have built a compiler chain to automatically map an image processing application developed on top of a dedicated software interface, FREIA [14], considered as a domain specific language, onto the MPPA processor using the ΣC dataflow language and runtime. Images are streamed line-by-line into a task graph whose vertices are image operators.

Streaming languages [31] have been studied for a long time, and have recently received more attention [28] for exposing pipelining, data parallelism and task paralellism, as well as hiding memory management. The Kahn Process Networks [22] are one of the first streaming model, relying on FIFOs for interprocess communication. As subclasses of this model, Synchronous DataFlow [25,26] languages are statically determined in order to avoid deadlocks and to ensure safety.

J. Brodman and P. Tu (Eds.): LCPC 2014, LNCS 8967, pp. 224–238, 2015.
DOI: 10.1007/978-3-319-17473-0_15

Common SDF languages include LUSTRE [20], Signal [24] and StreamIt [1]. The latter shares some ground concepts with the ΣC dataflow language such as decomposing programs in a graph of basic interconnected units which consume and produce data or focusing on easing programming onto multi-core and many-core targets. In particular, some optimizations of StreamIt applications (operator replication to enable data parallelism, operator fusion to reduce communication overheads [18]) are close to those presented in this paper. Other projects such as DAGuE [9] or FlumeJava [10] propose frameworks for managing and optimizing tasks targetting current multi-core architectures.

Optimization of image processing applications on massively parallel architectures have been the subject of multiple studies. Ragan-Kelley [29] proposes an image processing Domain Specific Language and an associated compiler for optimizing parallelism onto standard CPUs or GPUs. Clienti [12] presents several dataflow architectures for specific image analysis applications. Stencil operators are the most limitating operators in our implementation. Several alternative techniques for optimizing this class of operators have been proposed [6,16].

This paper focusses on the design of a compiler and a runtime using the MPPA chip as an accelerator, including: (1) domain specific transformations on the input code to expose larger image expressions; (2) code generation for a dataflow language; (3) automatic operator aggregation to achieve a better task balance; (4) a small runtime to provide streaming image operators; (5) data-parallel agents for slower operators. We also demonstrate the effectiveness of our approach with a source-to-source implementation in PIPS [15,21] by reporting the time and energy consumption on a sample of image applications.

We first describe the overall compilation chain in Sect. 2, then in Sect. 3 we focus on our hardware and software targets. Section 4 presents our key contributions about the compiler and runtime designs. Section 5 reports our time and energy performance results.

2 Compilation Chain Overview

The starting point of our compilation chain (Fig. 1) is an image processing application, built on top of the FREIA C language API. This provides a 2D image type, I/Os abstractions and dozens of functions to perform basic image operations (arithmetics, logical, morphological, reductions), as well as composed operators which combine several basic ones. Typical applications locate text in an image, smooth visible blocks from JPEG decoding, or detect movements in an infra-red surveillance video. An example of FREIA code is shown in Fig. 2. Our test case applications typically include up to hundreds of basic image operations, with 42 as a median. These operations are grouped in few (1–3, up to 8) independent static image expressions that can be accelerated, stuck together with control code.

The ANR FREIA [7] project developed a source-to-source compilation chain [14] from such inputs to various hardware and software targets: the SPoC image processing FPGA accelerator [13], the TERAPIX 128 processing elements SIMD array FPGA accelerator [8], and multi-cores and GPUs using

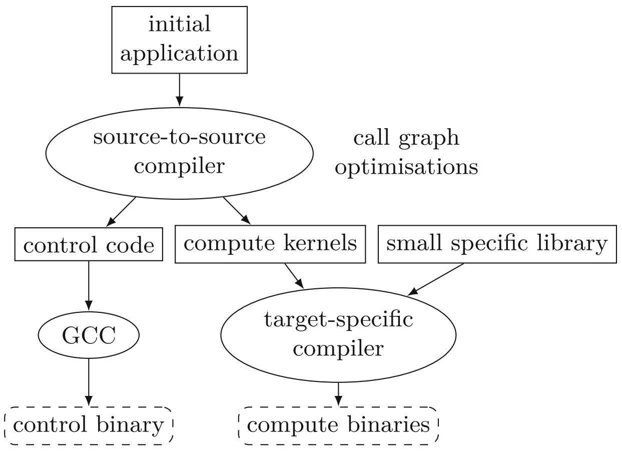


Fig. 1. Overview of our compilation chain

```
freia_aipo_erode_8c(im1, im0, kernel);   // morphological
freia_aipo_dilate_8c(im2, im1, kernel);  // morphological
freia_aipo_and(im3, im2, im0);           // arithmetic
```

Fig. 2. Example FREIA code with a sequence of 3 operations

OpenCL [23]. This new development adds Kalray's MPPA-256 chip as a target hardware by providing a new code generation phase and its corresponding runtime.

