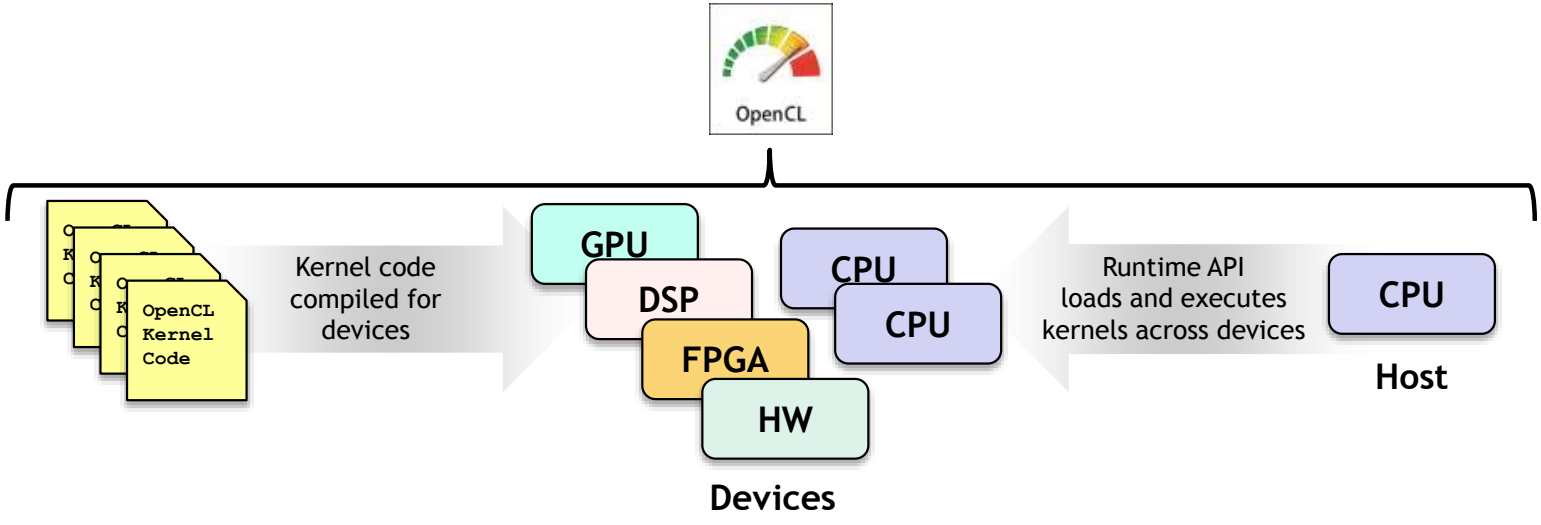
FPGA-Based Accelerator Design from a Domain-Specific Language

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.



https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf

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)

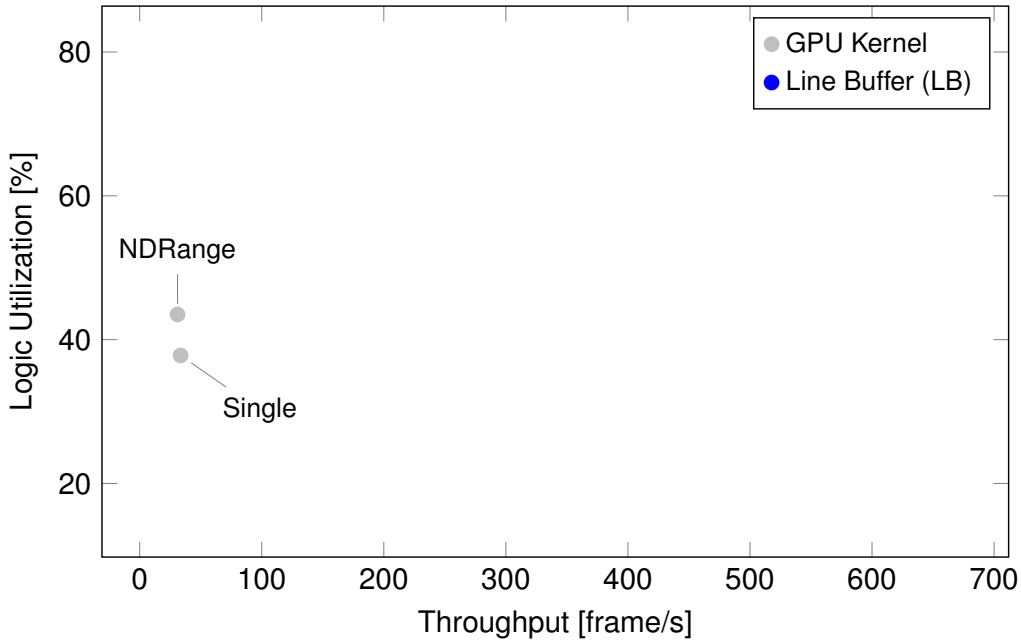


Figure: Design points for a 5×5 Gaussian blur filter.

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)

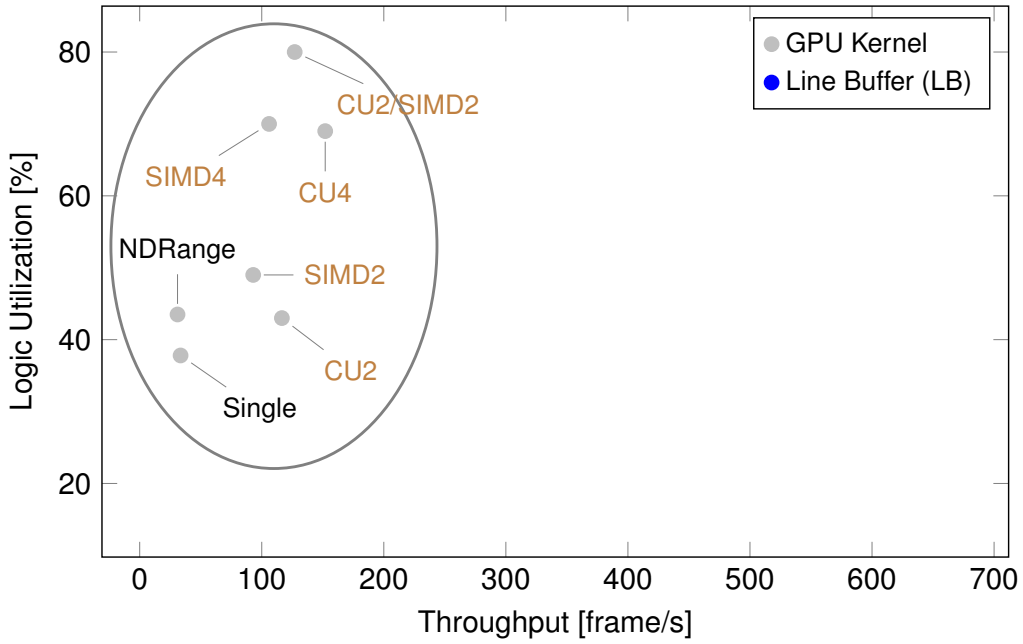


Figure: Design points for a 5×5 Gaussian blur filter.

