Automatic Streamization of Image Processing Applications

LCPC 2014

Pierre Guillou Fabien Coelho François Irigoin

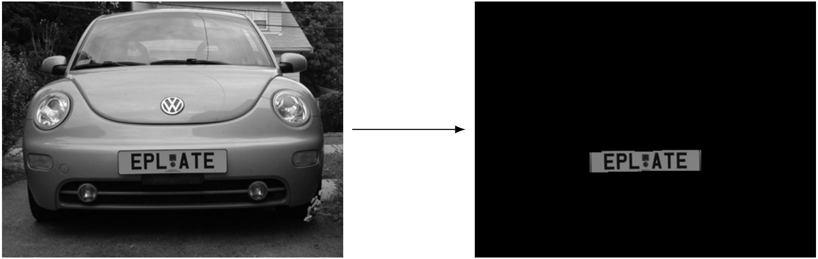
MINES ParisTech, PSL Research University

Hillsboro, OR, September 15, 2014

MINES ParisTech

Context

- Image processing applications



- Computing systems
  - CPUs (multi/many cores)
  - Accelerators (GPUs, FPGAs. . .)

DSL ⟶ Streaming Language ⟶ Manycore Accelerator

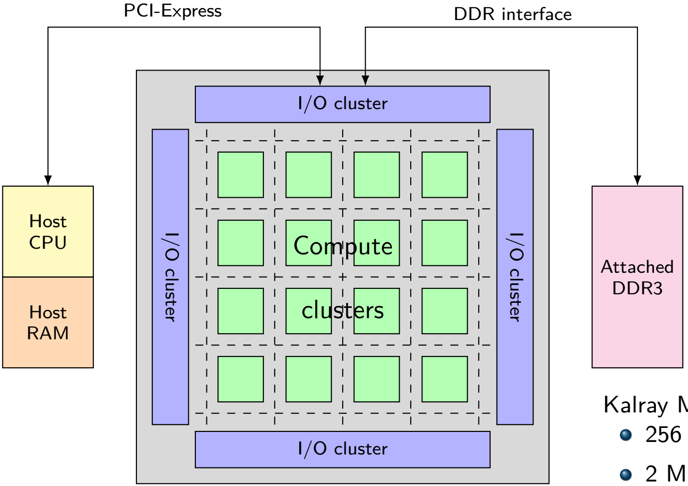
Domain Specific Languages:

- High-level
- Easy-to-use
- Hardware agnostic
- C Embedded language: **FREIA**

Streaming languages:

- Target easily multi/many cores architectures
- Image processing applications
- Verbose
- Examples: StreamIt, **Sigma-C**

Manycore Processor



Kalray MPPA-256:

- 256 VLIW cores
- 2 MB/cluster
- 10 W

Outline

Image Processing DSL: FREIA

FRamework for Embedded Image Applications:

- Sequential Embedded C code
- High-level image processing operators
- Example:

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

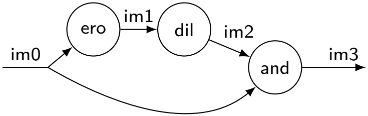
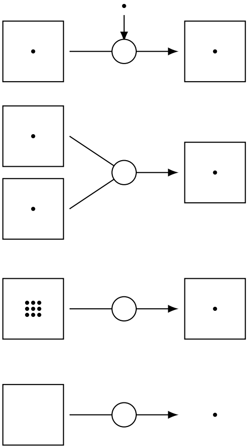
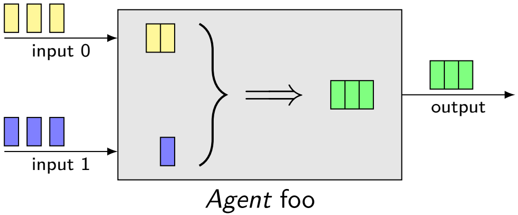


Image Operators

- Arithmetic operators
  - unary
  - binary
  - $+ \;\; - \;\; \times \;\; / \;\; \text{min} \;\; \text{max} \;\; = \;\; \& \;|\; \sim$
- Morphological operators
  - selection + min/max/avg
- Reduction operators
  - min/max/sum

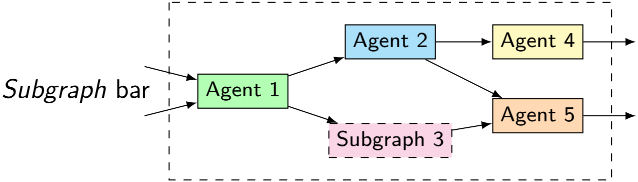


Sigma-C Agents




```
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];
  }
}
```

