FREIA SPoC Compilation

Fabien Coelho – fabien.coelho@mines-paristech.fr

François Irigoin – francois.irigoin@mines-paristech.fr

THALES TELECOM Bretagne MINES ParisTech ARMINES

typeset with LaTeX, revision 956

FREIA: FRamework for Embedded Image Applications

`http://freia.enstb.org/`

software environment

to speed-up image application development

on modern architectures

- soft & hard targets: fulguro, SPoC, Ter@pix, OpenCL (GPGPU)
- how: Image API + coarse/medium/fine grain compilation

Partners on the project

ARMINES/MINES CMM mathematical morphology, hardware

MINES/ARMINES CRI compilation

TELECOM Bretagne INFO compilation

THALES TRT applications, hardware environment

Motivation: Bridging the Gap

FREIA portable image library API Thales

- image I/Os, memory management, debug
- atomic (basic) operators *AIPO* morpho, arith, linear, red...
- complex (composed) operators *CIPO* dilate, gradient, opening...

SPoC vector hardware accelerator CMM/Thales Christophe Clienti

- dozens of elementary operators
- two-path pipeline of vector units
- FPGA implementation and functional software simulator

Applications

LP: License plate – character extraction

OOP: Out-of-position – airbag trigger prevention

VS: Video Survey – motion detection

ANR999: small (slightly simplified) running example

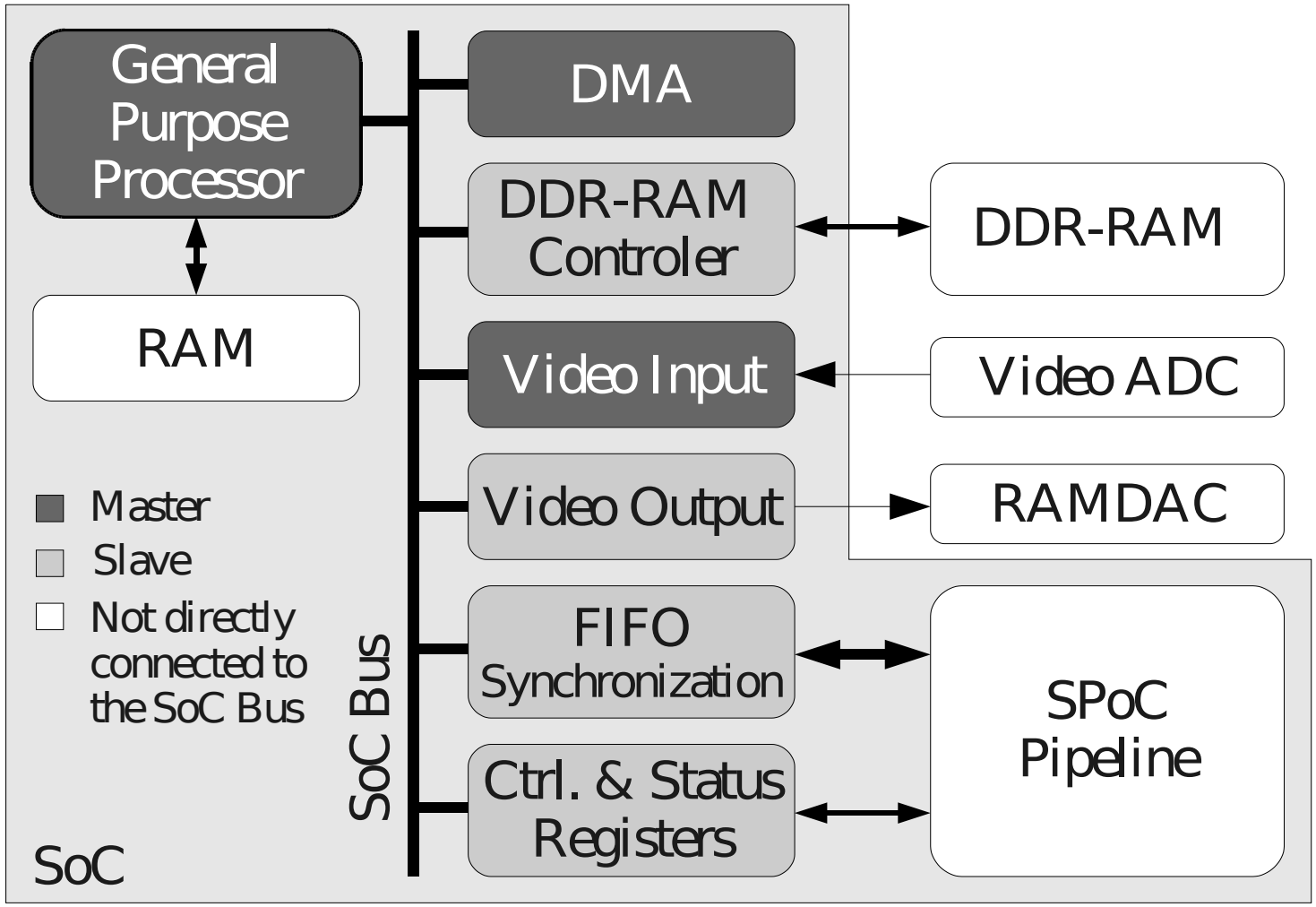
```
#include <stdio.h>
#include <freia.h>
int main(void) {
  freia_dataio fin, fout;
  freia_data2d *in, *og, *od;
  int32_t min, vol;
  // initializations
  freia_common_open_input(&fin, 0);
  freia_common_open_output(&fout, 0, ...)
  in = freia_common_create_data(fin.bpp, ...);
  od = freia_common_create_data(fin.bpp, ...);
  og = freia_common_create_data(fin.bpp, ...);
```

```
// get input image
freia_common_rx_image(in, &fin);

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

// output results
printf("input global min = %d\n", min);
printf("input global volume = %d\n", vol);
freia_common_tx_image(od, &fout);
freia_common_tx_image(og, &fout);
```

```
  // cleanup
  freia_common_destruct_data(in);
  freia_common_destruct_data(od);
  freia_common_destruct_data(og);
  freia_common_close_input(&fin);
  freia_common_close_output(&fout);
  return 0;
}
```