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)

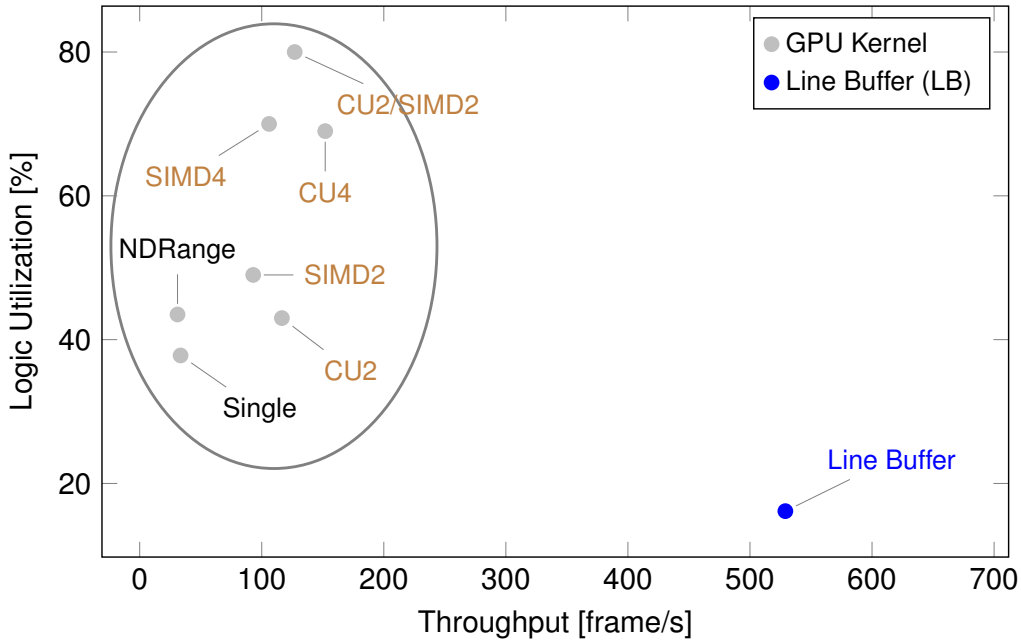


Figure: Design points for a 5×5 Gaussian blur filter.

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)

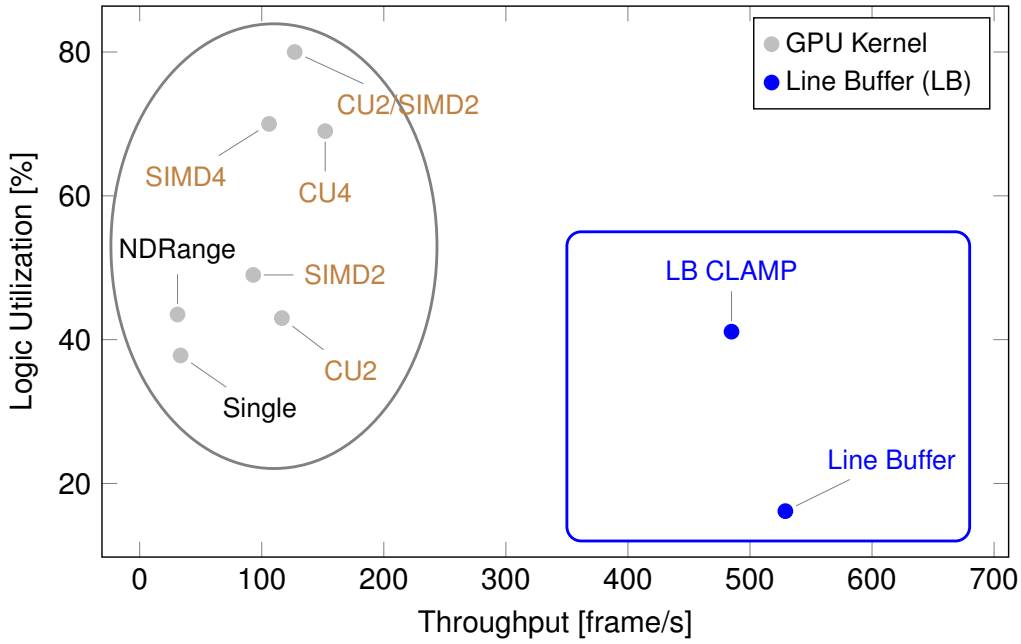


Figure: Design points for a 5×5 Gaussian blur filter.

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)

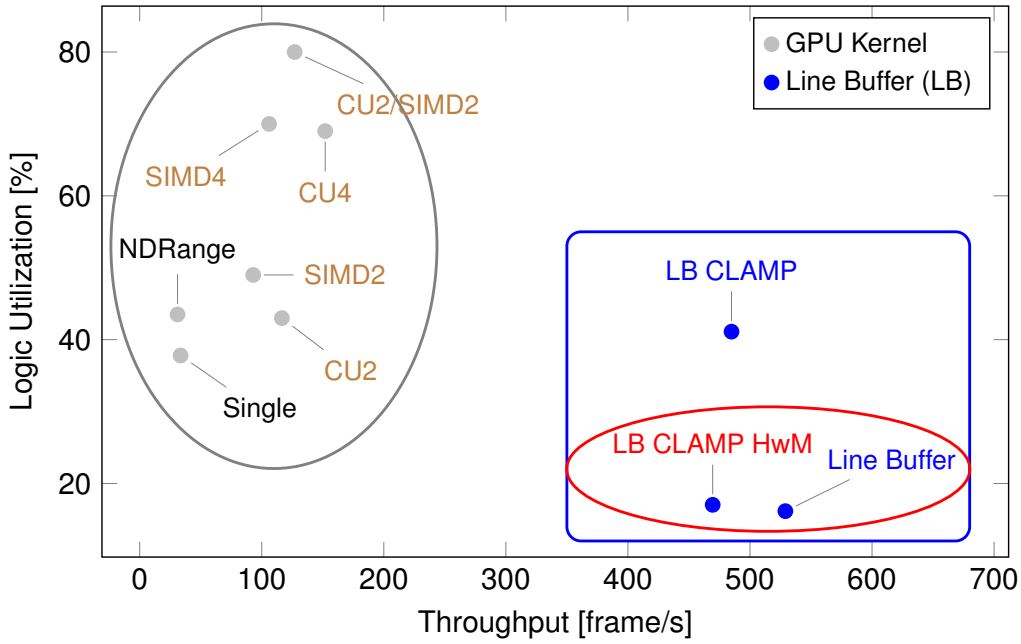


Figure: Design points for a 5×5 Gaussian blur filter.