The FREIA compiler chain is made of three phases. The first two phases are generic, and the last one is the target specific code generation. Phase 1 builds sequences of basic image operations, out of which large image expressions can be extracted. For this purpose, inlining of composed and user operators, partial evaluation, loop unrolling, code simplifications and dead code suppression are performed. An important preliminary transformation for a dataflow hardware target such as SPoC and MPPA is while-loop unrolling as detailed below in Sect. 4. Phase 2 extracts and optimizes image expressions, as directed acyclic graphs (DAG) of basic image operations. Optimizations include detecting common subexpressions, removing unused image computations, and propagating copies forwards and backwards.

The target execution model depicted in Fig. 3 uses the parallel hardware as an accelerator for heavy image computation, while the host processor runs the control code and I/Os. The runtime environment includes functions for manipulating images such as allocate, receive, emit, and the various operators. The accelerated version has to manage the transfers between the host and the device used for operator computations. For our MPPA target, this is achieved by using named pipes to send images to agents on the host. Theses images are first streamed to the device for computation, then streamed back on a host agent, then back to the main program.

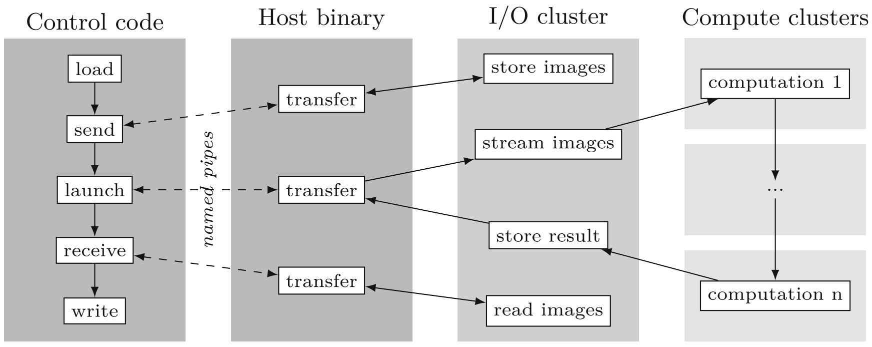


Fig. 3. Summary of our runtime environment

3 Hardware and Software Target

We are targetting the Kalray MPPA-256 many-core architecture through the ΣC dataflow language, compiler and runtime, which allows us to build a runtime for streaming image operators that can be connected to process large expressions.

3.1 MPPA-256 Architecture

Kalray MPPA-256 [17] is a high-efficiency 28 nm *System-on-Chip* featuring 256 compute cores providing 500 GOPs with a typical power consumption of 10 W. Competing MPPA architectures include the Tilera TILE*Pro*64 [3], the Adapteva Epiphany [4], or the TSAR Project [2]. This massively parallel processor aims at a wide range of embedded applications and boasts a fast time-to-market for complex systems.

Figure 4 shows MPPA-256's computes cores divided into sixteen compute clusters. Each of these clusters includes a 2 MB non-coherent shared L2 cache. The compute cores offer instruction parallelism as 32 bits multithreaded *Very Long Instruction Word* cores. SIMD instructions operating on pairs of fixed-point and floating-point instructions are also supported. Every compute cluster also runs a minimalistic real-time Operating System (*NodeOS*) on a separate and dedicated core. This OS manages the 16 compute cores of one cluster by executing multithreaded binary programs onto them. The compute clusters communicate with each other through a high-speed toroidal *Network-on-Chip*.

The MPPA-256 chip also includes four additional clusters for managing external communications. These input/output clusters provide several interfaces, such as PCI-Express for connecting to a host machine, DDR interface to access local RAM, Ethernet or Interlaken for directly connecting several chips together.

We used the MPPA-256 as a hardware accelerator. The chip is placed aside a 4 GB dedicated RAM onto a PCI-Express extension card. This card is then accessed through the PCI-Express bus of a typical computer workstation.