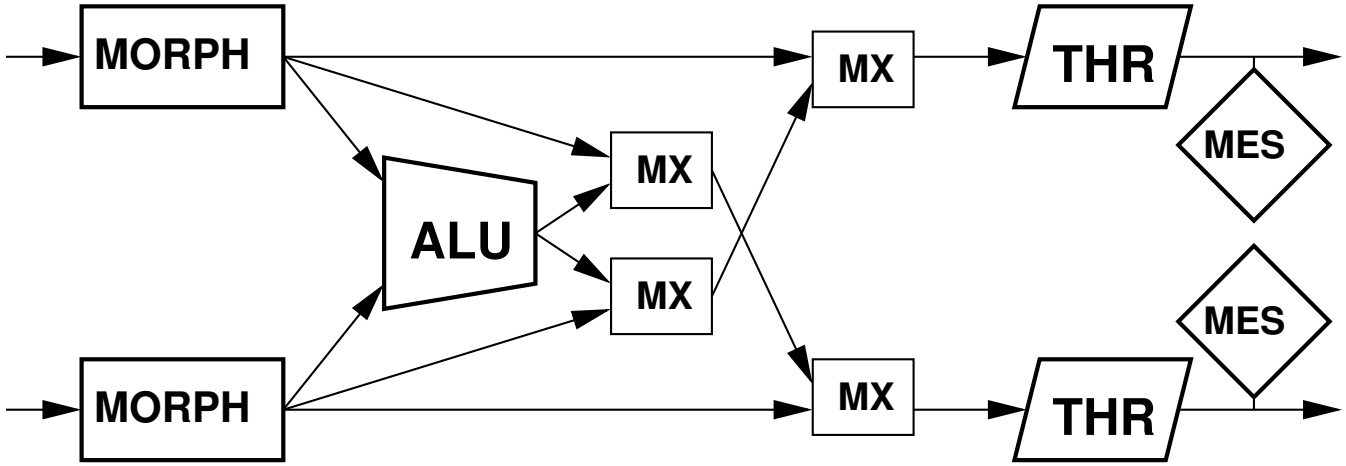
Target Architecture

SPoC Heterogeneous Architecture

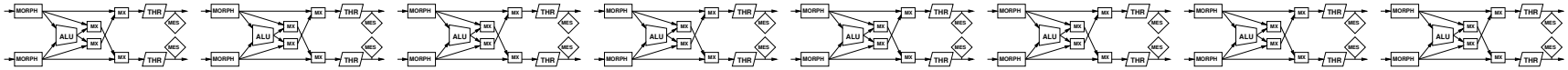


SPoC Vector Unit

2 paths, 5 image ops + reductions



Pipeline of 8 units



SPoC Hardware Constraints

- available operators

  others (general convolutions) must be computed outside

- 2 image paths

  2 image on input, 2 lives, 2 image on output

  rigid stucture, with multiplexer selection

- pipeline length: from 8 to 32

- no internal scalar dependency

- cost: total number of accelerator calls

Compilation Strategy

Standard techniques for low-cost implementation

to replace high-cost manual mapping

1. build basic blocks of elementary operations *generic*

   inlining, partial eval, full unroll, dead code elim, flatten code

2. build and optimize DAGs of image operations *generic*

   build DAG, CSE, copy propagation

3. generate code for SPoC target *specific*

   DAG splitting and scheduling, compaction, cutting

Why not try loop transformations?

- software implementation based on highly parametric generic code
- a lot of intermediate copies, allocations, checks. . .
- no obvious image semantics
- not needed at all!

Fulguro *dilate* implementation

```
void flgr2d_native_dilate_8_connexity_fgINT16(FLGR_Data2D *nhb)
{
  FLGR_Data2D *nhbcopy;
  fgINT16 **seodd, **seeven, **se;
  int i, j, k, l, m, n;
  fgINT16 valse;
  FLGR_Vector *vecPixValue = flgr_vector_create(nhb->spp,
nhb->type);
  FLGR_Vector *vecPixMax = flgr_vector_create(nhb->spp,
nhb->type);
  FLGR_Vector *vecSeValue = flgr_vector_create(nhb->spp,
nhb->type);
```

```
nhbcopy = flgr2d_create_from(nhb);
flgr2d_copy(nhbcopy, nhb);
seodd = (fgINT16 **) flgr_malloc(sizeof(fgINT16 *)*3);
seodd[0] = (fgINT16 *) flgr_malloc(sizeof(fgINT16)*3);
seodd[1] = (fgINT16 *) flgr_malloc(sizeof(fgINT16)*3);
seodd[2] = (fgINT16 *) flgr_malloc(sizeof(fgINT16)*3);
seeven = (fgINT16 **) flgr_malloc(sizeof(fgINT16 *)*3);
seeven[0] = (fgINT16 *) flgr_malloc(sizeof(fgINT16)*3);
seeven[1] = (fgINT16 *) flgr_malloc(sizeof(fgINT16)*3);
seeven[2] = (fgINT16 *) flgr_malloc(sizeof(fgINT16)*3);
flgr2d_set_data_array_fgINT16(seodd, 0, 0, 1);
flgr2d_set_data_array_fgINT16(seodd, 0, 1, 1);
flgr2d_set_data_array_fgINT16(seodd, 0, 2, 1);
flgr2d_set_data_array_fgINT16(seodd, 1, 0, 1);
```

