Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators

Oliver Reiche, Moritz Schmid, Richard Membarth, Frank Hannig, and Jürgen Teich

Hardware/Software Co-Design, University of Erlangen-Nürnberg

CODES+ISSS, October 14, 2014, New Delhi
Motivation: e. g. Driver Assistance Systems

Mostly based on image feature detection:

(a) Edge detection  (b) Corner detection  (c) Optical flow

Where to compute features?


Write once, decide later!
Outline

HIPA^{cc} Framework

FPGA Targets

Results

Conclusion
HIPA$^{cc}$ Framework
HIPA$^{cc}$: The Heterogeneous Image Processing Acceleration Framework

- **Domain Knowledge**
- **Architecture Knowledge**

**Source-to-Source Compiler**

**C++ embedded DSL**

- **CUDA** (GPU)
- **OpenCL** (x86/GPU)
- **C/C++** (x86)
- **Renderscript** (x86/ARM/GPU)

**CUDA/OpenCL/Renderscript Runtime Library**
HIPA\textsuperscript{cc}: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

- **IterationSpace** defines ROI of the output image
- **Accessor** input ROI with filtering
- **BoundaryCondition** boundary handling modes
- **Mask** convolution mask

Output image

Crop of output image

Crop of output image with offset
HIPA\textsuperscript{cc}: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

- **IterationSpace**: defines ROI of the output image
- **Accessor**: input ROI with filtering
- **BoundaryCondition**: boundary handling mode
- **Mask**: convolution mask

![Images](image1.jpg)  
Image and boundary  
Image crop  
Image crop with offset  
Image offset
HIPA\textsuperscript{cc}: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

\textbf{IterationSpace} defines ROI of the output image

\textbf{Accessor} input ROI with filtering

\textbf{BoundaryCondition} boundary handling modes

\textbf{Mask} convolution mask
HIPA$^{cc}$: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

- **IterationSpace** defines ROI of the output image
- **Accessor** input ROI with filtering
- **BoundaryCondition** boundary handling modes
- **Mask** convolution mask

[Diagram with numerical values and labels]
Example: Laplacian Operator

```c++
// coefficients for Laplacian operator
const int coef[3][3] = {
    { 0, 1, 0 },
    { 1, -4, 1 },
    { 0, 1, 0 }
};

Mask<int> mask(coef);
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// reading from in with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, BOUNDARY_MIRROR);
Accessor<uchar4> acc(bound);

// output image
IterationSpace<uchar4> iter(out);

// define kernel
Laplacian filter(iter, acc, mask);

// execute kernel
filter.execute();
```
Example: Laplacian Operator Kernel

class Laplacian : public Kernel<uchar4> {
private:
    Accessor<uchar4> &input;
    Mask<int> &mask;

public:
    Laplacian(IterationSpace<uchar4> &iter,
              Accessor<uchar4> &input, Mask<int> &mask)
        : Kernel(iter), input(input), mask(mask) {
        addAccessor(&input);
    }

    void kernel() {
        int4 sum = convolve(mask, HipaccSUM, [&] () -> int4 {
            return mask() * convert_int4(input(mask));
        });
        sum = max(sum, 0);
        sum = min(sum, 255);
        output() = convert_uchar4(sum);
    }
};
FPGA Targets
Workflow for FPGA Targets

C++ Embedded DSL

Golden Reference

RTL Simulation

Vivado HLS

FPGA

Domain/Architecture Knowledge

HDL

C++
Generating the Streaming Pipeline

Trace host code and translate it to *internal representation*:

- model as combination of *processes* and *spaces*
- create unique stream objects for each *space*
- identify memory reuse
- insert *copy processes*
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Streaming Pipeline: Example

Transform sequential execution order...

Figure: HIPA^{cc}’s sequential execution for the Harris corner detector

...into streaming pipeline of Vivado kernels.

Figure: Representation of Vivado kernels
Mapping of Local Operators

For each local operator kernel, a separate line and window buffer is allocated.

Figure: Line buffer and group delay
Packing Vector Types

Pack multiple vector channels into wider stream type (e.g. `uint32_t`)
- only use single window and line buffer for all channels
- apply vector operations by operator overloading

Figure: RGBA vector channels packed into single stream

The result is basically a SIMD unit realized in an FPGA
Results
Experimental Setup

Algorithms

Three feature detection algorithms:

(a) Laplacian operator
(b) Harris corner detector
(c) Census transform (optical flow)

Evaluation Environment

Zynq 7045  Kintex FPGA
Image size  1024 × 1024 pixels
Xilinx OpenCV  a Vivado HLS-specific image processing library
Results: Performance OpenCV vs. HIPA\textsuperscript{cc}

![Bar chart comparing throughput of OpenCV and HIPAcc for different filters: HC, LP 5×5, LPD 3×3, LHV 3×3.](chart)

- Throughput in [MPixel/s]

- Bar chart showing performance comparisons across different filters.
Results: Resource Usage OpenCV vs. HIPA$^{cc}$
Results: Performance GPU vs. FPGA

All implementations are stemming from the exact same DSL code base.
Conclusion
Conclusion

Advantages of DSL-based Approach

Productivity
- compact algorithm description
- less error-prone

Performance
- efficient target-specific code generation

Portability
- flexible target choice
- performance-portability, not just functional portability

HIPA\textsuperscript{cc} DSL code serves as baseline implementation \Rightarrow Test bench

Moving to higher abstraction level than HLS allows to further postpone design decisions and therefore achieve higher quality.
Questions?

Thanks for listening.
Any questions?

HIPACC
Heterogeneous Image Processing Acceleration

http://github.com/hipacc/hipacc-vivado

**Title**  Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators

**Author**  Oliver Reiche
References I


Backup Slides
## Results: Power Consumption

Table: Comparison of throughput and energy consumption for the ARM Mali-T604, Xilinx Zynq 7045, and Nvidia Tesla K20.

<table>
<thead>
<tr>
<th></th>
<th>Mali-T604</th>
<th></th>
<th>Zynq 7045</th>
<th></th>
<th>Tesla K20</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LPHV 3x3</td>
<td>100.4</td>
<td>41.8</td>
<td>333.3</td>
<td>1423.1</td>
<td>10000.0</td>
<td>74.1</td>
</tr>
<tr>
<td>LPD 3x3</td>
<td>62.2</td>
<td>25.9</td>
<td>324.7</td>
<td>1387.6</td>
<td>6250.0</td>
<td>46.3</td>
</tr>
<tr>
<td>LP 5x5</td>
<td>22.0</td>
<td>9.2</td>
<td>209.2</td>
<td>846.9</td>
<td>2631.6</td>
<td>19.5</td>
</tr>
<tr>
<td>HC</td>
<td>11.9</td>
<td>4.9</td>
<td>228.3</td>
<td>458.5</td>
<td>1098.9</td>
<td>8.1</td>
</tr>
<tr>
<td>OF</td>
<td>0.4</td>
<td>0.2</td>
<td>192.5</td>
<td>409.6</td>
<td>452.5</td>
<td>3.4</td>
</tr>
</tbody>
</table>