FPGA-Based Accelerator Design from a Domain-Specific Language

M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Friedrich-Alexander University Erlangen-Nürnberg
FPL, August 29, 2016, Lausanne, Switzerland
Motivation

**FPGAs**  High throughput, great energy efficiency

**OpenCL**  Portable and open programming model for heterogeneous systems, including CPUs, GPUs, DSPs etc.

Motivation: High Level Synthesis using Altera OpenCL

+ Plug and accelerate: Automatic interface design
+ Directives for exploiting data-level parallelism
- Efficiency comes with hardware design patterns
! Develop Altera OpenCL (AOC) software in Hardware Manner (HwM)

Figure: Design points for a $5 \times 5$ Gaussian blur filter.
Motivation: High Level Synthesis using Altera OpenCL

+ Plug and accelerate: Automatic interface design
+ Directives for exploiting data-level parallelism
  - Efficiency comes with hardware design patterns
  - Develop Altera OpenCL (AOC) software in Hardware Manner (HwM)

Figure: Design points for a $5 \times 5$ Gaussian blur filter.
Motivation: High Level Synthesis using Altera OpenCL

+ Plug and accelerate: Automatic interface design
+ Directives for exploiting data-level parallelism
- Efficiency comes with hardware design patterns

Develop Altera OpenCL (AOC) software in Hardware Manner (HwM)

Figure: Design points for a $5 \times 5$ Gaussian blur filter.
Motivation: High Level Synthesis using Altera OpenCL

+ Plug and accelerate: Automatic interface design
+ Directives for exploiting data-level parallelism
- Efficiency comes with hardware design patterns
- Develop Altera OpenCL (AOC) software in Hardware Manner (HwM)

![Graph showing design points for a 5x5 Gaussian blur filter.](image)

**Figure:** Design points for a 5×5 Gaussian blur filter.
Motivation: High Level Synthesis using Altera OpenCL

+ Plug and accelerate: Automatic interface design
+ Directives for exploiting data-level parallelism
- Efficiency comes with hardware design patterns
! Develop Altera OpenCL (AOC) software in Hardware Manner (HwM)

Figure: Design points for a 5×5 Gaussian blur filter.
Outline

Programming Methodology
  Developing Software in Hardware Manner

Efficient Code Generation through a Domain-Specific Language

Extensions to HIPA\textsuperscript{cc} Framework
  Automatic Bit-width Reduction

Evaluation and Results

Conclusion
Programming Methodology
Efficient OpenCL program for AOC

- Use hardware design patterns
  - Single-item line-buffered kernels for image processing
Efficient OpenCL program for AOC

- Use hardware design patterns
- Apply known hardware design optimizations
  - Strength Reduction
  - Bit-width reduction
  - Use 1-bit control signals instead of repeated conditions

**Listing 1: Source Code**

```c
    c = a * 16;
```

**Listing 2: Strength Reduction**

```c
    c = a << 4;
```
Efficient OpenCL program for AOC

- Use hardware design patterns
- Apply known hardware design optimizations
  - Strength Reduction
  - Bit-width reduction
  - Use 1-bit control signals instead of repeated conditions
Efficient OpenCL program for AOC

- Use hardware design patterns
- Apply known hardware design optimizations
  - Strength Reduction
  - Bit-width reduction
  - Use 1-bit control signals instead of repeated conditions
Efficient OpenCL program for AOC

- Fetch best practices
  - Remove the else branch by moving its content to preceding area
  - Nested loops should always have constant bounds
  - Use temporary registers wherever it is possible
  - Limit life time of variables using scope operators
  - Avoid using 2-dimensional arrays
  - Writing to arrays should be sequential and simple as possible

Listing 1: Source Code

```c
if(condition1){
  a = 3;
} else{
  a = 5;
}
```

Listing 2: Remove else condition

```c
a = 5;
if(condition1){
  a = 3;
}
```
Efficient OpenCL program for AOC

- Fetch best practices
  - Remove the else branch by moving its content to preceding area
  - Nested loops should always have constant bounds
  - Use temporary registers wherever it is possible
  - Limit lifetime of variables using scope operators
  - Avoid using 2-dimensional arrays
  - Writing to arrays should be sequential and simple as possible

Listing 1: Source Code
```c
for (int i = 0; i < 5; ++i){
    for (int j = 0; j < i; ++j){
        /* operations */
    }
}
```

Listing 2: Constant loop bounds
```c
for (int i = 0; i < 5; ++i){
    for (int j = 0; j < 5; ++j){
        if (j < i) {
            /* operations */
        }
    }
}
```
Efficient OpenCL program for AOC