From Agents to Subgraphs




```
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 a3 = new Subgraph3();
    ...
    connect (in0[0], a1.input0); // I/O connections
    ...
    connect (a5.output, out1);
    connect (a1.output0, a2.input); // internal connections
    ...
    connect (a3.output, a5.input1);
  }
}
```

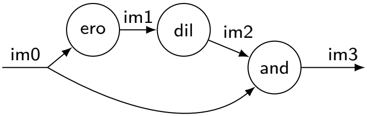
Input & Output

- From FREIA sequential C code:

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

- To Sigma-C subgraph:

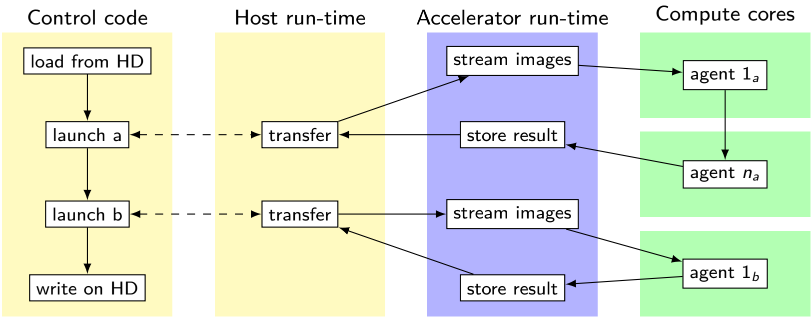
```
subgraph foo() {
  int16_t kernel[9] = {0,1,0, 0,1,0, 0,1,0};
  ...
  agent ero = new img_erode(kernel);
  agent dil = new img_dilate(kernel);
  agent and = new img_and_img();
  ...
  connect(ero.output, dil.input);
  connect(dil.output, and.input);
  ...
}
```

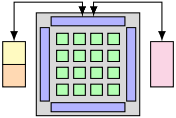


From DSL Code to Streaming Code

1. Build sequences of basic image operations
   - composed operator inlining
   - partial evaluation
   - loop unrolling
2. Extract and optimize image expressions $\longrightarrow$ DAG
   - common subexpression elimination
   - unused image computations removal
   - copy propagation
3. Generate target code
   - 1 DAG $\rightsquigarrow$ 1 subgraph
   - 1 vertex $\rightsquigarrow$ 1 agent
   - Subgraph activation
4. Use image operator library

Execution Scheme





Mapping Sigma-C Graphs

Graph throughput constraints:

- Slowest node in critical path

$\Longrightarrow$ split slow nodes, merge fast nodes

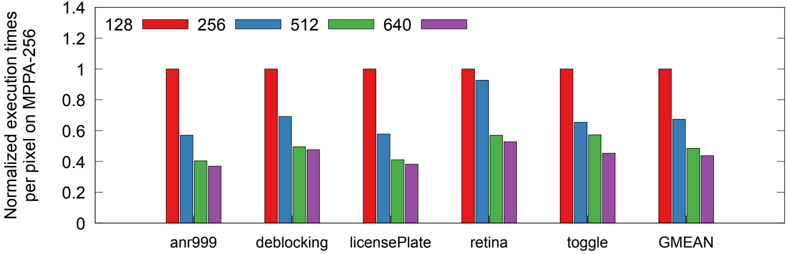
Agent constraints:

- 1 agent / compute core — $\sum$ *agents* $\leq$ *256*
- 2 MB for 16 cores — *mem(1 agent)* $\leq$ *128 kB*
- Fixed iteration overhead — *pack pixels*

Mapping constraints:

- NoC comms between clusters — *use few clusters*
- Constant activation time — *use few large graphs*

Agent Granularity



- Fixed iteration overhead $\longrightarrow$ pack pixels
- Small memory $\longrightarrow$ avoid large structures
- Stencil ops $\longrightarrow$ manage overlap

$\Longrightarrow$ **operate on image rows**

Optimization of Morphological Agents

Morphological agents are the bottlenecks:

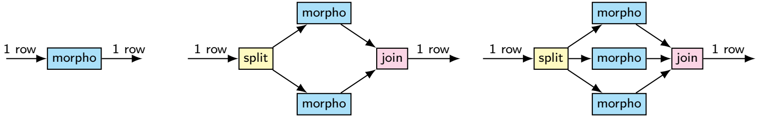
- 3×3 boolean matrix mask for selecting neighbors
- min, max or avg on selected neighbors
- Often combined in deep pipelines

Some optimizations have been implemented:

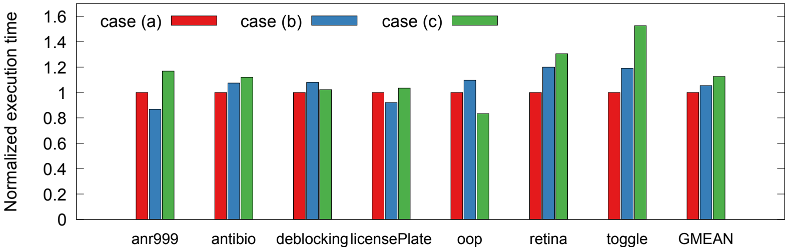
- Agent buffer of 3 rows fed in a round-robin manner
- Innermost loop written in VLIW assembly code

Bottleneck Reduction: Graph Transformation

Data Parallelization of Morphological Agents



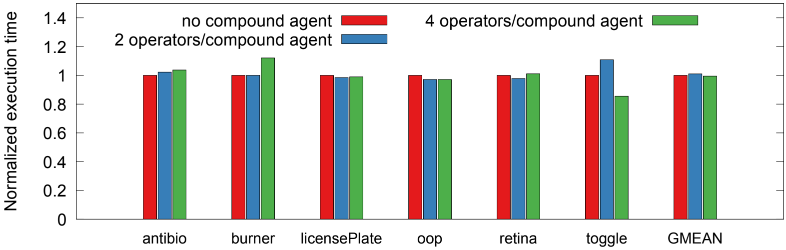
(a) one row (b) two half-rows (c) three thirds of a row



Reduce Number of Used Cores: Graph Transformation

Aggregation of Arithmetic Agents

- Fast agents can be aggregated to use fewer cores $\sum agents \leq 256$
- Arithmetic operators are fast: good candidates for aggregation

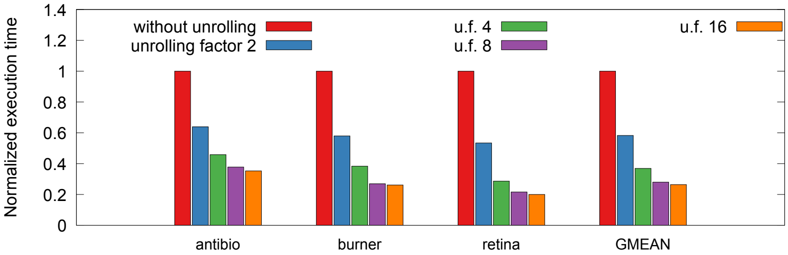


$\Longrightarrow$ **fewer cores used/same execution time**

Reduce Control Overhead: Enlarge Graphs

While Unrolling for Convergent Transformations

```
do {
  p = c;        // p and c depend on the processed image
  ...           // a converging operation
  freia_aipo_global_vol(img, &c);
} while(c != p);
```



- #control overhead ↘
- #agents ↗
- #speculative execution ↗

⟹ **tradeoff: unroll by 8**

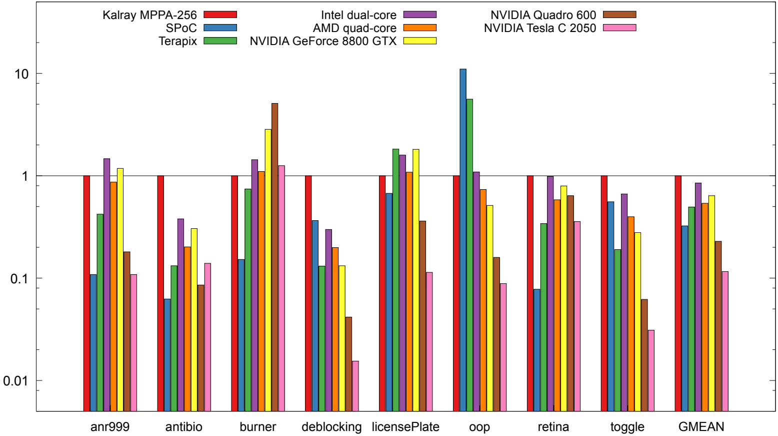
Benchmark Suite

Apps.	LoC	#operators				#subg	#clust	image size
		arith	morpho	red	Total			
anr999	87	1	20	2	23	1	2	224×288
antibio	200	8	41	25	74	8	6	256×256
burner	510	18	410	3	431	3	16	256×256
deblocking	161	23	9	2	34	2	10	512×512
licensePlate	203	4	65	0	69	1	5	640×383
oop	442	7	10	0	17	1	2	350×288
retina	469	15	38	3	56	3	4	256×256
toggle	143	8	6	1	15	1	1	512×512