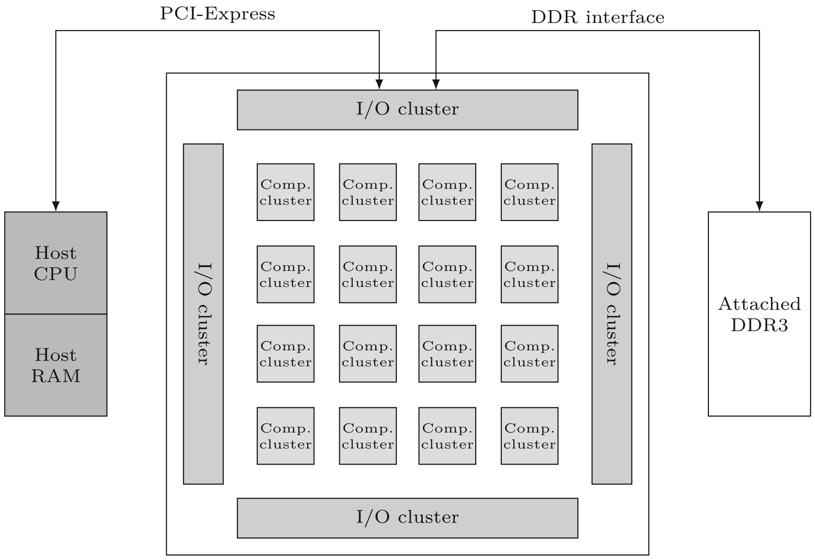


Fig. 4. The MPPA-256 chip and its environment

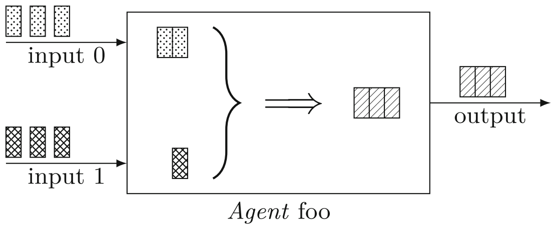


Fig. 5. A ΣC agent with two inputs and one output (see code in Fig. 6)

3.2 The ΣC Programming Language

To ease the programming on their manycore chip [5], Kalray offers the classic parallel libraries PThread and OpenMP [27] as well as a specific dataflow programming language called ΣC [19]. ΣC is a superset of C which provides a dataflow programming model and relies on the concept of *agents*, which are basic compute blocks similar to Kahn Processes [22]. These blocks receive and consume a fixed amount of data from input channels, and produce data on their output channels. The agent represented in Fig. 5 has two input channels and one output channel. When two pieces of data are available on the first input channel and one on the second, the agent produces three pieces of data on its sole output. The corresponding ΣC code is shown in Fig. 6.

This model has two consequences. First, the scheduling of agents on available cores and the inter-agent buffer allocation requires the ΣC compiler to know the number of input/output channels of an agent, and the number of data items processed. This implies that image sizes must be known at compile time.

```
agent foo() {
  interface {              // define I/O channels
    in<int> in0, in1;      // 2 input integer channels
    out<int> out0;         // 1 output integer channel
    spec{in0[2],in1,       // define flow scheduling
        out0[3]};
  }
  void start() exchange    // DO SOMETHING!
    (in0 i0[2], in1 i1, out0 o[3]) {
    o[0] = i0[0], o[1] = i1, o[2] = i0[1];
  }
}
```

Fig. 6. ΣC example: a basic agent merging two integer streams (see Fig. 5)

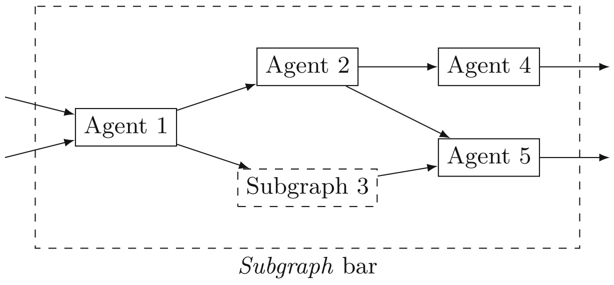


Fig. 7. A ΣC subgraph composed of 4 agents and one subgraph (see code in Fig. 8)

Then, when several independent graphs are mapped, the compiler assumes that these graphs may be active at the same time, thus the mapping reserves non-overlapping memory and cores for the tasks. If only one graph at a time is really active, resources can be under-used.

The performance model implies that tasks must provide significant computations in order to amortize communication costs. In particular, communication times include a constant overhead, which must be amortized with significant data volumes. However, as memory is scarse, it is best to require small inter-task buffers: a trade-off must be made. Another key point of the static dataflow model is that the slowest task in the graph determines the overall performance. Therefore, elementary tasks must be as fast and as balanced as possible.

Agents can be connected to each other in order to compose a ΣC *subgraph* which can recursively compose upper-level subgraphs. A subgraph representation and the corresponding ΣC code are respectively showed in Figs. 7 and 8. The top-level subgraph is called `root` and corresponds to the classic C `main` function. A ΣC agent can be executed either by one of MPPA-256's compute cores or by a core of an input/output cluster. Agents can also be executed by the processor of the host machine, therefore providing access to files. Kalray provides also a ΣC compiler for MPPA-256, which handles the mapping of ΣC agents on the compute cores of their chip.

```
subgraph bar() {
  interface {                        // define I/O channels
    in<int> in0[2];
    out<int> out0, out1;
    spec{ { in0[][3]; out0 }; { out1[2] } };
  }
  map {
    agent a1 = new Agent1();         // instantiate agents
    agent a2 = new Agent2();
    agent a3 = new Subgraph3();
    agent a4 = new Agent2();
    agent a5 = new Agent4();
    connect (in0[0], a1.input0);     // I/O connections
    connect (in0[1], a1.input1);
    connect (a4.output, out0);
    connect (a5.output, out1);
    connect (a1.output0, a2.input);  // internal connections
    connect (a1.output1, a3.input);
    connect (a2.output0, a4.input);
    connect (a2.output1, a5.input0);
    connect (a3.output, a5.input1);
  }
}
```