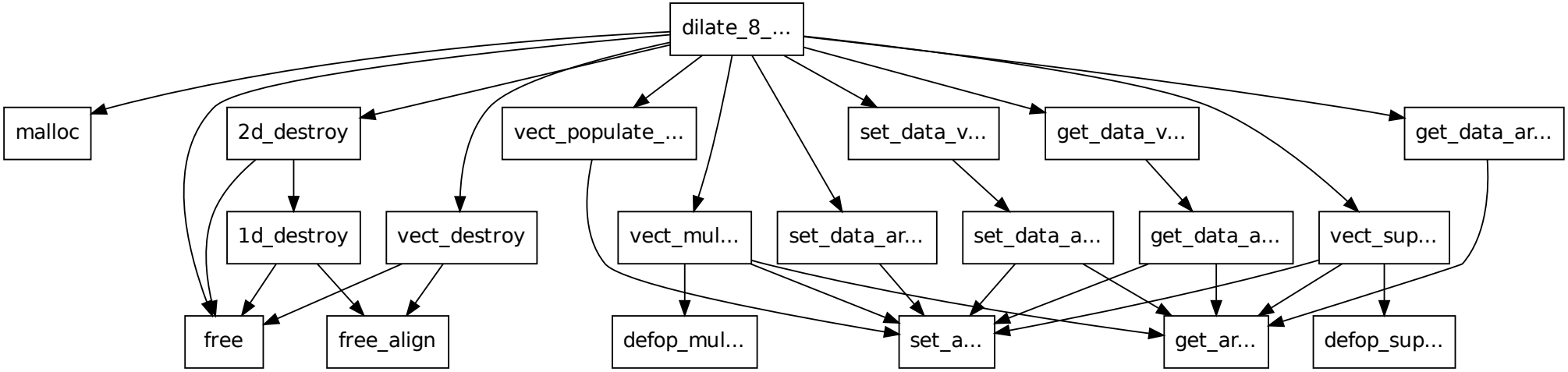
```
flgr2d_set_data_array_fgINT16(seodd, 1, 1, 1);
flgr2d_set_data_array_fgINT16(seodd, 1, 2, 1);
flgr2d_set_data_array_fgINT16(seodd, 2, 0, 1);
flgr2d_set_data_array_fgINT16(seodd, 2, 1, 1);
flgr2d_set_data_array_fgINT16(seodd, 2, 2, 1);
flgr2d_set_data_array_fgINT16(seeven, 0, 0, 1);
flgr2d_set_data_array_fgINT16(seeven, 0, 1, 1);
flgr2d_set_data_array_fgINT16(seeven, 0, 2, 1);
flgr2d_set_data_array_fgINT16(seeven, 1, 0, 1);
flgr2d_set_data_array_fgINT16(seeven, 1, 1, 1);
flgr2d_set_data_array_fgINT16(seeven, 1, 2, 1);
flgr2d_set_data_array_fgINT16(seeven, 2, 0, 1);
flgr2d_set_data_array_fgINT16(seeven, 2, 1, 1);
flgr2d_set_data_array_fgINT16(seeven, 2, 2, 1);
```

```
  for (i = 0;i<nhb->size_y;i++)
    for (j = 0;j<nhb->size_x;j++) {
      flgr_vector_populate_from_scalar_fgINT16(vecPixMax, 0);
      for (k = i-1, m = 0;k<=i+1;k++, m++)
        if (k>=0&&k<nhb->size_y)
          for (l = j-1, n = 0;l<=j+1;l++, n++)
            if (l>=0&&l<nhb->size_x) {
              se = i%2==1?seodd:seeven;
              valse = flgr2d_get_data_array_fgINT16(se, m, n);
              flgr_vector_populate_from_scalar_fgINT16(vecSeValue,
valse);
              flgr2d_get_data_vector_no_norm_fgINT16(nhbcopy, k,
l, vecPixValue);
              flgr_vector_mult_fgINT16(vecPixValue, vecPixValue,
```

```
vecSeValue);
                flgr_vector_sup_fgINT16(vecPixMax, vecPixMax,
vecPixValue);
              }
        flgr2d_set_data_vector_fgINT16(nhb, i, j, vecPixMax);
      }
    flgr_vector_destroy(vecPixValue);
    flgr_vector_destroy(vecPixMax);
    flgr_vector_destroy(vecSeValue);
    flgr2d_destroy(nhbcopy);
    flgr_free(seodd[0]);
    flgr_free(seodd[1]);
    flgr_free(seodd[2]);
    flgr_free(seodd);
```

```
    flgr_free(seeven[0]);
    flgr_free(seeven[1]);
    flgr_free(seeven[2]);
    flgr_free(seeven);
    return;
}
```