- Fetch best practices
  - Remove the else branch by moving its content to preceding area
  - Nested loops should always have constant bounds
  - Use temporary registers wherever it is possible
  - Limit life time of variables using scope operators
  - Avoid using 2-dimensional arrays
  - Writing to arrays should be sequential and simple as possible

Listing 1: Source Code

```c
a = array[i];
if (condition){
  b = array[i];
}
c = array[i]*5;
```

Listing 2: Source Code

```c
char temp = array[i];
a = temp;
if (condition){
  b = temp;
}
c = temp*5;
```
Efficient OpenCL program for AOC

- Fetch best practices
  - Remove the else branch by moving its content to preceding area
  - Nested loops should always have constant bounds
  - Use temporary registers wherever it is possible
  - Limit life time of variables using scope operators
  - Avoid using 2-dimensional arrays
  - Writing to arrays should be sequential and simple as possible

Listing 1: Source Code

```c
/* code portion1 */
char temp;
/* code_portion2(temp) */
/* code_portion3 */
```

Listing 2: Scope Operators

```c
/* code portion1 */
{
    char temp;
    /* code_portion2(temp) */
}
/* code_portion3 */
```
Efficient OpenCL program for AOC

- Fetch best practices
  - Remove the else branch by moving its content to preceding area
  - Nested loops should always have constant bounds
  - Use temporary registers wherever it is possible
  - Limit life time of variables using scope operators
  - Avoid using 2-dimensional arrays
  - Writing to arrays should be sequential and simple as possible

Listing 1: Source Code

```c
char array[5][5];
```

Listing 2: Avoid 2 dimensional arrays

```c
char arrayRow1[5];
char arrayRow2[5];
char arrayRow3[5];
char arrayRow4[5];
char arrayRow5[5];
```
Efficient OpenCL program for AOC

• Fetch best practices
  • Remove the else branch by moving its content to preceding area
  • Nested loops should always have constant bounds
  • Use temporary registers wherever it is possible
  • Limit life time of variables using scope operators
  • Avoid using 2-dimensional arrays
  • Writing to arrays should be sequential and simple as possible

Listing 1: Source Code

```c
char array[7];
// Load new data
for(int i=0; i<2; i++){
    char newdata = newdata[i];
    array[i+3] = newdata;
    array[i+5] = newdata;
}
// Shift only a portion
if(condition){
    array[5] = array[6];
}
```

Listing 2: Simplify array access

```c
char array1[5];
char array2[2];
// Load new data
for(int i=0; i<2; i++){
    char newdata = newdata[i];
    array1[i+3] = newdata;
    array2[i] = newdata;
}
// Shift only a portion
if(condition){
    array2[0] = array2[1];
}
```
Proposed Approach: Develop Software in Hardware Manner

- Write Altera OpenCL code following these two principles:
  1. Calculate all the conditional cases first, decide at the end
  2. Exploit temporal locality
Proposed Approach: Develop Software in Hardware Manner

- Write Altera OpenCL code following these two principles:
  1. Calculate all the conditional cases first, decide at the end
  2. Exploit temporal locality

Listing 3: Software Manner

```
if (condition){
    calculate_a(...);
    result = a;
} else{
    calculate_b(...);
    result = b;
}
```

Listing 4: Hardware Manner

```
calculate_a(...) ;
calculate_b(...) ;
result = b;
if (condition){
    result = a;
}
```
Proposed Approach: Develop Software in Hardware Manner

- Write Altera OpenCL code following these two principles:
  1. Calculate all the conditional cases first, decide at the end
  2. Exploit temporal locality

Listing 3: Software Manner

```c
for(int i=0; i<SIZE; i++){
    x = in[i-2];
    y = in[i-1];
    z = in[i];
    out = compute(x, y, z);
}
```

Listing 4: Hardware Manner

```c
for(int i=0; i<SIZE; i++){
    x = y;
    y = z;
    z = in[i];
    out = compute(x, y, z);
}
```
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

Figure: Clamp boundary condition on an image
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>D</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Figure: Clamp boundary condition on an image
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

[Diagram of hardware design with MUXes and reg 00 to reg 44]
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

Requires hardware design expertise!
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

Requires hardware design expertise!

Best practices on syntactic description matters!

5x5 LOCAL OPERATOR

[0:2] ROWs

[3:4] ROWs SELECTION

Pixel Stream

Line Buffer

Line Buffer

Line Buffer

M. Akif Özkan | Hardware/Software Co-Design | FPGA-Based Accelerator Design from a Domain-Specific Language
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

Requires hardware design expertise!