Outline

Programming Methodology
Developing Software in Hardware Manner

Efficient Code Generation through a Domain-Specific Language

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

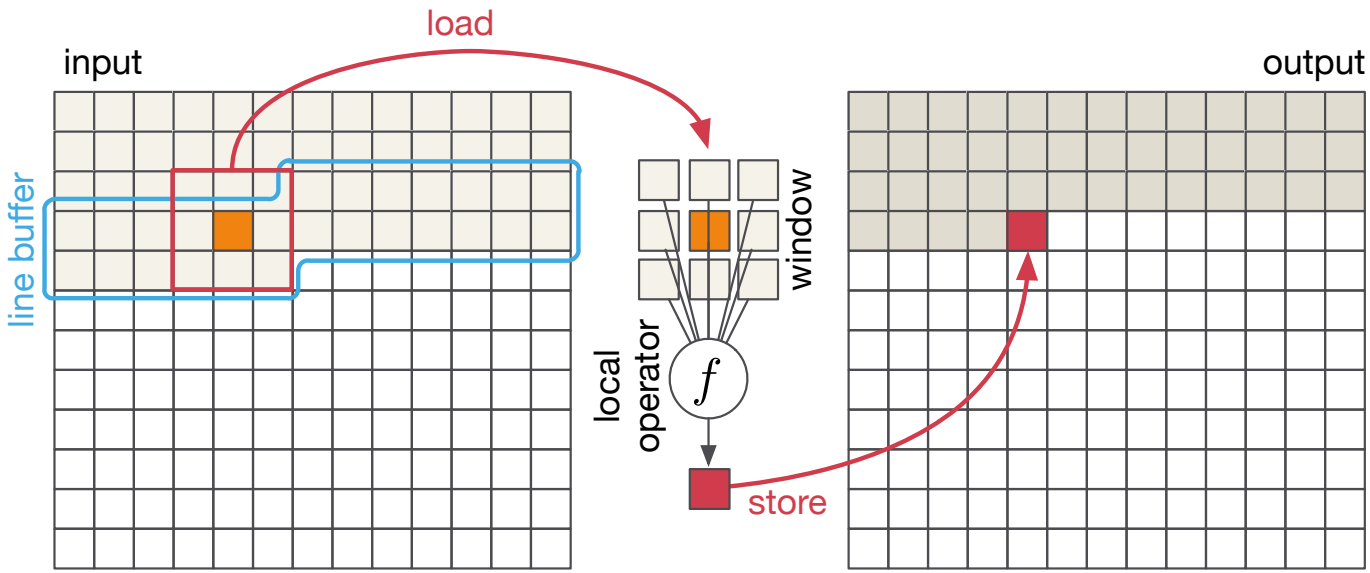
Evaluation and Results

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 = a * 16;
```

Listing 2: Strength Reduction

```
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

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

Listing 2: Remove else condition

```
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 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

```
for (int i = 0; i < 5; ++i){
  for (int j = 0; j < i; ++j){
      /* operations */
  }
}
```

Listing 2: Constant loop bounds

```
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

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

Listing 2: Source Code

```
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

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

Listing 2: Scope Operators

```
/* 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

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

Listing 2: Avoid 2 dimensional arrays

```
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

```
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

```
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

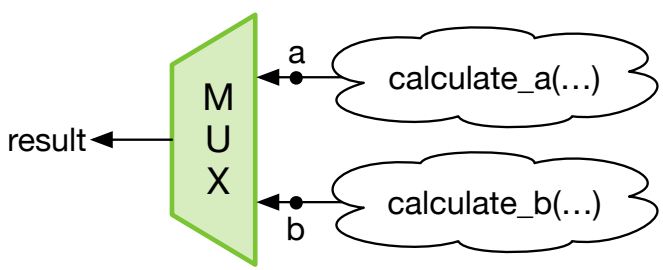
- **Write Altera OpenCL code following these two principles:**
  1. **Calculate all the conditional cases first, decide at the end**
  2. **Exploit temporal locality**

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

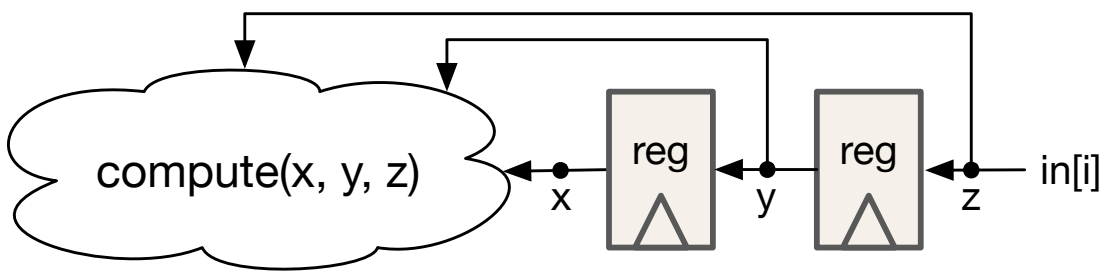
- **Write Altera OpenCL code following these two principles:**
  1. **Calculate all the conditional cases first, decide at the end**
  2. **Exploit temporal locality**

Listing 3: Software Manner

