API-Compilation for Image Hardware Accelerators

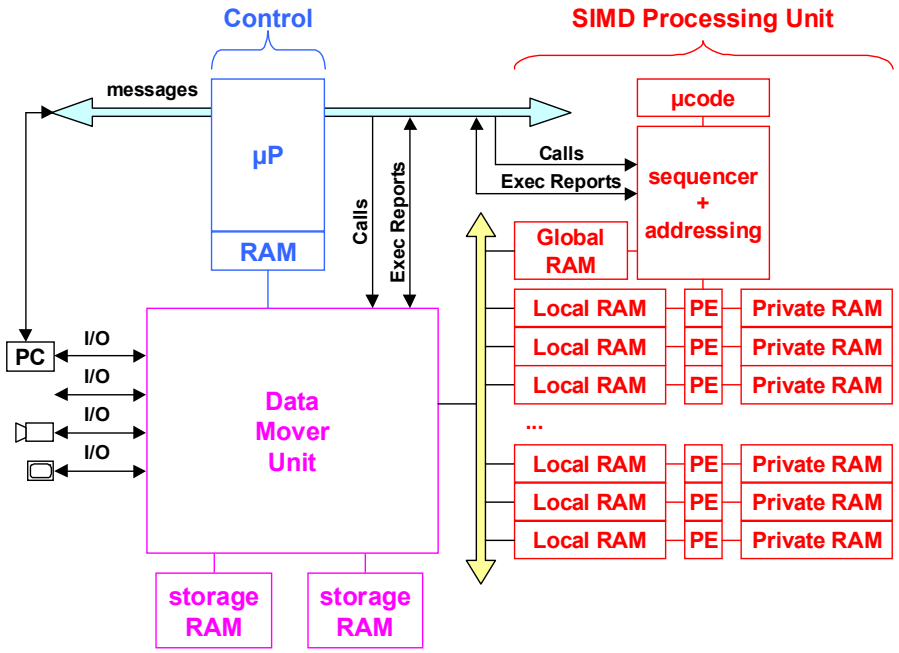
Fabien Coelho & François Irigoin

ANR project: FREIA software environment for

image application development on modern architectures

THALES TELECOM Bretagne ARMINES MINES ParisTech

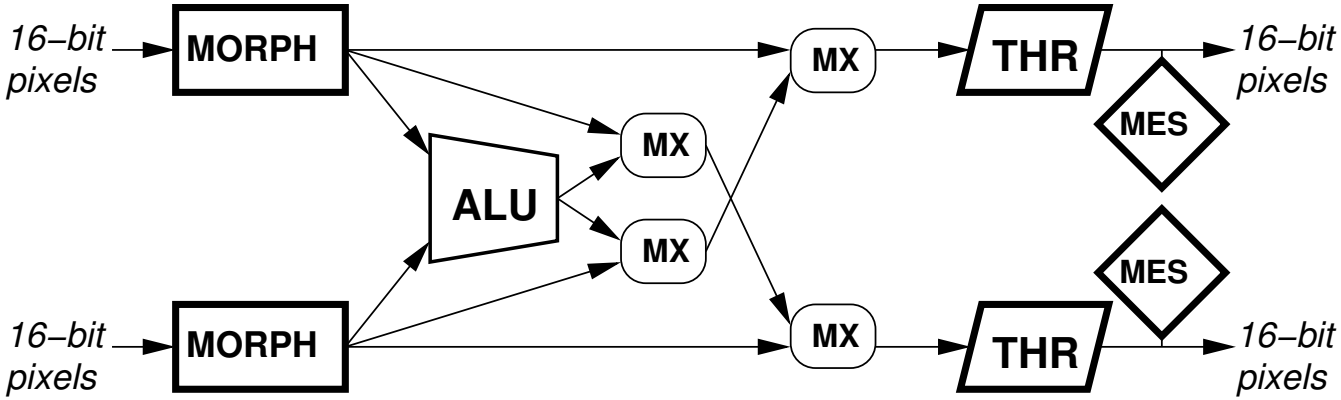
Terapix Hardware Accelerator



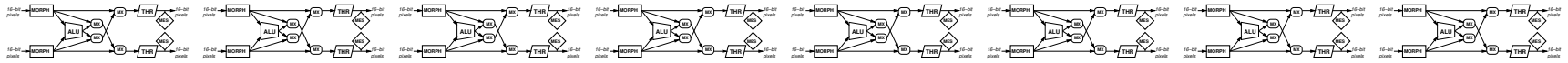
- μP + 128 SIMD PE array, 1024 pixels per PE, neighbor coms
- computation // communication (in or out) *double buffer*
- issues: small memory implies tiles, **5.3** pixels/cycle bandwidth with DDR

SPoC Hardware Accelerator Vector Unit

2 paths, 5 image ops + reductions, 4 pixels/cycle bandwidth



Pipeline of 8 units



Portability vs Performance?

Portability write one generic code

Performance re-write code for every accelerator

(Pure) Library Approach?

- domain-specific API, optimized (by hand)
- small library: not enough operator aggregation, missed opportunities
- large library: cost? portability? *VSIPL 1000s functions*

(Pure) Compiler Approach?

- start from source, inline functions, loop fusion. . .
- issues: complexity, impact of stencils, conditions for borders. . .

Mixed Library/Compiler Approach