Best practices on syntactic description matters!

No automatic kernel and accelerator replication!

[Diagram of hardware design with labels: reg 00, reg 10, reg 30, reg 20, reg 01, reg 40, reg 11, reg 41, reg 21, reg 02, reg 42, reg 12, reg 43, reg 22, reg 03, reg 44, reg 13, reg 44', reg 23, reg 04, reg 44', reg 24, MUX, MUX, MUX, MUX, Line Buffer, Pixel Stream, [3:4] ROWs SELECTION, 5x5 LOCAL OPERATOR, [0:2] ROWs]
Requires hardware design expertise!

Best practices on syntactic description matters!

No automatic kernel and accelerator replication!

Code Size increases upto 10x!
Efficient Code Generation through a Domain-Specific Language
HIPA$^{cc}$: The Heterogeneous Image Processing Acceleration Framework
Operators in the Image Processing Domain

We can define three characteristic data operations in the domain:

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

**Local Operators:**
Output data is determined by local region of the input data

**Global Operators:**
Output data is determined by all of the input data
Streaming Pipeline: Harris Corner Detection Example

Transform sequential execution order…

Figure: HIPA\textsuperscript{cc}'s sequential execution for the Harris corner detector
Streaming Pipeline: Harris Corner Detection Example

Transform sequential execution order . . .

Figure: HIPAC's sequential execution for the Harris corner detector

. . . into streaming pipeline of FPGA kernels.

Figure: Representation of AOC kernels
Example: Laplacian Operator

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

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

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write 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();
Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

// Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

// Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

// Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write 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();

Image<uchar4> in(width, height, inputImage);
Image<uchar4> out(width, height);

// load image data
in = image_bits;

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

// 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();

//Write output
uchar4* out = out.data();
```
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);
    }
};
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) {
    add_accessor(&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);
  }
};
Extensions to HIPA$^{	ext{cc}}$ Framework
Arbitrary Bit Width Hardware Design

- The OpenCL standard contains primitive data types only, e.g., char/short/int
Arbitrary Bit Width Hardware Design

- The OpenCL standard contains primitive data types only, e.g., char/short/int

+ AOC supports arbitrary bit width declaration

- AOC arbitrary bit width declaration is masking

\[ x = (RHS) \& 0x7; \]
Arbitrary Bit Width Hardware Design

- The OpenCL standard contains primitive data types only, e.g., char/short/int

+ AOC supports arbitrary bit width declaration

- AOC arbitrary bit width declaration is masking
  \[ x = (RHS) \& 0x7 \; \]

- Compound assignment operations need to be expanded
  \[ // x += (RHS) \& 0x7; \\
  x = ((x \& 0x7) + RHS) \& 0x7; \]

- Unary operations need to be expanded
  \[ // x++; \\
  x = ((x \& 0x7) + 1) \& 0x7; \]
Arbitrary Bit Width Hardware Design

- The OpenCL standard contains primitive data types only, e.g., char/short/int
  
+ AOC supports arbitrary bit width declaration

- AOC arbitrary bit width declaration is masking
  
\[
x = (RHS) & 0x7;
\]

- Compound assignment operations need to be expanded

  
// x += (RHS) & 0x7;

\[
x = ((x & 0x7) + RHS) & 0x7;
\]

- Unary operations need to be expanded

  
// x++;

\[
x = ((x & 0x7) + 1) & 0x7;
\]

Utilizing exact bit widths is tedious in Altera OpenCL 14.1!
Automatic Bit Width Reduction: An extension to HIPA\textsuperscript{cc}

Listing 5: Gaussian blur in HIPA\textsuperscript{cc} with annotated bit widths

```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 6: Gaussian blur in HIPA\textsuperscript{cc} with annotated bit widths

```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] * getwindowat(Input, 1 + x, 1 + y);
    }
}
sum /= 16;
return sum;
```
Automatic Bit Width Reduction: An extension to HIPA$^{cc}$

Listing 5: Gaussian blur in HIPA$^{cc}$ with annotated bit widths

```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 6: Gaussian blur in HIPA$^{cc}$ 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]
               * getWindowAt(Input, 1 + ((x) & 3) - 1, 1 + ((y) & 3) -1)) & 4095);
    }
}
sum = (((sum) & 4095) / 16) & 4095;
return sum;
```
Evaluation and Results
**HIPA\textsuperscript{cc} vs Handwritten Examples provided by Altera**

![Graph showing hardware utilization and clock frequency comparison between Edge-Detector, Lukas Kanade, Altera, and HIPA\textsuperscript{cc} for various designs.]