```
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

```
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)

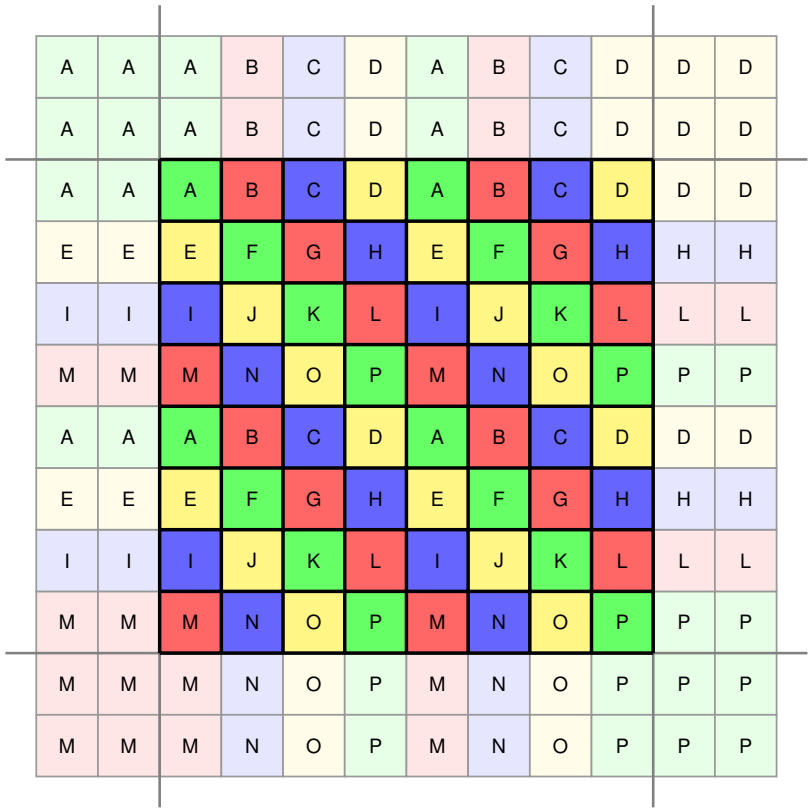


Figure: Clamp boundary condition on an image

Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

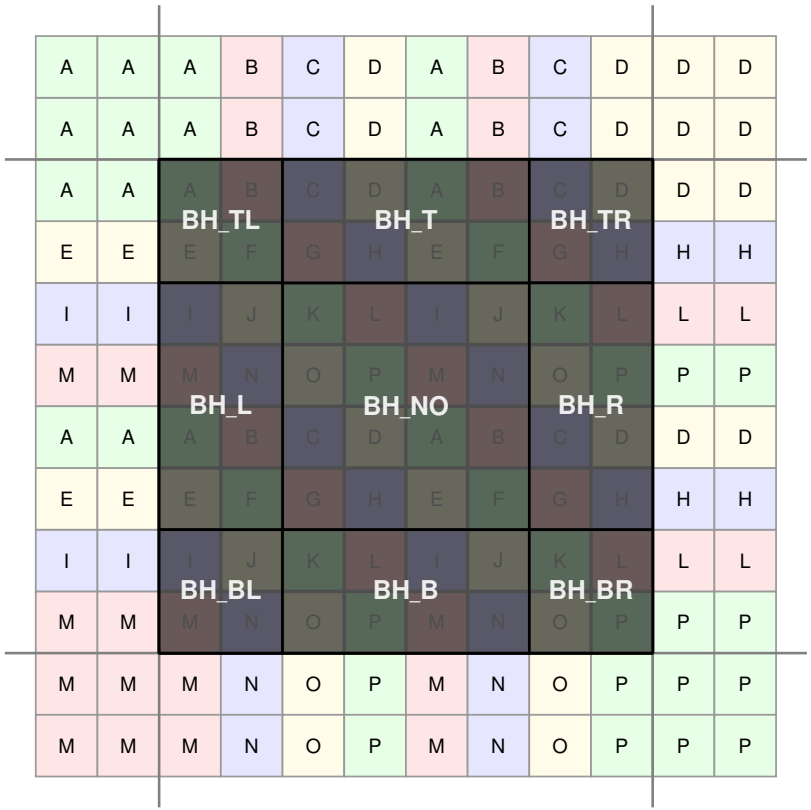
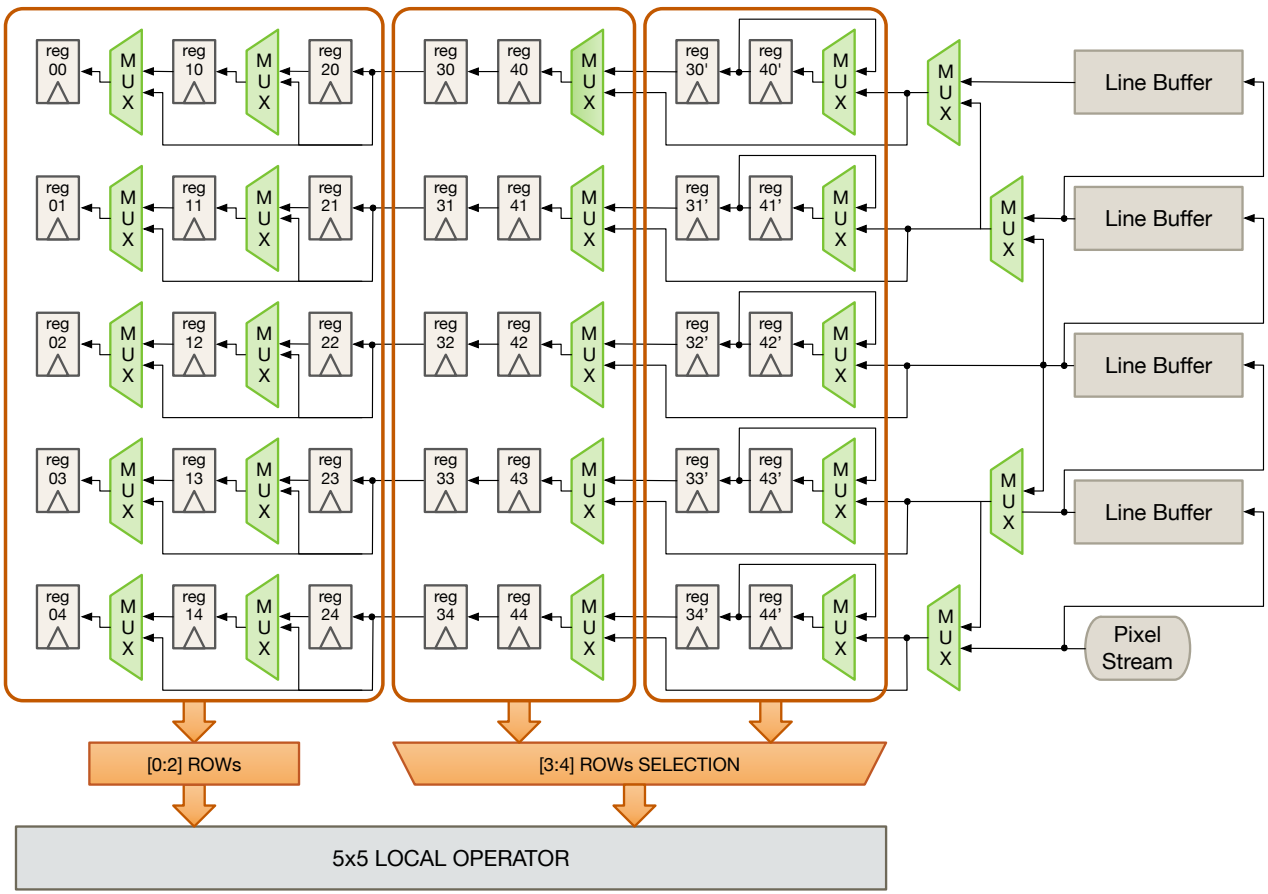
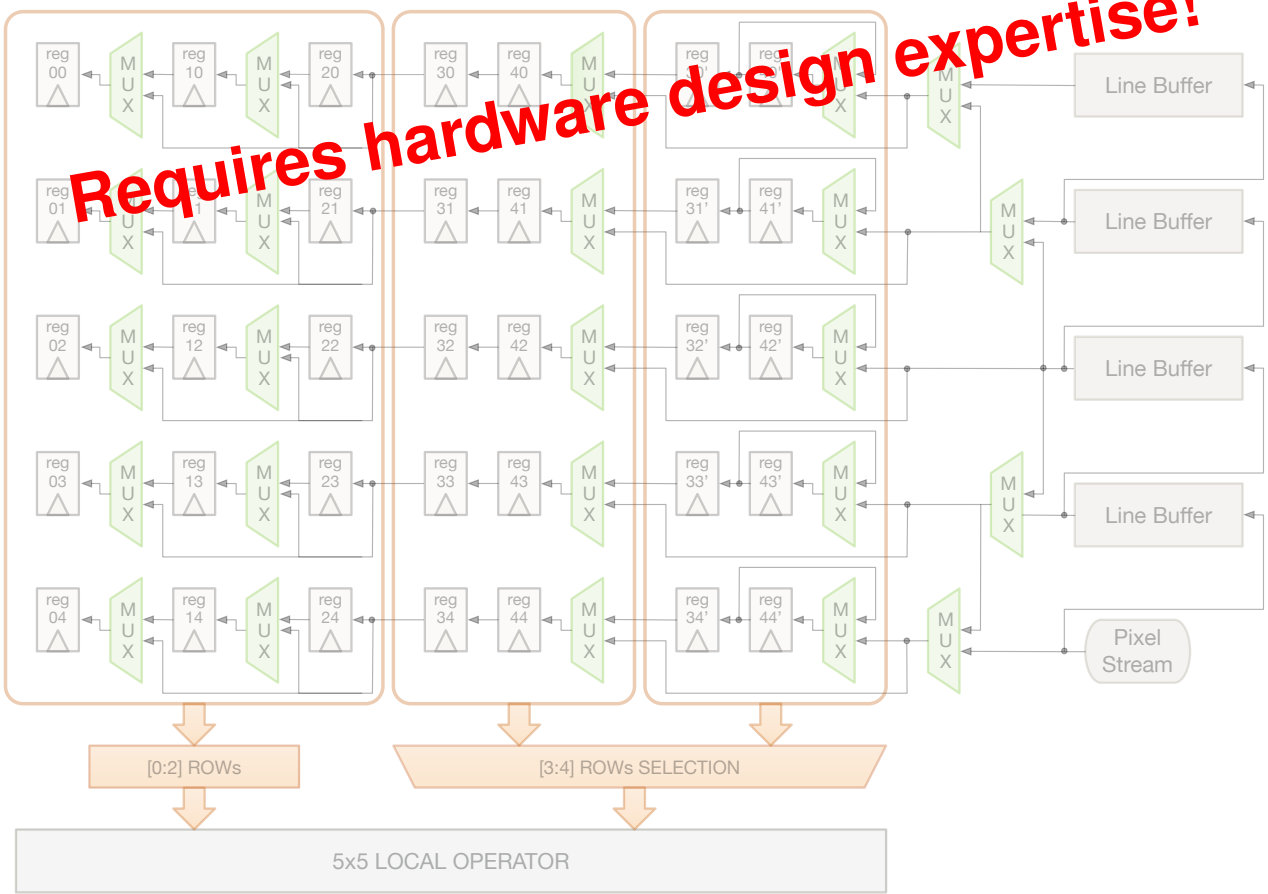


Figure: Clamp boundary condition on an image

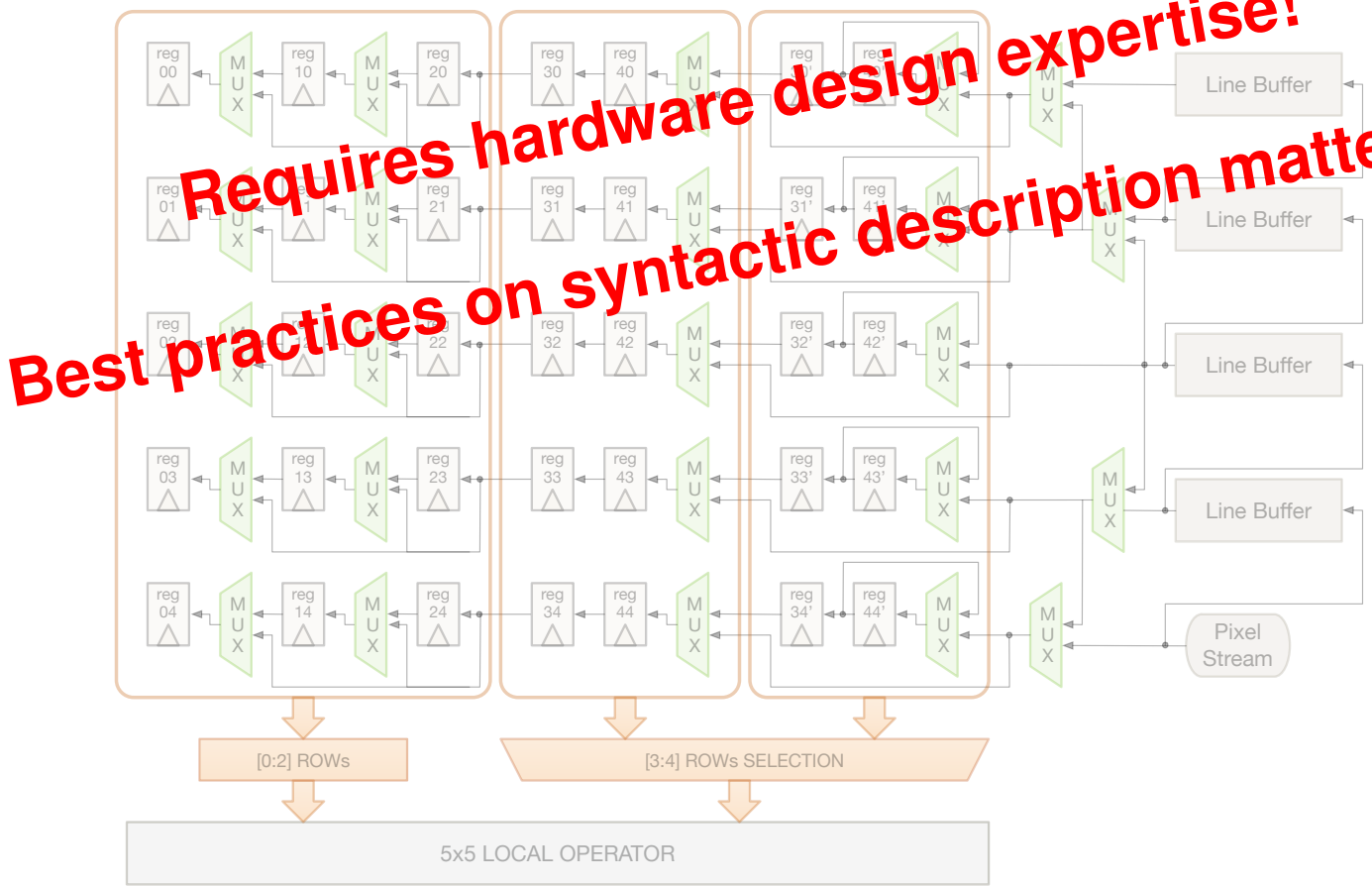
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)



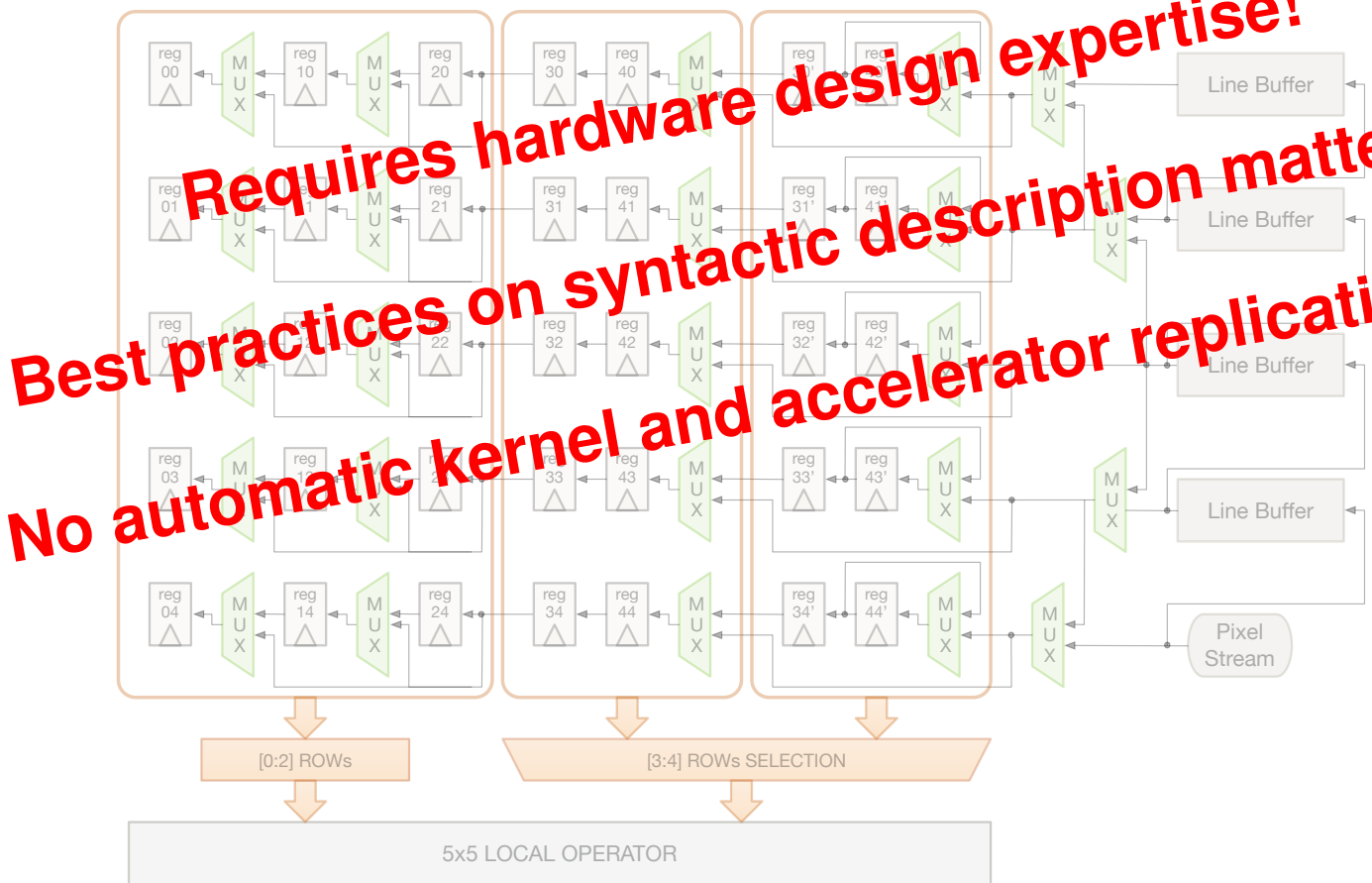