Input small domain-specific image-level API in plain C

basic/composed operators relevant to application developers

library implemented (optimized?) by hand – quickly available

Locality hardware and runtime handle loop fusion details!

SPoC: delay lines with cyclic buffers

Terapix: overlapping tiling induces redundant computations, μ-code

Compilation get ops, merge ops, schedule, allocate

ANR999: running example excerpt

```
// SKIPPED declarations and inits
freia_common_rx_image(in, &fin);      // INPUT

freia_global_min(in, &min);           // COMPUTE
freia_global_vol(in, &vol);
freia_dilate(od, in, 8, 10);
freia_gradient(og, in, 8, 10);

printf("min=%d, vol=%d\n", min, vol); // OUTPUT
freia_common_tx_image(od, &fout);
freia_common_tx_image(og, &fout);
```

Compilation Strategy

Standard techniques for low-cost implementation

1. Build large basic blocks of elementary operations: **generic**

   inlining, scalar const. prop., loop unroll., dead-code elimination

2. Build and optimize DAGs of image operations: **generic**

   constant propagation, CSE, SDC, copy propagation

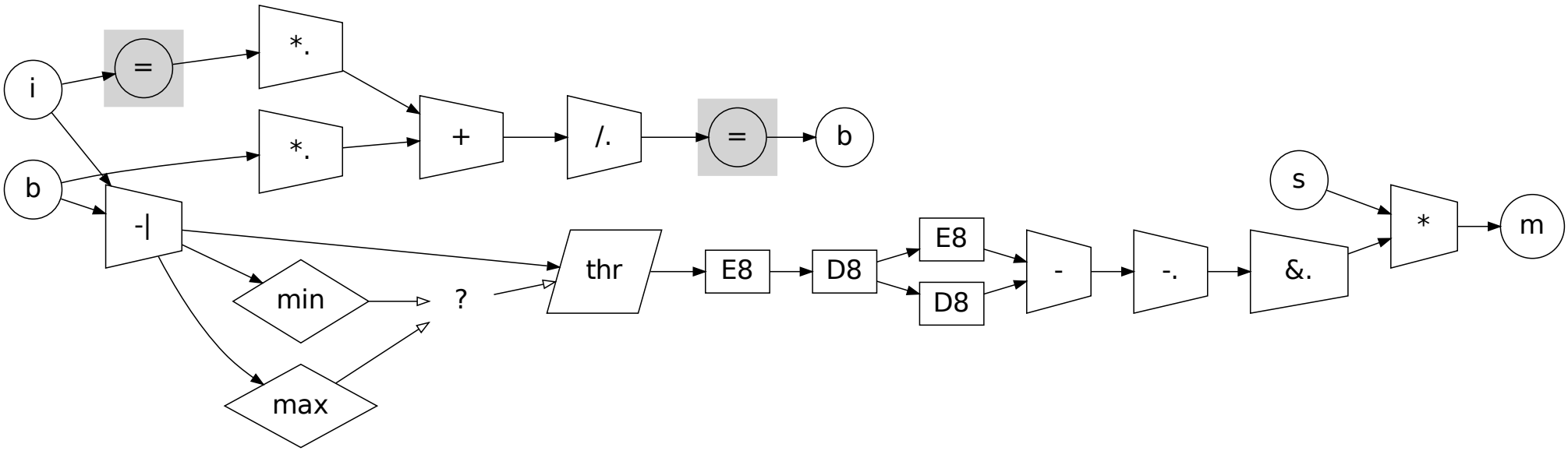
3. Generate code for target: **specific**