Fulguro *dilate* callgraph



1. Build Basic Blocs

sequences of image operations

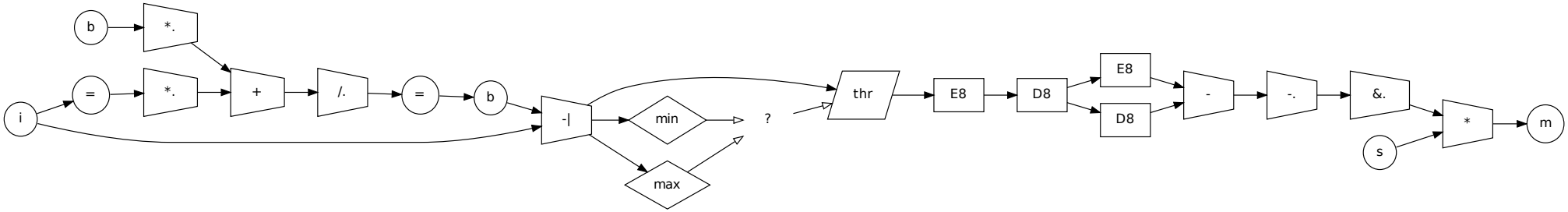
- inlining down to *basic* operations

  stop at basic operations

- partial eval, unroll loops, flatten block. . .
- result for ANR999 example

```
freia_global_min(in, &min);
freia_global_vol(in, &vol);
freia_dilate_8c(od, in, k8c);
freia_dilate_8c(od, od, k8c);    // x 10
I_0 = 0;
tmp = freia_common_create_data(...);
freia_dilate_8c(tmp, in, k8c);
freia_dilate_8c(tmp, tmp, k8c); // x 10
freia_erode_8c(og, in, k8c);
freia_erode_8c(og, og, k8c);     // x 10
freia_sub(og, tmp, og);
freia_common_destruct_data(tmp);
```

2.1 Build Image Expression DAG



from video survey

- expression DAG of simple image operations

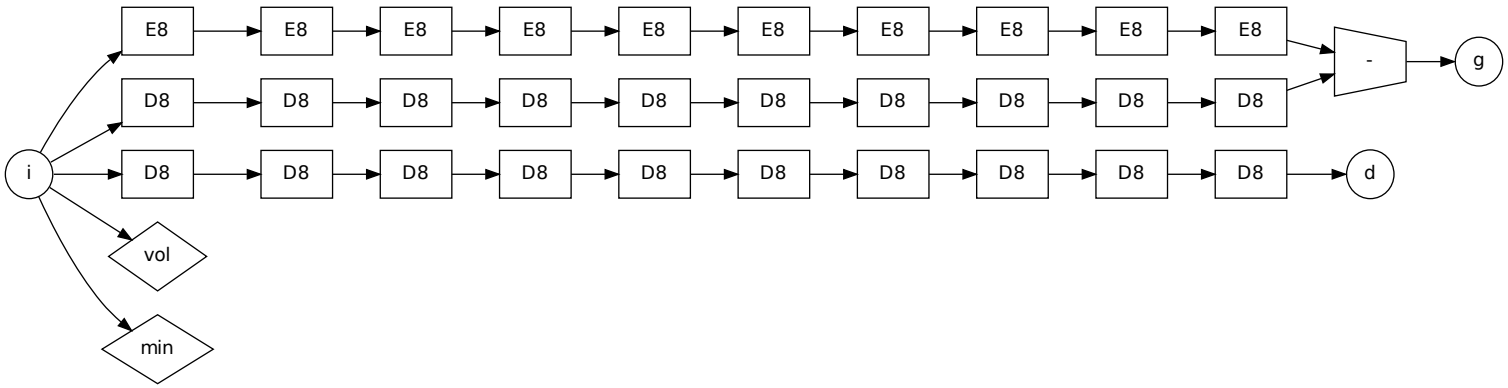
  morpho, ALU, threshold, measure, copies, scalar ops