Writing HLS Code for AOC: CLAMP Boundary Handling (HW)



Writing HLS Code for AOC: CLAMP Boundary Handling (HW)



Writing HLS Code for AOC: CLAMP Boundary Handling (HW)

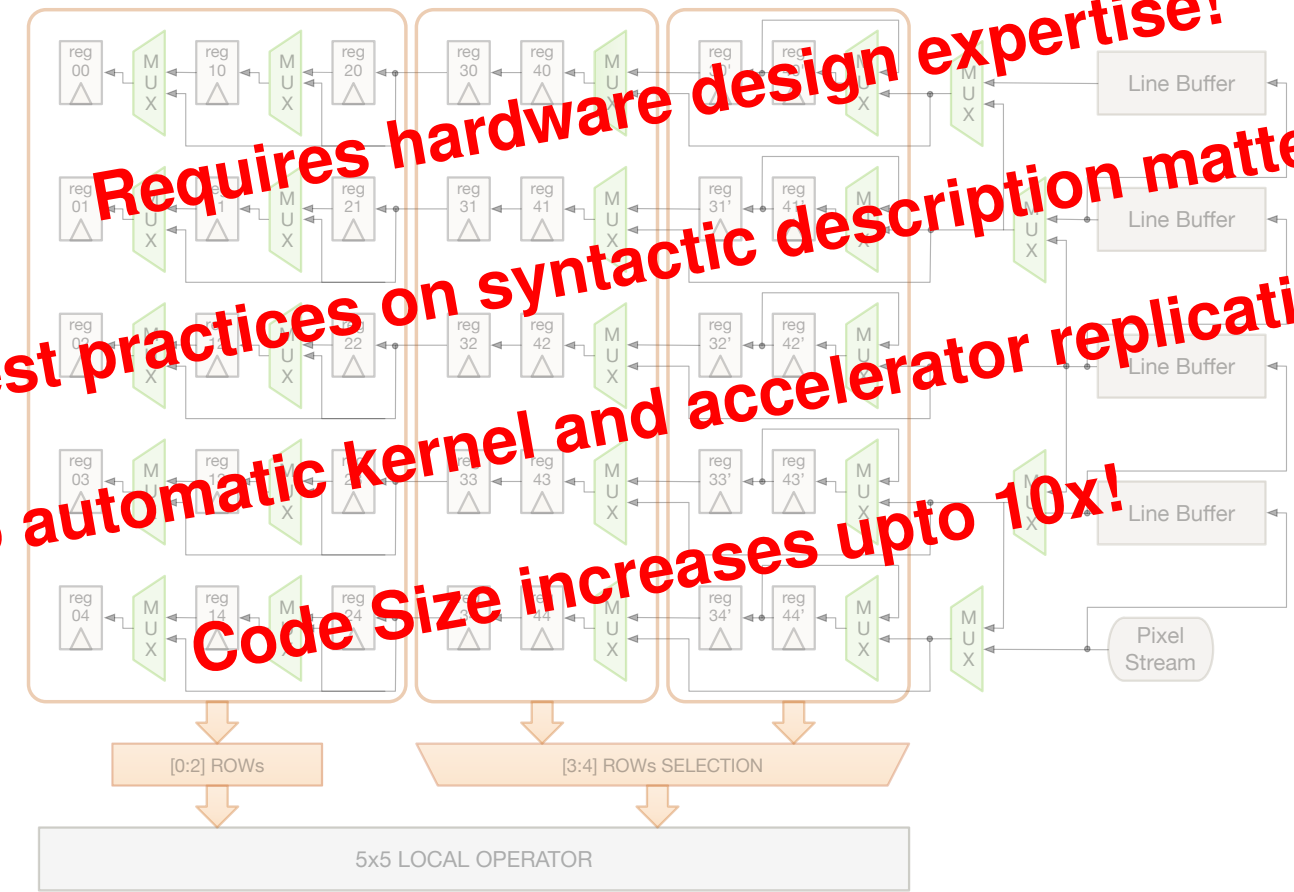


Requires hardware design expertise!

Best practices on syntactic description matters!

No automatic kernel and accelerator replication!

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!

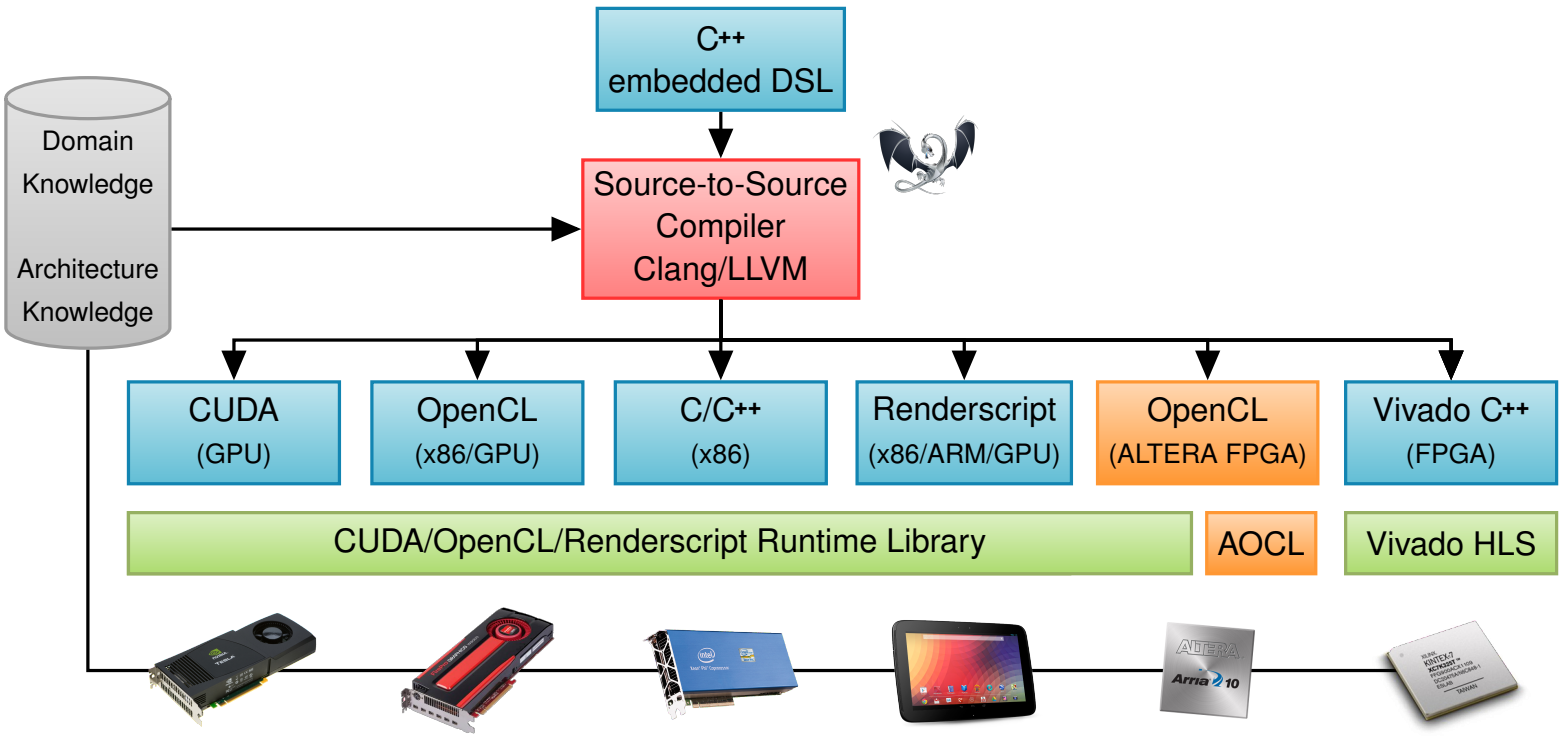
Code Size increases upto 10x!

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

FACULTY OF ENGINEERING

Efficient Code Generation through a Domain-Specific Language