   SPoC: DAG splitting and scheduling, compaction, cutting

   Terapix: DAG splitting, scheduling, memory allocation

   OpenCL: DAG splitting, simple operation aggregation

2.1 Build Image Expression DAG



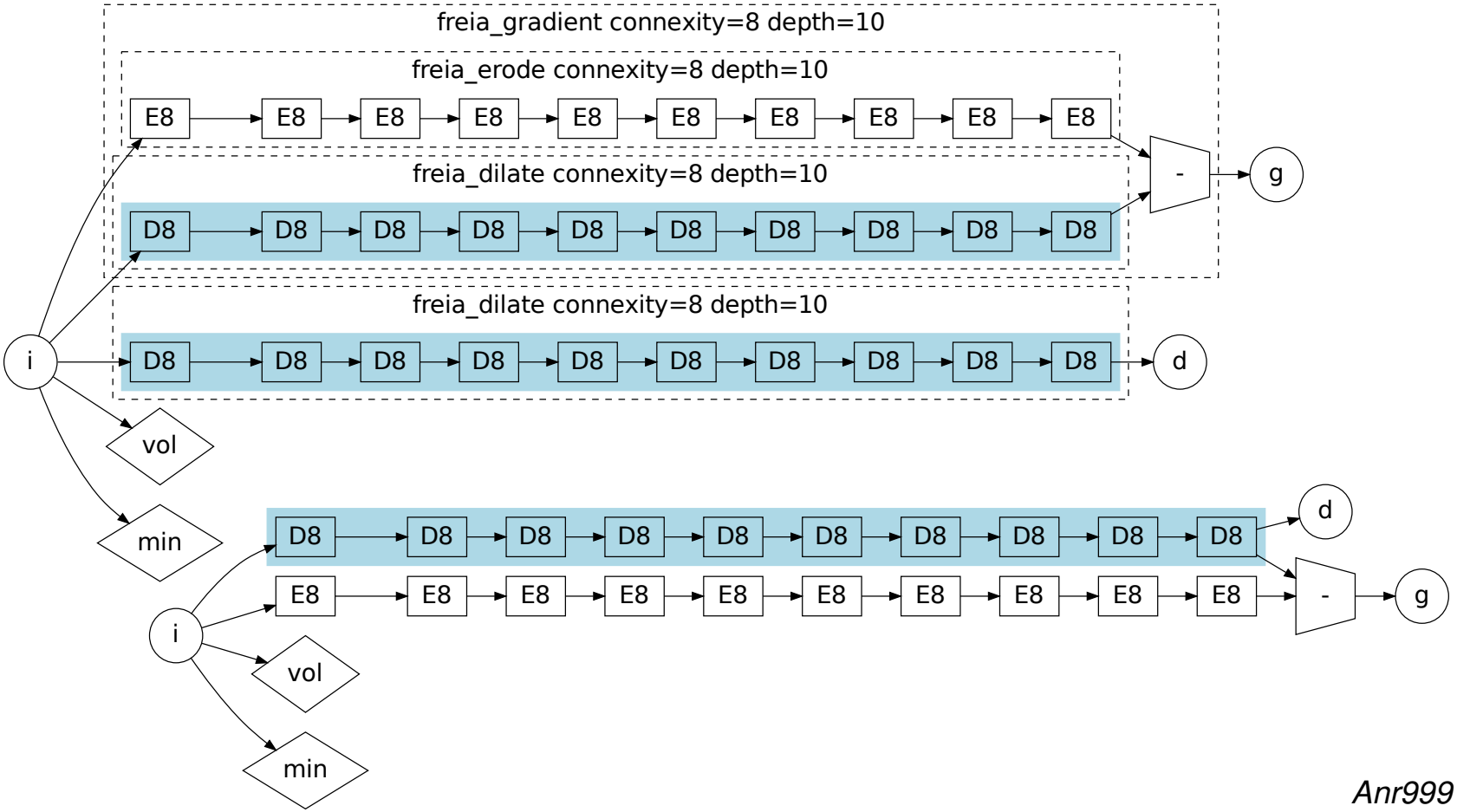
from Video Survey

- expression DAG of simple image operations

  morpho, ALU, threshold, measure, copies, scalar ops

- arrows: image and scalar dependencies

2.2 Optimize DAG

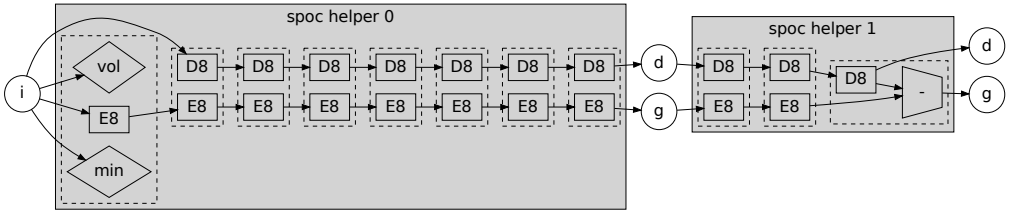


Anr999

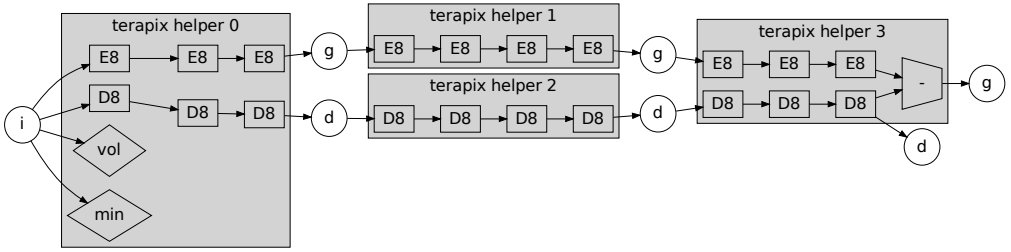