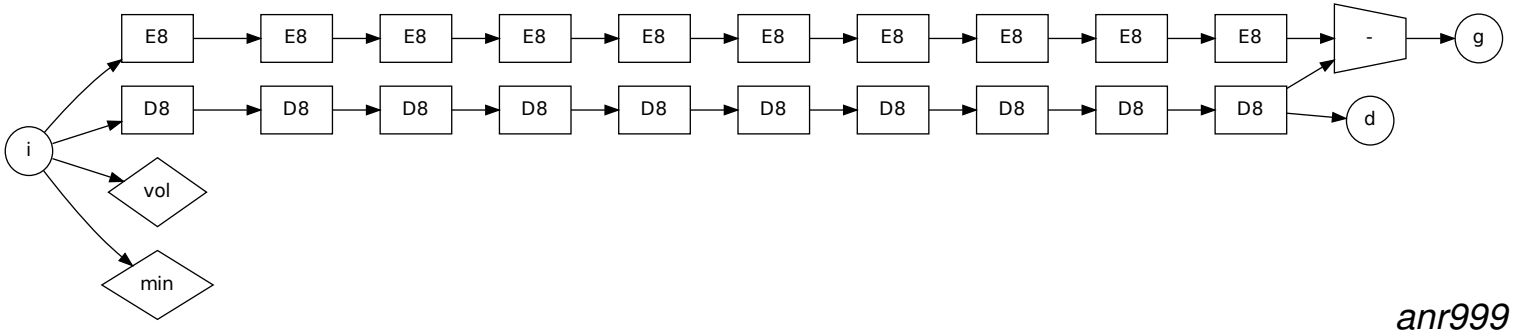
- arrows: image and scalar dependencies

- identify input and output images

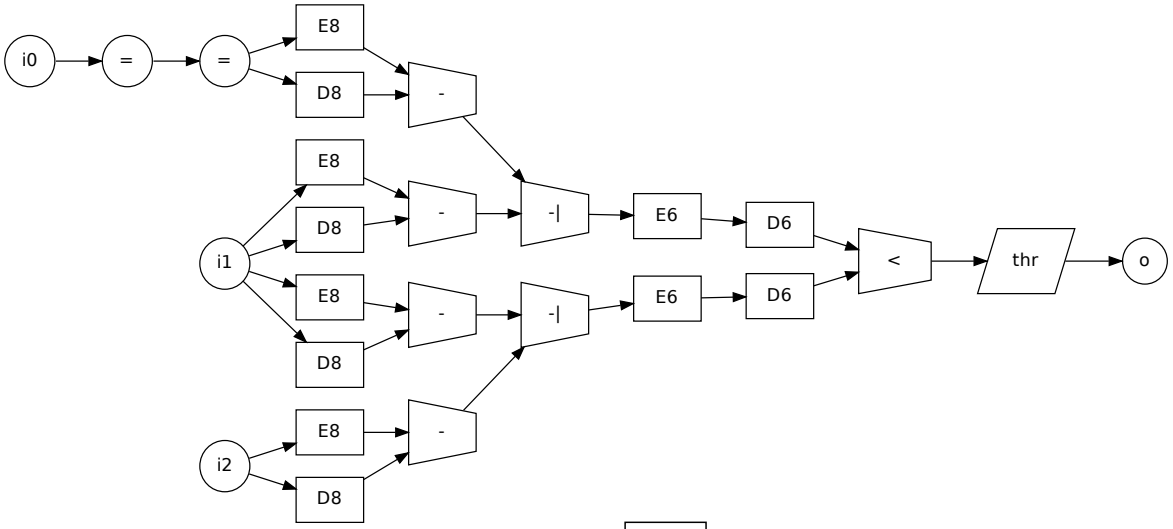
2.2 Common Sub-expression Elimination (CSE)

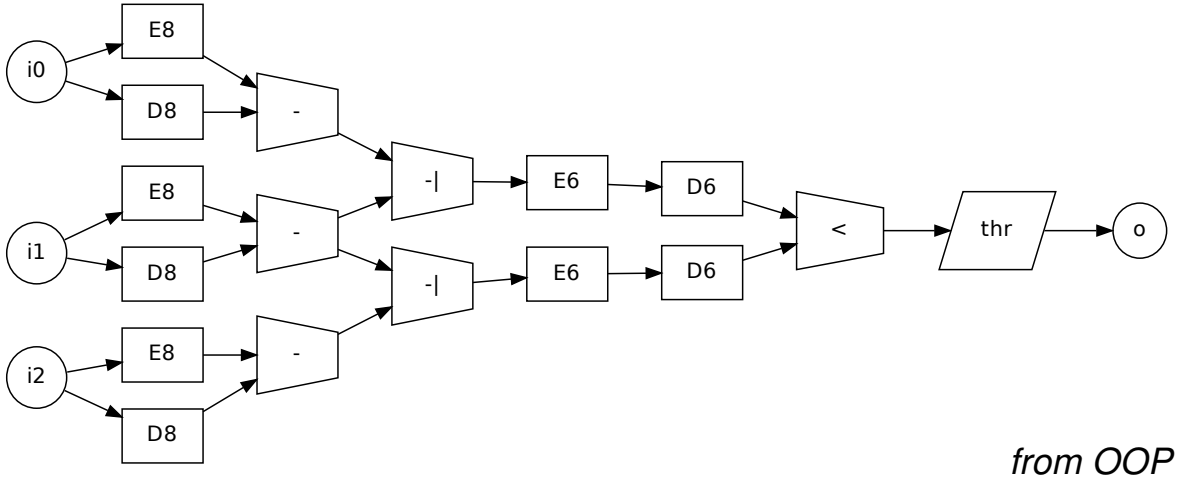
using operator commutativity if necessary





