Generating FPGA-based Image Processing Accelerators with Hipacc

Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, Frank Hannig

1Hardware/Software Co-Design, Friedrich-Alexander University Erlangen-Nürnberg
2German Research Center for Artificial Intelligence, Saarland University

ICCAD, November 13, 2017, Irvine
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

(a) Edge detection
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

(a) Edge detection
(b) Corner detection
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

(a) Edge detection
(b) Corner detection
(c) Optical flow
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

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

Where to compute features?

ECU [1]
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

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

Where to compute features?

- ECU [1]
- μC [2]
Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

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

Where to compute features?

Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

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

Where to compute features?

Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

(a) Edge detection

(b) Corner detection

(c) Optical flow

Where to compute features?

Introductory Example: Driver Assistance Systems

Mostly based on image feature detection:

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

Where to compute features?


Question: Where to run and who implements the algorithm?
Outline

Image Processing Domain and Hipacc

Generating Hardware Accelerators

Evaluation and Results

Conclusion
Image Processing Domain and Hipacc
Hipacc: The Heterogeneous Image Processing Acceleration Framework

- **C++ embedded DSL**
- **Source-to-Source Compiler Clang/LLVM**
- **CUDA** (GPU)
- **OpenCL** (x86/GPU)
- **C/C++** (x86)
- **Renderscript** (x86/ARM/GPU)
- **OpenCL** (Intel/Altera FPGA)
- **Vivado C++** (Xilinx FPGA)
- **CUDA/OpenCL/Renderscript Runtime Library**
- **AOCL**
- **Vivado HLS**

**Domain Knowledge**
**Architecture Knowledge**
Image Processing Applications

We can define three characteristic data operations in image processing applications:

**Point Operators:**
Output data is determined by single input data

**Local Operators:**
Output data is determined by a local region of the input data  
(stencil pattern-based calculations)

**Global Operators:**
Output data is determined by all of the input data
Domain-Specific Extensions

- **IterationSpace** defines ROI of the output image
- **Accessor** input ROI with filtering
- **BoundaryCondition** border handling modes
- **Mask** convolution mask
Domain-Specific Extensions

**IterationSpace** defines ROI of the output image

**Accessor** input ROI with filtering

**BoundaryCondition** border handling modes

**Mask** convolution mask

(a) Output image

(b) Crop of output image

(c) Crop of output image with offset
Domain-Specific Extensions

**IterationSpace** defines ROI of the output image

**Accessor** input ROI with filtering

**BoundaryCondition** border handling modes

**Mask** convolution mask

(a) Image and border
(b) Image crop
(c) Image crop with offset
(d) Image offset
Domain-Specific Extensions

**IterationSpace** defines ROI of the output image

**Accessor** input ROI with filtering

**BoundaryCondition** border handling modes

**Mask** convolution mask

(a) Constant

(b) Clamp

(c) Repeat

(d) Mirror
Domain-Specific Extensions

**IterationSpace** defines ROI of the output image

**Accessor** input ROI with filtering

**BoundaryCondition** border handling modes

**Mask** convolution mask
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> bound(in, mask, Boundary::MIRROR);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
```
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
```
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
```
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
```
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
```
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
```
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
Example: Laplacian Operator

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

// read input
uchar4 *image_bits = readImage();

// input and output images
Image<uchar4> in(width, height);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

// mask (stencil) of local operator
Mask<int> mask(coef);

// input region with mirroring as boundary condition
BoundaryCondition<uchar4> bound(in, mask, Boundary::MIRROR);
Accessor<uchar4> acc(bound);

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

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

// execute kernel
filter.execute();

// read output
uchar4* out = out.data();
Example: Laplacian Operator Kernel

```cpp
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, [&](int) -> int4 {
                return mask() * convert_int4(input(mask));
            });
            sum = max(sum, 0);
            sum = min(sum, 255);
            output() = convert_uchar4(sum);
        }
};
```

Listing 1: Hipacc kernel code
Example: Laplacian Operator Kernel

```cpp
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);
        }
};
```

Listing 1: Hipacc kernel code
Extension for HLS: Bit-Width Annotations

More than primitive data types, e.g., char/short/int, are supported

```cpp
#pragma hipacc bw(sum,12)
uint sum = 0;
#pragma hipacc bw(x,2)
uint x = 0;
#pragma hipacc bw(y,2)
uint y = 0;
for (y = 0; y < size; ++y) {
    for (x = 0; x < size; ++x) {
        sum += mask[y][x] * Input(x-1,y-1);
    }
}
sum /= 16;
output() = sum;
```

Listing 2: Example bit-width annotation with Hipacc
Generating Hardware Accelerators
Hipacc Workflow for Generating HLS Code
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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*
Generating Streaming Pipeline

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

- model as a 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*
Dependency Graph Restructuring: Harris Corner Example

Transform sequential execution order...

Figure: Hipacc’s sequential execution for the Harris corner detector
Dependency Graph Restructuring: Harris Corner Example

Transform sequential execution order...

