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
• $\mu$P + 128 SIMD PE array, 1024 pixels per PE, neighbor coms

• computation // communication (in or out)  

• 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? \(\text{VSIPL} 1000s \text{ functions}\)

(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, \( \mu \)-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: \textbf{generic}
   \begin{itemize}
   \item inlining, scalar const. prop., loop unroll., dead-code elimination
   \end{itemize}

2. Build and optimize DAGs of image operations: \textbf{generic}
   \begin{itemize}
   \item constant propagation, CSE, SDC, copy propagation
   \end{itemize}

3. Generate code for target: \textbf{specific}
   \begin{itemize}
   \item \textbf{SPoC}: DAG splitting and scheduling, compaction, cutting
   \item \textbf{Terapix}: DAG splitting, scheduling, memory allocation
   \item \textbf{OpenCL}: DAG splitting, simple operation aggregation
   \end{itemize}
2.1 Build Image Expression DAG

- expression DAG of simple image operations
  - morpho, ALU, threshold, measure, copies, scalar ops
- arrows: image and scalar dependencies

from Video Survey
2.2 Optimize DAG

freia\_gradient connexity=8 depth=10
freia\_erode connexity=8 depth=10
freia\_dilate connexity=8 depth=10
freia\_dilate connexity=8 depth=10

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

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

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)