3. Target-dependent code generator

mostly NP-Complete, greedy heuristics to split DAG and schedule ops

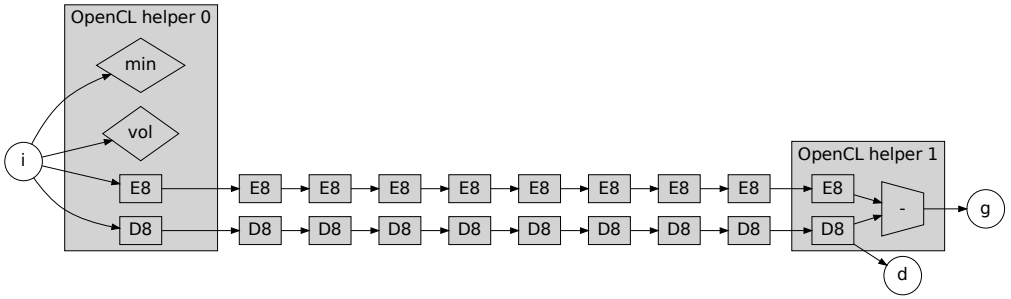
SPoC



Terapix



OpenCL



Performance aggregated speedups for 9 applications

Hardware	Target	H/L	L/C	H/C
FPGA	**SPoC**	14.2	**6.5**	91.5
Accelerators	**Terapix**	20.5	**2.3**	47.6
Multi-cores	**Intel dual-core**	0.9	**2.0**	1.9
OpenCL	**AMD quad-core**	1.3	**2.7**	3.5
GPGPU	**GeForce 8800 GTX**	–	**7.8**	–
NVIDIA	**Quadro 600**	–	**22.1**	–
OpenCL	**Tesla C 2050**	–	**10.2**	–