**Figure:** HIPA\textsuperscript{cc} vs Altera handwritten examples for 1024 × 1024 image size.
FPGA vs GPU

Figure: Comparison of throughput for the Nvidia Tegra K1, Nvidia Tesla K20, and Altera Stratix V.
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
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$^{cc}$ DSL code serves as baseline implementation ⇒ Test bench

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

Thanks for listening.
Any questions?

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

**Title**  FPGA-Based Accelerator Design from a Domain-Specific Language

**Speaker**  M. Akif Özkan, akif.oezkan@fau.de
Backup Slides
Additional Results
Boundary Handling: SW Manner vs HW Manner

(a) Single-Item Line-Buffered Kernel developed in Software Manner

(b) Single-Item Line-Buffered Kernel developed in Hardware Manner

Figure: Boundary conditions for a $5 \times 5$ Gaussian blur on a $1024 \times 1024$ image.
Automatic Bit Width Reduction: Results

- AOC is very successful when inner loop trip counts are known during compile time and the unroll directive has been specified for synthesis.
- Great improvement in clock frequency for dynamic loops.

Table: Automatic bit width reduction on local operators of size $3 \times 3$ with image size $1024 \times 1024$.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Type</th>
<th>II</th>
<th>ALUTs</th>
<th>Registers</th>
<th>Logic (%)</th>
<th>M10K</th>
<th>DSP</th>
<th>Freq [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel</td>
<td>normal</td>
<td>2</td>
<td>8552</td>
<td>12433</td>
<td>19.76</td>
<td>84</td>
<td>4</td>
<td>131.25</td>
</tr>
<tr>
<td></td>
<td>reduced</td>
<td>2</td>
<td>8230</td>
<td>12098</td>
<td>19.11</td>
<td>84</td>
<td>3</td>
<td>152.09</td>
</tr>
<tr>
<td>Gaussian</td>
<td>normal</td>
<td>2</td>
<td>8593</td>
<td>12423</td>
<td>19.86</td>
<td>84</td>
<td>4</td>
<td>132.50</td>
</tr>
<tr>
<td></td>
<td>reduced</td>
<td>2</td>
<td>8439</td>
<td>11843</td>
<td>19.26</td>
<td>83</td>
<td>3</td>
<td>153.11</td>
</tr>
</tbody>
</table>
Kernel Vectorization
Kernel Vectorization

Vectorization by loop coarsening [1]: unroll the outer loop by factor $v_i$

- replicate only the kernel: sublinear increase of resource usage
- better exploitation of bandwidth: nearly linear speedup

Kernel Vectorization

Figure: Vectorization of a 3 × 3 bilateral filter with clamping on an image of size 1024 × 1024.
Generating the Streaming Pipeline
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
  - build dependency graph
  - traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Generating the Streaming Pipeline

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

- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*
Altera OpenCL System Level Interfaces
Channels  Data sharing between kernels

Altera OpenCL System Level Interfaces

Channels  Data sharing between kernels

Table: Impact of channels on line-buffered single-work item implementations.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Type</th>
<th>II</th>
<th>ALUTs</th>
<th>Registers</th>
<th>LU (%)</th>
<th>BRAM</th>
<th>DSP</th>
<th>Freq [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Kernel</td>
<td>single</td>
<td>1</td>
<td>6821</td>
<td>7812</td>
<td>14.04</td>
<td>43</td>
<td>0</td>
<td>152.89</td>
</tr>
<tr>
<td>Mean Filter</td>
<td>single</td>
<td>1</td>
<td>8248</td>
<td>9415</td>
<td>16.78</td>
<td>51</td>
<td>0</td>
<td>151.52</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>single</td>
<td>1</td>
<td>8244</td>
<td>9694</td>
<td>16.84</td>
<td>66</td>
<td>2</td>
<td>132.56</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>channel</td>
<td>1</td>
<td>11281</td>
<td>13653</td>
<td>23.75</td>
<td>64</td>
<td>3</td>
<td>148.75</td>
</tr>
</tbody>
</table>