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

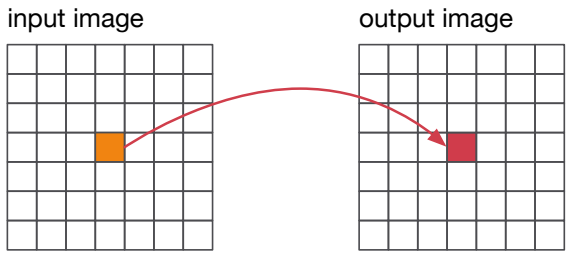


Operators in the Image Processing Domain

We can define three characteristic data operations in the domain:

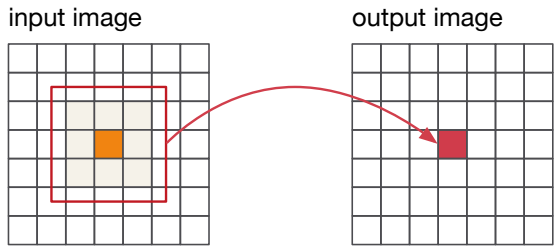
Point Operators:

Output data is determined by single input data



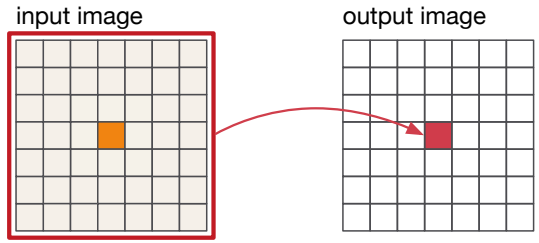
Local Operators:

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. . .

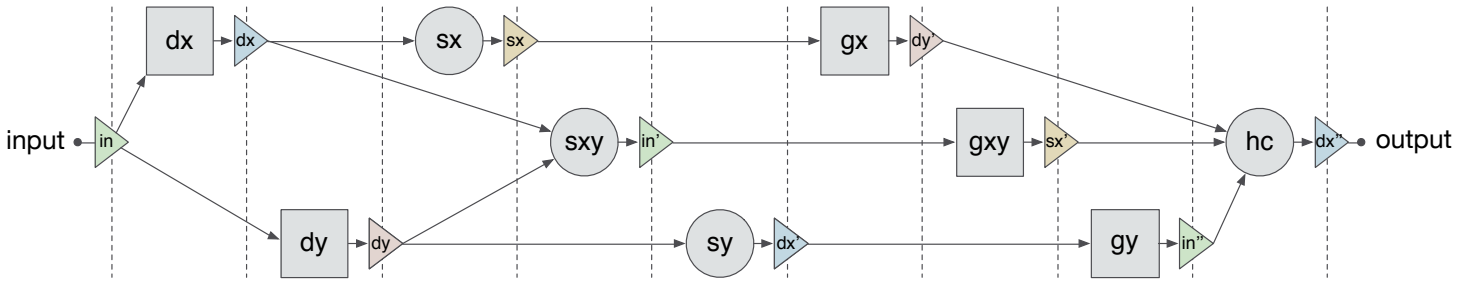


Figure: HIPA^{cc}'s sequential execution for the Harris corner detector

Streaming Pipeline: Harris Corner Detection Example