H one thread on host, **L** library version, **C** compiled version

Implementation in PIPS: add 5% to code base

- source-to-source, easier to debug output
- phase 1 – reuse (more or less) standard phases: 155000 LOCs
- phase 2 – DAG building, optimization, utils: 4000 LOCs
- phase 3 – code generation for three targets: 4400 LOCs

  SPoC 1900 LOCs **Terapix** 1400 LOCs **OpenCL** 1100 LOCs

`http://pips4u.org/`

Benefits: Cost effective reusable applications!

Portability through small common API

Performance through high-level *coarse-grain* low-cost compilation

Key success factors

Co-design API / compiler / runtime / hardware

- overlapping tiling moved from compiler to runtime
- double buffers moved from runtime to compiler
- borders management moved to runtime and hardware

Source-to-source ease development and testing

Functional simulators help testing

Applicability

Apps quite *static* (but not only!) structure and behavior

API one data type, few dozen ops, a lot of parallelism

Hardware well suited, hides loop fusion...

Future Work

- Kalray MPPA data-flow model target?
- new applications? new transformations?
- consider other application domains?

Questions?

Hardware Accelerators

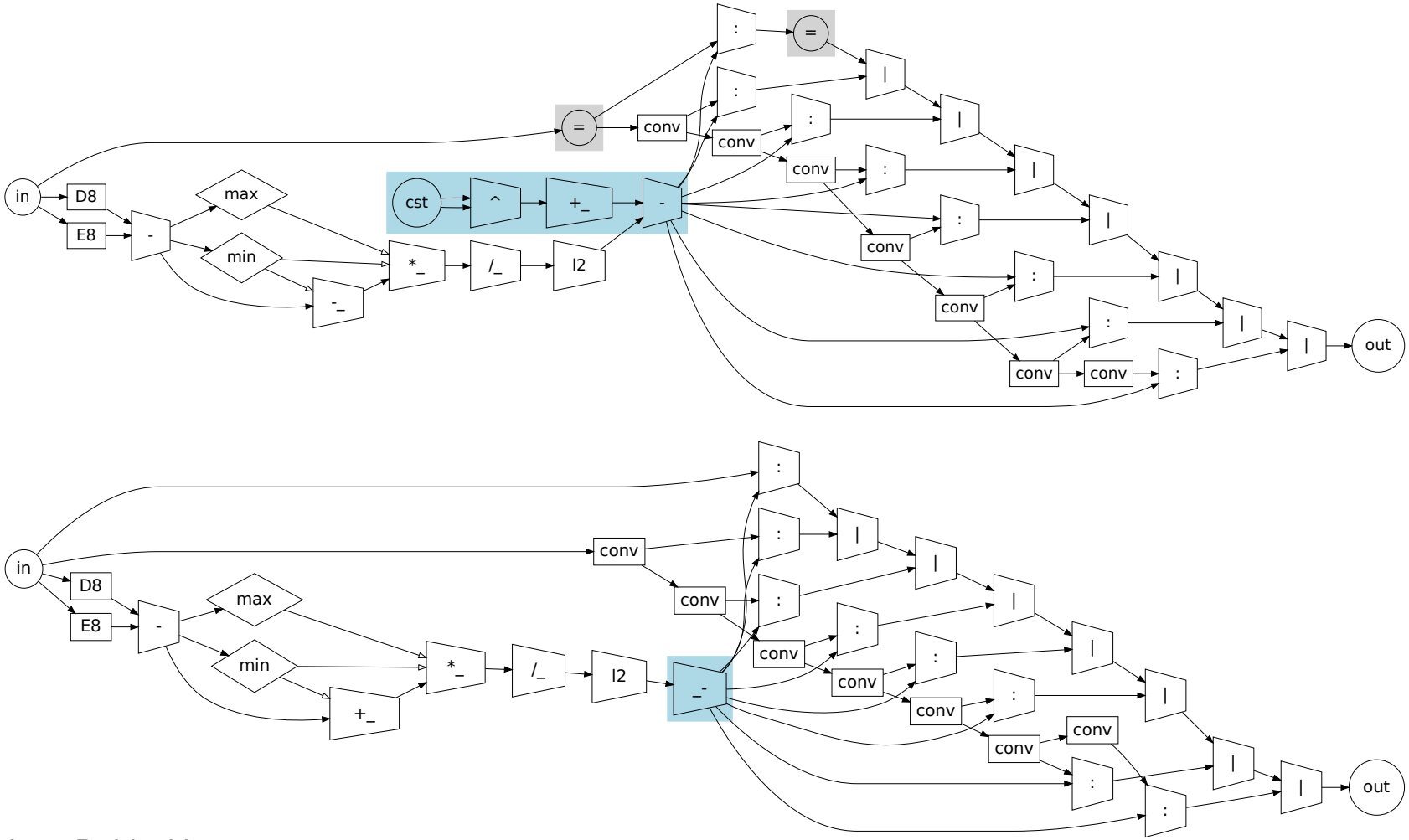
- more or less domain specific
- ASIC, FPGA, GPGPU, multi-cores...
- embedded? real-time? systems