- Channels use considerable amount of logic!
### Channels

Data sharing between kernels

**Table:** Impact of channels on line-buffered single-work item implementations.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Type</th>
<th>II</th>
<th>ALUTs</th>
<th>Registers</th>
<th>LU (%)</th>
<th>BRAM</th>
<th>DSP</th>
<th>Freq [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Kernel</td>
<td>single</td>
<td>1</td>
<td>6821</td>
<td>7812</td>
<td>14.04</td>
<td>43</td>
<td>0</td>
<td>152.89</td>
</tr>
<tr>
<td>Mean Filter</td>
<td>single</td>
<td>1</td>
<td>8248</td>
<td>9415</td>
<td>16.78</td>
<td>51</td>
<td>0</td>
<td>151.52</td>
</tr>
<tr>
<td>2 Mean Filter</td>
<td>separate</td>
<td>1</td>
<td>12104</td>
<td>15132</td>
<td>25.36</td>
<td>100</td>
<td>0</td>
<td>145.78</td>
</tr>
<tr>
<td>2 Mean Filter</td>
<td>channel</td>
<td>1</td>
<td>10439</td>
<td>12466</td>
<td>21.47</td>
<td>60</td>
<td>0</td>
<td>154.14</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>single</td>
<td>1</td>
<td>8244</td>
<td>9694</td>
<td>16.84</td>
<td>66</td>
<td>2</td>
<td>132.56</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>channel</td>
<td>1</td>
<td>11281</td>
<td>13653</td>
<td>23.75</td>
<td>64</td>
<td>3</td>
<td>148.75</td>
</tr>
</tbody>
</table>

- Channels use considerable amount of logic!

+ Channels are cheaper than kernel IOs.
Altera OpenCL System Level Interfaces

Channels  Data sharing between kernels

Table: Impact of channels on line-buffered single-work item implementations.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Type</th>
<th>II</th>
<th>ALUTs</th>
<th>Registers</th>
<th>LU (%)</th>
<th>BRAM</th>
<th>DSP</th>
<th>Freq [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Kernel</td>
<td>single</td>
<td>1</td>
<td>6821</td>
<td>7812</td>
<td>14.04</td>
<td>43</td>
<td>0</td>
<td>152.89</td>
</tr>
<tr>
<td>Mean Filter</td>
<td>single</td>
<td>1</td>
<td>8248</td>
<td>9415</td>
<td>16.78</td>
<td>51</td>
<td>0</td>
<td>151.52</td>
</tr>
<tr>
<td>2 Mean Filter</td>
<td>separate</td>
<td>1</td>
<td>12104</td>
<td>15132</td>
<td>25.36</td>
<td>100</td>
<td>0</td>
<td>145.78</td>
</tr>
<tr>
<td>2 Mean Filter</td>
<td>channel</td>
<td>1</td>
<td>10439</td>
<td>12466</td>
<td>21.47</td>
<td>60</td>
<td>0</td>
<td>154.14</td>
</tr>
<tr>
<td>2 Mean Filter</td>
<td>fused</td>
<td>1</td>
<td>9480</td>
<td>11715</td>
<td>19.47</td>
<td>64</td>
<td>0</td>
<td>122.47</td>
</tr>
<tr>
<td>3 Mean Filter</td>
<td>channel</td>
<td>1</td>
<td>12829</td>
<td>15451</td>
<td>25.36</td>
<td>67</td>
<td>0</td>
<td>145.57</td>
</tr>
<tr>
<td>3 Mean Filter</td>
<td>fused</td>
<td>1</td>
<td>10899</td>
<td>13651</td>
<td>22.24</td>
<td>79</td>
<td>0</td>
<td>125.00</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>single</td>
<td>1</td>
<td>8244</td>
<td>9694</td>
<td>16.84</td>
<td>66</td>
<td>2</td>
<td>132.56</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>channel</td>
<td>1</td>
<td>11281</td>
<td>13653</td>
<td>23.75</td>
<td>64</td>
<td>3</td>
<td>148.75</td>
</tr>
<tr>
<td>Luma+Mean</td>
<td>fused</td>
<td>1</td>
<td>9635</td>
<td>11846</td>
<td>20.09</td>
<td>72</td>
<td>2</td>
<td>123.53</td>
</tr>
<tr>
<td>Harris corner</td>
<td>channel</td>
<td>1</td>
<td>43127</td>
<td>54334</td>
<td>89.87</td>
<td>186</td>
<td>9</td>
<td>142.52</td>
</tr>
<tr>
<td>Harris corner</td>
<td>fused</td>
<td>1</td>
<td>33263</td>
<td>43319</td>
<td>70.87</td>
<td>177</td>
<td>9</td>
<td>107.35</td>
</tr>
</tbody>
</table>

- Channels use considerable amount of logic!
+ Channels are cheaper than kernel IOs.
- Smaller the size of kernel body, faster the clock frequency.