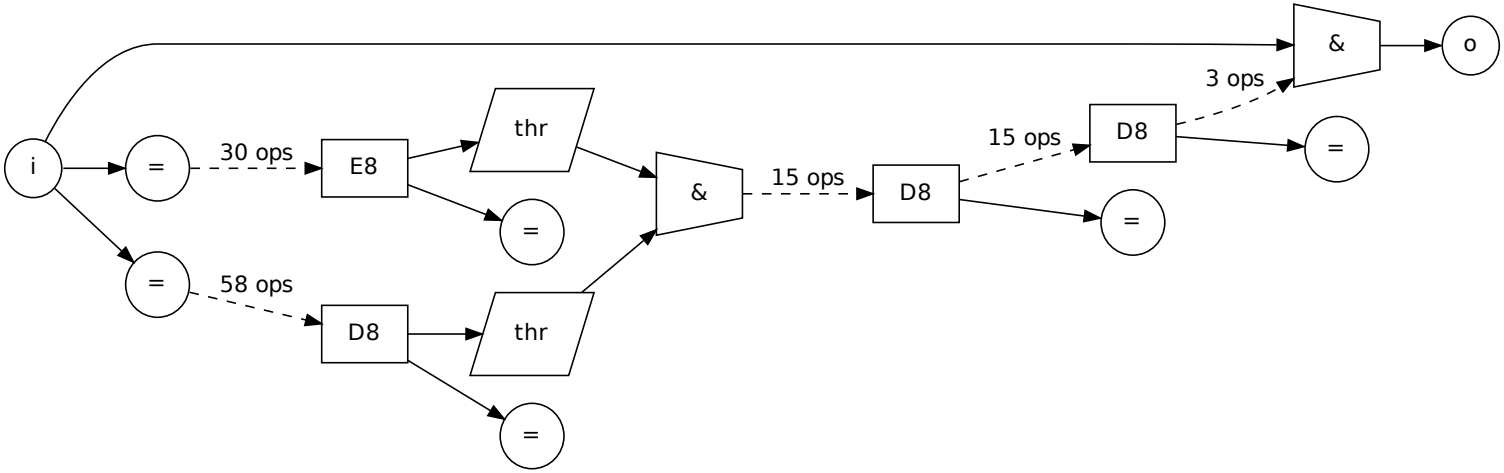
anr999

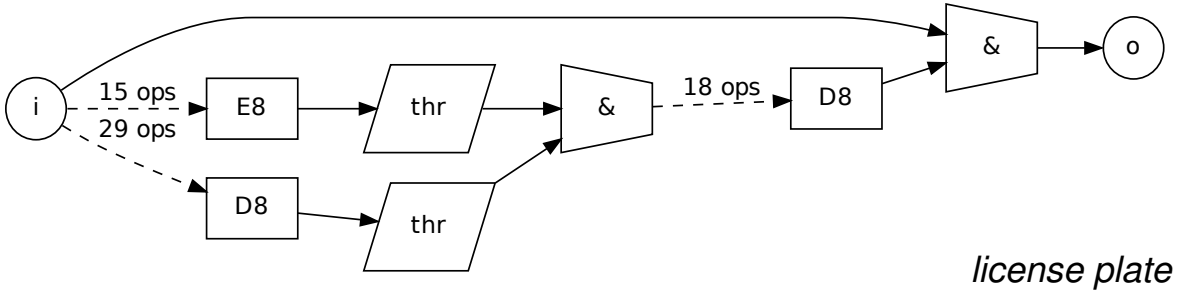

i0
=
=
E8
D8
-
i1
E8
D8
-
E8
D8
-
i2
E8
D8
-
-|
-|
E6
D6
E6
D6
<
thr
o


i0
E8
D8
-
i1
E8
D8
-
i2
E8
D8
-
-|
-|
E6
D6
E6
D6
<
thr
o

from OOP

2.3 Copy propagation (forward & backward)





license plate

3. Code Generation for SPoC – 3 stages

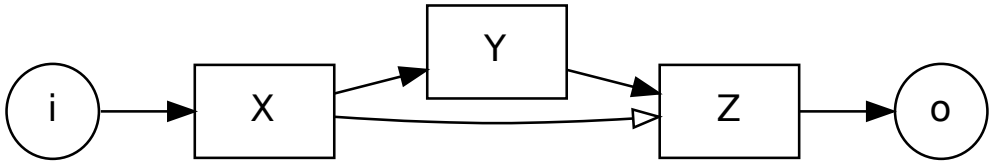
- constraints: 2 paths, img & scalar deps, operations, pipe depths
- mimimize number of calls to accelerator *NPC?*