Motivation?

- better execution time
- lower energy footprint
- (hide) intellectual property
- product life time: up to 30 years

Two accelerators: **Terapix** (128 PE SIMD) and **SPoC** (chained vector)

2.2 Optimize DAG (1)



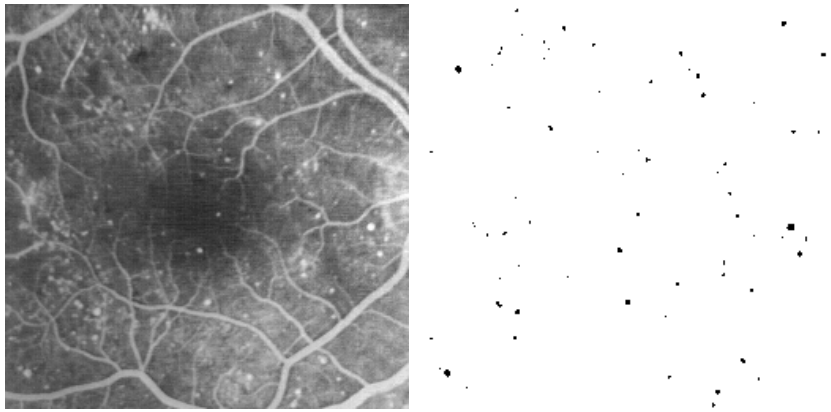
from Deblocking

Application Domain: image processing

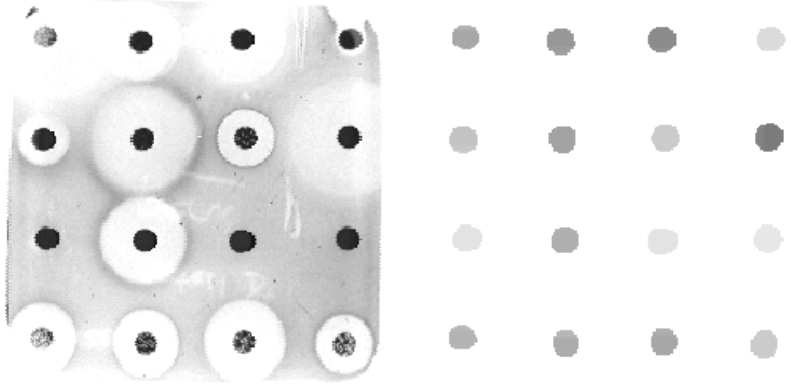
algebra on images: one data type, basic (hw) and composed ops

OOP (22 ops)

Retina (106 ops)



Antibio (49 ops)



Burner (422 ops)