Fig. 8. ΣC example: a basic subgraph (see graph in Fig. 7)

The ΣC programming language provides an effective way to take advantage of the MPPA-256, and serves as the main target language for demonstrating our automatic streamization compiler and runtime.

4 Compiler and Runtime Design

A DAG produced by our compilation chain has a structure similar to streaming programs. Indeed, image analysis operators can be directly transposed as ΣC agents, and DAGs as ΣC subgraphs.

4.1 ΣC Image Processing Library

Aside from our compilation chain, we developed a ΣC library of elementary image analysis operators. Each operator is implemented as one ΣC agent, such as: (1) arithmetic operators performing elementary operations onto pixels of input images; (2) morphological operators [30], which are the more compute intensive operators; (3) reduction operators returning a scalar value.

The 2 MB per compute cluster memory limit implies that our ΣC agents cannot operate on a whole image. Since the transition between two states of an agent is rather slow, we cannot afford to operate on one pixel at a time. Thus our agents process images line by line. Measuring per pixel execution time of several applications for several input image sizes (see Fig. 9) reveals that larger lines are computed more efficiently than shorter ones, as communication times are better amortized. This also simplifies the implementation of stencil operators by easing the access to neighboring pixels.

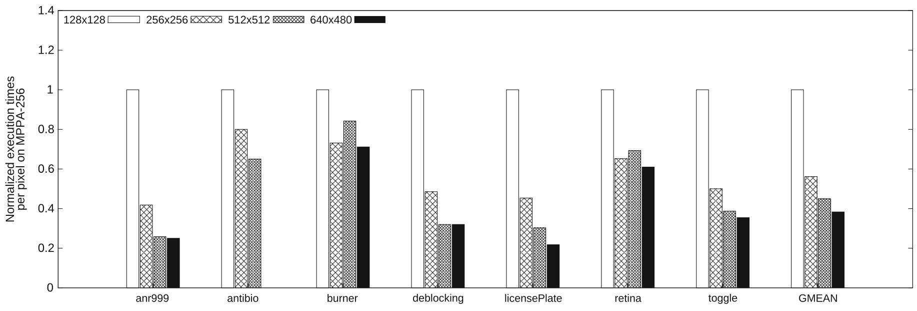
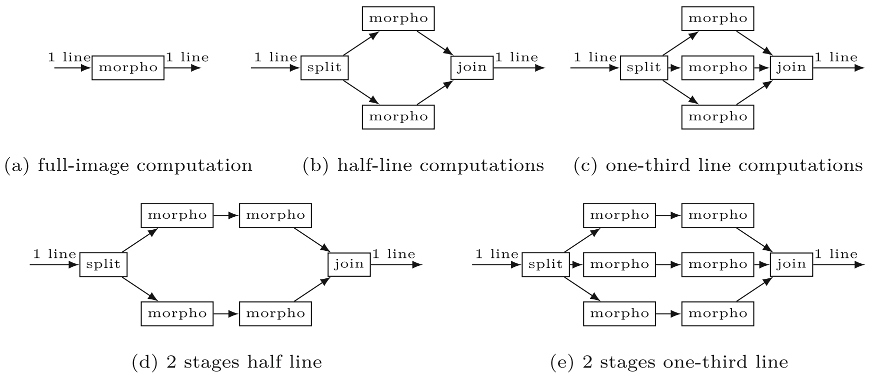


Fig. 9. Impact of the image size on the average execution time per pixel



1 line
morpho
split
join
(a) full-image computation
(b) half-line computations
(c) one-third line computations
(d) 2 stages half line
(e) 2 stages one-third line

Fig. 10. Analysed cases of data parallelism for morphological operators

The morphological agents, the most complex operators, have a direct impact on the performance of our applications. Consisting of an aggregate function (min, max, average) on the value of a subset of the neighbor pixels, they are often used in large pipelines in image analysis applications. We used several optimizations during their implementation. Firstly, as stencil operators, they need to access not only the current processed line, but also the previous and the next lines. As a consequence, each agent has a 3-line internal buffer to store the input lines needed for computation. Also, the incoming input lines are stored into this 3-lines buffer and processed in a round-robin manner, avoiding time-consuming copies. Finally, these agents benefitted from an optimized assembly kernel to use guarded instructions not automatically generated by the compiler.