...into a streaming pipeline of FPGA kernels.

Figure: Hipacc’s sequential execution for the Harris corner detector

Figure: Representation of AOC kernels
Common AST Transformations: Memory Architecture Design

Achieves throughput of one pixel per cycle!
How to Increase Throughput Even Further?

A naive way would be replicating the accelerator hardware:
Common AST Transformations: Loop Coarsening

Is there a more resource-efficient approach?
Common AST Transformations: Loop Coarsening

Coarse-grained parallelization of point operators is rather straightforward:

\[
\begin{align*}
\text{input} & \quad f & \quad \text{output} \\
\text{input} & \quad \{f, f, f, f\} & \quad \text{output}
\end{align*}
\]
Common AST Transformations: Loop Coarsening

Coarsening of local operators requires a more complicated approach

- Accelerator receives packed data elements (superpixels)
- Stores these in line buffer and superwindow
- Packed data is disassembled into subwindows
- Subwindows processed by unmodified operator implementation

**Pros:** Less overhead and true linear speedup

**Cons:** Only if accelerator smaller than 50%
Vendor-Specific Transformation: Bit-Width Reduction

In case of *Altera OpenCL*:

```c
#pragma hipacc bw(sum,12)

uint sum = 0;
#pragma hipacc bw(x,2)
uint x = 0;
#pragma hipacc bw(y,2)
uint y = 0;
for (y = 0; y < size; ++y) {
    for (x = 0; x < size; ++x) {
        sum += mask[y][x] * Input(x-1,y-1);
    }
}
sum /= 16;
output() = sum;
```

**Listing 3:** Gaussian blur in Hipacc with annotated bit widths

```c
uint sum = 0;
uint x = 0;
uint y = 0;
for (y = ((0) & 3); ((y) & 3) < size; y = (((y) & 3) + 1) & 3)){
    for (x = ((0) & 3); ((x) & 3) < size; x = (((x) & 3) + 1) & 3)){
        sum = (((sum) & 4095) + mask[y][x] * getTokenAt(Input, 1 + ((x) & 3) - 1, 1 + ((y) & 3) -1)) & 4095;
    }
}
sum = (((sum) & 4095) / 16) & 4095;
return sum;
```

**Listing 4:** Generated OpenCL code with reduced bit widths
Evaluation and Results
Results

1 Local Operator & 1 Invocation

Hipacc: $8 + 24 \text{ LoC}$
Tesla K20: $4103.0 \text{ FPS}$
Tegra K1: $72.1 \text{ FPS}$
Kintex 7: $4514.3 \text{ FPS} \quad (v = 16)$
Stratix V: $4162.1 \text{ FPS} \quad (v = 16)$

1 Local + 1 Point Operator & 3 Invocations

Hipacc: $11 + 10 \text{ LoC}$
Tesla K20: $3356.0 \text{ FPS}$
Tegra K1: $25.3 \text{ FPS}$
Kintex 7: $676.3 \text{ FPS} \quad (v = 4)$
Stratix V: $4301.7 \text{ FPS} \quad (v = 16)$

2 Local + 3 Point Operators & 9 Invocations

Hipacc: $23 + 53 \text{ LoC}$
Tesla K20: $921.7 \text{ FPS}$
Tegra K1: $33.5 \text{ FPS}$
Kintex 7: $1620.2 \text{ FPS} \quad (v = 8)$
Stratix V: $1057.8 \text{ FPS} \quad (v = 4)$

2 Local + 1 Point Operator & 5 Invocations

Hipacc: $31 + 52 \text{ LoC}$
Tesla K20: $452.5 \text{ FPS}$
Tegra K1: $18.4 \text{ FPS}$
Kintex 7: $483.6 \text{ FPS} \quad (v = 4)$
Stratix V: $249.6 \text{ FPS} \quad (v = 4)$

Figure: Comparison of generated image algorithms for GPUs and FPGAs ($v$ is coarsening factor)

Table: Resource utilization of synthesized algorithms.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Kintex 7</th>
<th>Stratix V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SLICE (%)</td>
<td>BRAM (%)</td>
</tr>
<tr>
<td>Bilateral</td>
<td>44.68</td>
<td>0.45</td>
</tr>
<tr>
<td>Sobel</td>
<td>28.79</td>
<td>3.60</td>
</tr>
<tr>
<td>Harris Corner</td>
<td>55.75</td>
<td>3.60</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>60.09</td>
<td>11.69</td>
</tr>
</tbody>
</table>
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

Hipacc DSL code serves as baseline implementation ⇒ Test bench
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

Hipacc DSL code serves as baseline implementation ⇒ Test bench

Develop the algorithm first, decide the target architecture afterward!
Questions?

Thanks for listening.

Any questions?

Generating FPGA-based Image Processing Accelerators with Hipacc

http://github.com/hipacc/hipacc-fpga
References