Target Systems

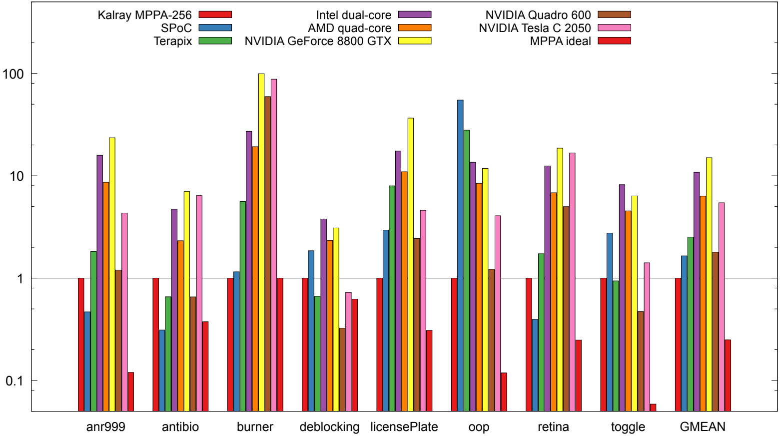
Targets	hardware kind	backend	max W
SPoC	FPGA	FPGA	26
Terapix	FPGA	FPGA	26
Intel dual-core	2c CPU	OpenCL	65
AMD quad-core	4c CPU	OpenCL	60
NV Geforce GTX 8800	GPU	OpenCL	120
NV Quadro 600	GPU	OpenCL	40
NV Tesla 2050C	GPU	OpenCL	240
Kalray MPPA-256	Manycore	Sigma-C	10

Relative Execution Times



Reference: MPPA = 1.0

Relative Energy Consumption



Reference: MPPA = 1.0

Conclusion

Summary:

- Image processing DSL $\longrightarrow$ streaming language
- Using a source-to-source compiler
- Targetting manycore processors

Contributions:

- Generation of Sigma-C subgraphs from FREIA applications
- Optimizations for running onto the Kalray MPPA-256
- Energy results: MPPA can compete with dedicated accelerators

Future Work:

- Better use of the MPPA compute power
  - Map non-concurrent subgraphs on the same cores
  - Power off unused clusters
- Automatic generation of specific convolutions with partial evaluation
- Exploit data parallelization when profitable

Automatic Streamization of Image Processing Applications

LCPC 2014

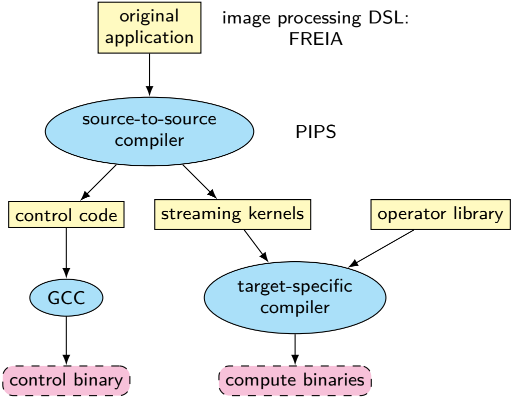
Pierre Guillou   Fabien Coelho   François Irigoin

MINES ParisTech, PSL Research University

Hillsboro, OR, September 15, 2014

MINES ParisTech

Compilation Chain



Applicability

Other manycore targets:

- Intel Xeon Phi
  - $\sim$ 60 cores on an interconnect ring
  - no clusters, no shared memory
  - 512 kB L2 cache/core
- Tilera TILE-Gx
  - up to 72 cores with L1 and L2 cache
  - no clusters, no shared memory
  - 2d NoC

Other streaming languages:

- StreamIt
  - agents $\rightsquigarrow$ *filters*
  - subgraphs $\rightsquigarrow$ *pipelines*/*splitjoins*/*feedback loops*