We also investigated data parallelism by splitting input lines and computing each portion with several morphologic agents. This approach allows us to take advantage of the MPPA-256 unused compute cores, since application DAGs are usually much smaller than the number of available cores. Because morphological operators are stencils, we use overlapping lines when splitting and joining. Our measures showed that having several stages of computing agents in the (d) and (e) cases of Fig. 10 slows down the whole process, so we focused on comparing

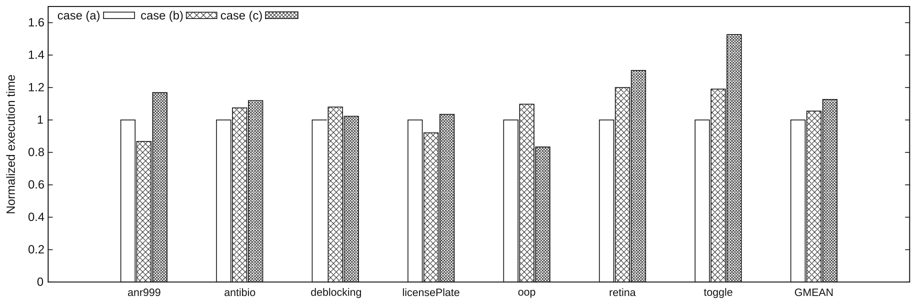


Fig. 11. Execution times with parallel morphological operators

```
subgraph foo() {
  int16_t kernel[9] = {0,1,0, 0,1,0, 0,1,0};
  ...
  agent ero = new img_erode(kernel);
  agent dil = new img_dilate(kernel);
  agent and = new img_and_img();
  ...
  connect(ero.output, dil.input);
  connect(dil.output, and.input);
  ...
}
```

Fig. 12. Corresponding ΣC code to FREIA code in Fig. 2

the (a), (b) and (c) 1-stage cases represented in Fig. 10. As shown in Fig. 11, the results are quite mixed and application-dependent. Replacing one morphological agent by four or more agents induces more inter-cluster communications, leading to a loss in performance, even if computed data is reduced by half.

4.2 ΣC Code Generation

As stated in Sect. 2, our compilation chain produces DAGs of elementary image analysis operators from which we generate ΣC subgraphs using our image analysis agents. For example, Fig. 12 shows an extract of the generated ΣC code from the FREIA source code in Fig. 2.

In order to be correctly transposed into a running dataflow program, we must ensure that there is no scalar dependency between two agents of the same subgraph. Indeed, since images are processed on a per line basis, a scalar produced from a whole image cannot be applied on the lines of the same image by an other agent on the same subgraph without causing lines accumulation in inter-agent buffers, and thus major performance loss. A split-on-scalar-dependencies pass is used ahead of our ΣC generator to provide scalar-independent DAGs, which can then be transposed directly to ΣC subgraphs.

Some complex image analysis operators involve a convergence loop over an image-dependent parameter. Such operators, being idempotents, can be unrolled with no consequences on the final result. However, this unrolling pass leads to a greater number of generated ΣC agents, and thus an increase occupation of the MPPA compute cores. We measured the influence of the unrolling factor of these particular loops on the execution times of the relevant applications (see Fig. 13). Our results show that unrolling dramatically increases the performance of our applications. For these applications, an unrolling factor of 8 leads to a fair speedup while mobilizing a reasonable amount of compute cores.

Split and unrolled image expression DAGs are then encoded as ΣC subgraphs. Our implementation of the generation of ΣC subgraphs is pretty straighforward: for each vertex of one image expression DAG, our compiler PIPS generates a ΣC instantiation statement for the corresponding ΣC agent first, then connection statements between the agent instance and its predecessors or successors in the DAG. Small differences between the input DAGs structure and the ΣC dataflow model have been addressed during this implementation: (1) since the number of inputs and outputs of our ΣC agents are predetermined, we have to insert replication agents when required; (2) DAG inputs and outputs are specific cases and must be dealt with separately; (3) scalar dependencies must be provided to the correct agents by a dedicated path. Similarly, scalar results must be sent back to the host.

In the dataflow model, the slowest task has the greatest impact on the global execution. Since arithmetic operators do little computation compared to morphologic ones, we investigated the fusion of connected arithmetic operators into compound ΣC agents, thus freeing some under-used compute cores. We implemented this pass on top of our ΣC generator. We tested our optimization pass onto several applications with a variable number of merged operators. Execution time results (Fig. 14) show little to no difference in performance compared to the reference one agent/operator application. These measures confirm that aggregated operators are not limiting the global execution while freeing computing power, therefore validating our approach.