1. split DAG
   - list scheduling with 2 live images constraint
   - sort according to image reuse, op order, number of ins/outs
2. instruction compaction, on conceptually infinite pipeline

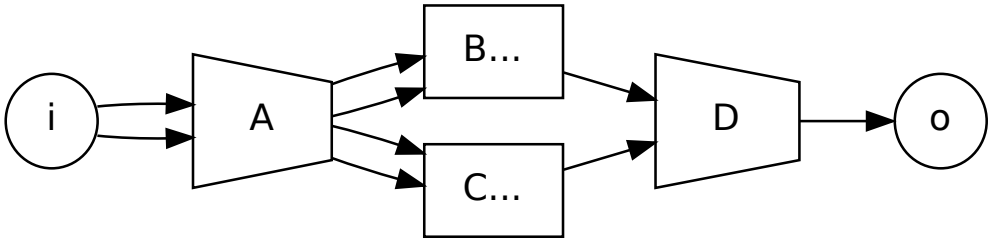
   weakly optimal (no reordering allowed)
3. cut pipeline, every pipeline depth

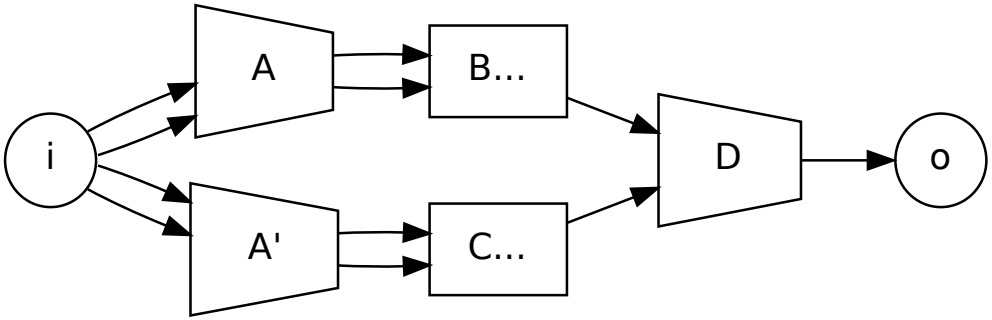
   locally optimal, if previous phase is optimal

Local optimality



Optimality if operation replication





Implementation in PIPS

`http://pips4u.org/`

- source to source, easier to debug output
- phase 1 – reuse (more or less) standard phases
- phase 2 – DAG building and optimization: 2 KLOCS
- phase 3 – SPoC code generation: 1.6 KLOCS

```
void helper_0(freia_data2d * o0, freia_data2d * o1,
  freia_data2d * i0, int32_t * red0, int32_t * red1,
  int32_t * kern2 /* ... up to kern16 */)
{
  // SKIPPED: declarations & initializations
  // - si & op: micro instructions
  // - sp & par: operation parameters
  // - redres & reduc: reduction results

  // set state of MUX stage 0 number 0
  si.mux[0][0].op = SPOC_MUX_IN0;
  // set state of POC stage 1 side 0
  si.poc[1][0].op = SPOC_POC_DILATE;
  si.poc[1][0].grid = SPOC_POC_8_CONNEX;
```

```
    // and its kernel
    for(i=0 ; i<9 ; i++)
      sp.poc[1][0].kernel[i] = kern2[i];
    // SKIPPED: more configurations...


    // actual call to the hardware accelerator
    // instructions, params, 2 images out, 2 images in
    freia_cg_process_2i_2o(op, par, o0, o1, i0, i0);
    // extract reductions results
    freia_cg_read_reduction_results(&redres);
    *red0 = (int32_t) reduc.measure[0][0].minimum;
    *red1 = (int32_t) reduc.measure[0][0].volume;
}
```

Experiments: 1213 cases

combinatorial 1005 (3 to 6 ops)

elementary 101 (1 to 13 ops)

atomic 40 (1 op) – generate accelerated version

functions 57 (5 to 135 ops)

Mostly optimal

Future Work

- more experiments
- Terapix SIMD target *in progress*
- OpenCL for GPGPU *in project*
- GP processor could simulate hardware for locality?