4.3 Runtime Environment

ΣC code generated by our compilation chain often includes several independant and non-connected subgraphs that are all mapped on the MPPA cores. In order to launch the adequate subgraphs at the right time and to control the I/Os, we developed a small runtime in C. It runs on top of the generated ΣC applications

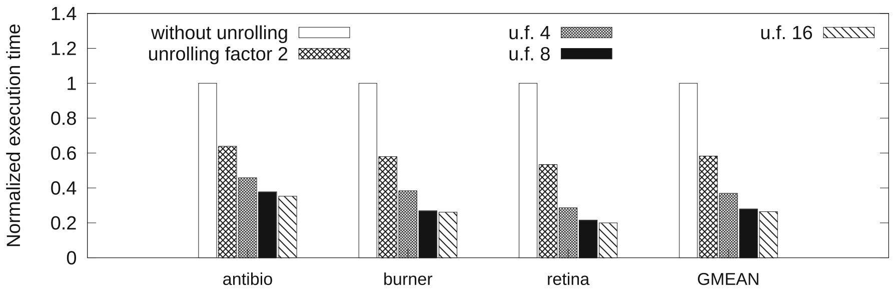


Fig. 13. Relative execution time as a function of unrolling factor

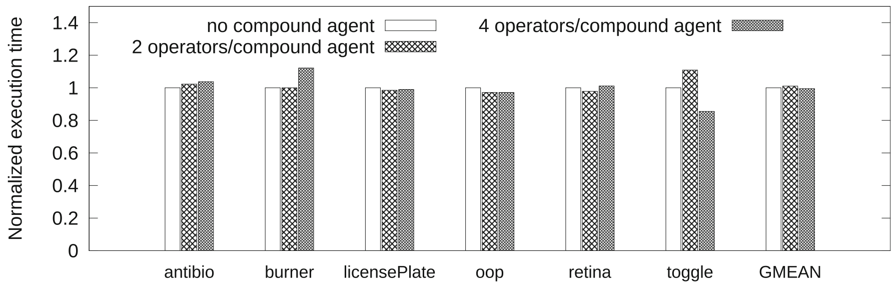


Fig. 14. Influence of the fusion of connected arithmetic operators on execution times

and communicates with them through Unix named pipes, as depicted in Fig. 3. For each function of this runtime, we added a dedicated ΣC subgraph with agents mapped on the host CPU and the I/O clusters cores to handle the task. This runtime is also used for loading and saving images from the host hard drive. To this end, we use a software implementation of FREIA, called Fulguro [11].

The other dedicated control functions allow us to allocate and free the accelerator embedded RAM, to send or receive images to or from the MPPA, and to launch a compute subgraph onto one or more images. On the ΣC side of the application, several independant subgraphs manage control signals sent through named pipes and transfer them to the chip.

The general design of our compilation chain, based on a source code generator, an elementary library and a small runtime environment, allowed us to quickly get functionnal applications on the MPPA-256. With the implementation of a set of specific optimizations (loop unrolling, fusion of fast tasks, splitting of slow tasks, bypassing of the MPPA RAM), we were able to take better advantage of the compute power of this processor. Once this was done, we compared the MPPA-256 to a set of hardware accelerators running the same applications generated by the same compiler.

5 Performance Results

We have evaluated our compilation chain with eight real image analysis applications covering a wide range of cases. Table 1 shows that our applications contain from 15 elementary operators (*toggle*) to more than 400 (*burner*), most of them morphological operators. Table 1 also illustrates the number of ΣC subgraphs generated by our compilation chain, and the number of occupied compute clusters when running on the MPPA-256. These applications generally include less than three independent image expression DAGs.

Our compilation chain targets several software and hardware accelerators: a reference software implementation, Fulguro [11], on an Intel Core i7-3820 quad-core CPU running at 3.6 GHz with an average power consumption of 130 W; ΣC code running directly on the same CPU (iΣC); the MPPA-256 processor (10 W) using ΣC; SPoC [13] and Terapix (TPX) [8], two image processing accelerators implemented on a FPGA (26 W); two CPUs using OpenCL [23]: an

Table 1. Characteristics of used image analysis applications

Apps.	#operators				#subgraphs	#clusters	image size
	arith	morph	reduc	total			
anr999	0	20	3	23	1	2	224 × 288
antibio	8	41	25	74	8	6	256 × 256
burner	18	410	3	431	3	16	256 × 256
deblocking	23	9	2	34	2	10	512 × 512
licensePlate	4	65	0	69	1	5	640 × 383
oop	7	10	0	17	1	2	350 × 288
retina	15	38	3	56	3	4	256 × 256
toggle	8	6	1	15	1	1	512 × 512

Intel dual-core (i2c - 65 W) and an AMD quad-core CPU (a4c - 60 W); and three NVIDIA GPUs again with OpenCL [23]: a GeForce 8800 GTX (120 W), a Quadro 600 (40 W) and a Tesla C 2050 (240 W).

We compared the output of our compilation chain from the previously described applications onto these 8 hardware targets, both in terms of execution times and energy consumption. For the MPPA-256 chip, time and energy measures were obtained using Kalray's `k1-power` utility software, ignoring transfers and control CPU consumption. Time figures for the FPGAs, CPUs and GPUs were taken from [14]. Energy figures were derived from target power consumption. Results are shown in Tables 2 and 3, with best performances in bold. Compared to the Fulguro monothreaded sofware implementation, ΣC on CPU is relatively slow, due to the numerous threads communicating with each other. To take individual MPPA-256 compute cluster power supply into account, we added in Table 3 a column "MPPA ideal" representing the energy of clusters

Table 2. Execution times (ms) of our applications on the different hardware targets

Apps	Software		Hardware							
				FPGA		OpenCL				
	Flgr	iΣC	MPPA	SPoC	TPX	i2c	a4c	GTX	Quadro	Tesla
anr999	4.3	36.5	8.3	**0.9**	3.5	12.2	7.2	9.8	1.5	**0.9**
antibio	80.2	1026	670	**41.9**	88.5	254.1	135.3	204.3	57.4	93.4
burner	795.3	814.4	113	**17.2**	83.8	162.2	124.4	321.0	576.2	142.0
deblocking	141.4	121.0	84	30.7	11	25.1	16.7	11.1	3.5	**1.3**
licensePlate	483.9	152.7	20.2	13.3	36.8	32.2	21.9	36.6	7.3	**2.3**
oop	4.0	39.3	11.3	124.6	63.3	12.3	8.3	5.8	1.8	**1.0**
retina	149.0	222.5	95	**7.4**	32.4	93.5	55.4	75.5	60.8	33.9
toggle	6.2	69.8	22.6	12.6	4.3	15.0	9.0	6.3	1.4	**0.7**
AVG/MPPA	1.13	3.28	1.00	0.32	0.49	0.85	0.54	0.64	0.23	0.12

Table 3. Energy (mJ) used by our test cases on different targets

Apps.	Software		Hardware								
			MPPA		FPGA		OpenCL				
	Flgr	iΣC	real	ideal	SPoC	TPX	i2c	a4c	GTX	Quadro	Tesla
anr999	559	4745	50	*6*	**23**	91	793	432	1176	60	221
antibio	10425	133380	3500	*1313*	**1089**	2301	16517	8118	24516	2296	22883
burner	103390	105900	**388**	*388*	447	2179	10543	7464	38520	23048	34790
deblocking	18382	15730	431	*269*	798	286	1632	1002	1332	**140**	319
licensePlate	62907	24284	**120**	*38*	354	957	2093	1314	4392	292	564
oop	520	5110	**59**	*7*	3240	1646	800	498	696	72	245
retina	19370	28925	487	*122*	**192**	842	6078	3324	9060	2432	8306
toggle	806	9074	119	*7*	328	112	975	540	756	**56**	172
AVG/MPPA	28.78	83.38	**1.00**	*0.25*	1.65	2.52	10.81	6.33	15.01	1.79	5.45

actually used for the computations, according to Table 1. Disconnecting unused compute cores would provide us an extra reduction in energy consumption.

These results show that although MPPA/ΣC is not faster than dedicated hardware targets, it provides the lowest average energy consumption for tested applications. The high degree of task parallelism induced by the use of the ΣC dataflow language on the 256 cores of the MPPA-256 processor is thus a strength facing dedicated hardware in low energy and embedded applications.

6 Conclusion and Future Work

We added a new hardware target to the FREIA ANR project: a 256 cores processor with a power consumption of 10 W through the use of the ΣC dataflow programming language. Using the PIPS source-to-source compiler, we generated ΣC dataflow code based upon a small image analysis library written in ΣC. The execution of the generated applications relies on a small runtime environment controlling the execution of the different ΣC subgraphs mapped on the cores of the MPPA-256 processor. We implemented a set of specific optimizations from automatic fast operator aggregation to data-parallel slow operators to achieve better performance. The performance of our approach is shown by comparing the MPPA-256 results to other hardware accelerators using the same compilation chain. MPPA/ΣC proves to be the most energy-efficient programmable target, which competes in performance with specific image-processing dedicated hardware such as the SPoC FPGA processor.

In the current approach, several subgraphs are mapped onto different compute cores, meaning only a fraction of the chip is used at a given time. Future work includes the investigation of dynamically mapping distinct ΣC subgraphs on the same cores when they do not need to be run concurrently. Another way to save energy, especially for small applications, would be the ability to disconnect unused clusters within the chip, as shown in column "MPPA ideal" in Table 3. More performance improvements could also be obtained on some applications by generating automatically kernel-specific convolutions, which would reduce execution time by skipping altogether null-weighted pixels.

Aknowledgements. Thanks to Danielle Bolan and Pierre Jouvelot for proof-reading, Antoniu Pop for his advises and bibliography pointers, Kalray engineers Frédéric Blanc, Jérôme Bussery and Stéphane Gailhard for their support and to anonymous reviewers whose comments greatly helped to improve this paper.

References

1. The streamit language (2002). http://www.cag.lcs.mit.edu/streamit/
2. Tera-scale architecture (2008). https://www-asim.lip6.fr/trac/tsar/wiki
3. The TilePro64 many-core architecture (2008). http://www.tilera.com/
4. The Epiphany many-core architecture (2012). http://www.adapteva.com/
5. Aubry, P., Beaucamps, P.E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., Dupont de Dinechin, B., Galea, F., Goubier, T., Harrand, M., Jones, S., Lesage, J.D., Louise, S., Chaisemartin, N.M., Nguyen, T.H., Raynaud, X., Sirdey, R.: Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In: Alexandrov, V.N., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J., Sloot, P.M.A. (eds.) ICCS. Procedia Computer Science, pp. 1624–1633. Elsevier, Amsterdam (2013)
6. Bandishti, V., Pananilath, I., Bondhugula, U.: Tiling stencil computations to maximize parallelism, November 2012
7. Bilodeau, M., Clienti, C., Coelho, F., Guelton, S., Irigoin, F., Keryell, R., Lemonnier, F.: FREIA: Framework for Embedded Image Applications (2008–2011). freia.enstb.org, French ANR-funded project with ARMINES (CMM, CRI), THALES (TRT) and Télécom Bretagne
8. Bonnot, P., Lemonnier, F., Edelin, G., Gaillat, G., Ruch, O., Gauget, P.: Definition and SIMD implementation of a multi-processing architecture approach on FPGA. In: Design Automation and Test in Europe, pp. 610–615. IEEE, December 2008
9. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: a generic distributed DAG engine for high performance computing. Parallel Comput. **38**(1–2), 37–51 (2012). http://linkinghub.elsevier.com/retrieve/pii/S0167819111001347
10. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines, p. 363. ACM Press (2010). http://portal.acm.org/citation.cfm?doid=1806596.1806638
11. Clienti, C.: Fulguro image processing library. Source Forge (2008)
12. Clienti, C.: Architectures flots de données dédiées autraitement d'images par la Morphologie MATHÉMATIQUE. Ph.D. thesis, MINES ParisTech, September 2009
13. Clienti, C., Beucher, S., Bilodeau, M.: A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In: EUSIPCO: European Signal Processing Conference, August 2008
14. Coelho, F., Irigoin, F.: API compilation for image hardware accelerators. ACM Trans. Archit. Code Optim. **9**(4), 1–25 (2013)
15. CRI, MINES ParisTech: PIPS (1989–2012). pips4u.org, open source research compiler, under GPLv3

16. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008: Conference on Supercomputing, pp. 1–12. IEEE Press (2008)
17. Dupont de Dinechin, B., Sirdey, R., Goubier, T.: Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In: Procedia Computer Science, vol. 18 (2013)
18. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. ACM SIGPLAN Not. **41**(11), 151 (2006). http://portal.acm.org/citation.cfm?doid=1168918.1168877
19. Goubier, T., Sirdey, R., Louise, S., David, V.: ΣC: a programming model and language for embedded manycores. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011, Part I. LNCS, vol. 7016, pp. 385–394. Springer, Heidelberg (2011)
20. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
21. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: an overview of the PIPS project. In: Proceedings of ICS 1991, pp. 244–251. ACM Press (1991)
22. Kahn, G.: The semantics of a simple language for parallel programming. p. 5 (1974)
23. KHRONOS group: OpenCL computing language v1.0, December 2008
24. Le Guernic, P., Benveniste, A., Bournai, P., Gautier, T.: Signal-a data flow-oriented language for signal processing. IEEE Trans. Acoust. Speech Signal Process. **34**(2), 362–374 (1986)
25. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
26. Murthy, P.K., Lee, E.A.: Multidimensional synchronous dataflow. IEEE Trans. Signal Process. **50**, 3306–3309 (2002)
27. OpenMP architecture review board: OpenMP application program interface, Version 3.0, May 2008
28. Pop, A.: Leveraging streaming for deterministic parallelization - an integrated language, compiler and runtime approach. Ph.D. thesis, MINES ParisTech, September 2011
29. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: PLDI 2013, p. 12, June 2013
30. Soile, P.: Morphological Image Analysis. Springer, Heidelberg (2003)
31. Stephens, R.: A Survey Of Stream Processing. Springer, Heidelberg (1995)