Transform sequential execution order. . .

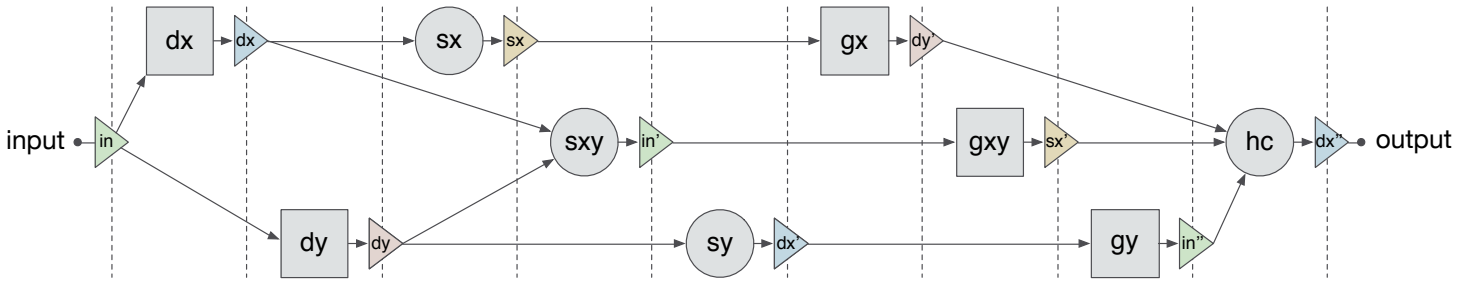


Figure: HIPA^{cc}'s sequential execution for the Harris corner detector

. . . into streaming pipeline of FPGA kernels.

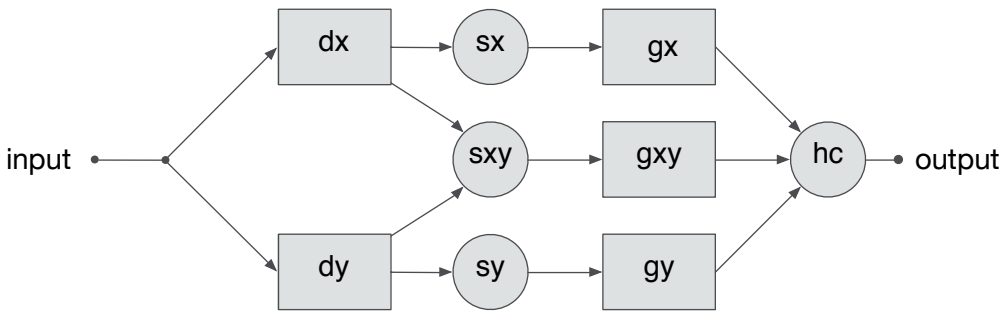


Figure: Representation of AOC kernels

Example: Laplacian Operator

```
// coefficients for Laplacian operator
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

Input Image

```
// 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();
```

Output Image

Example: Laplacian Operator

Input Image

```
// 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();
```

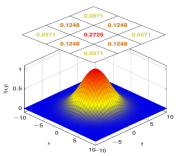
Output Image

Example: Laplacian Operator

Input Image

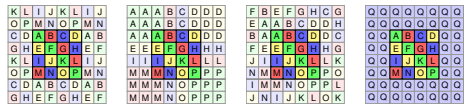
```
// coefficients for Laplacian operator
const int coef[3][3] = { {  0,  1,  0 },
                         {  1, -4,  1 },
                         {  0,  1,  0 } };
// read input
uchar4 *image_bits = readImage();

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();
```

Mask



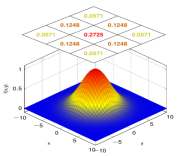
Output Image

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();
```

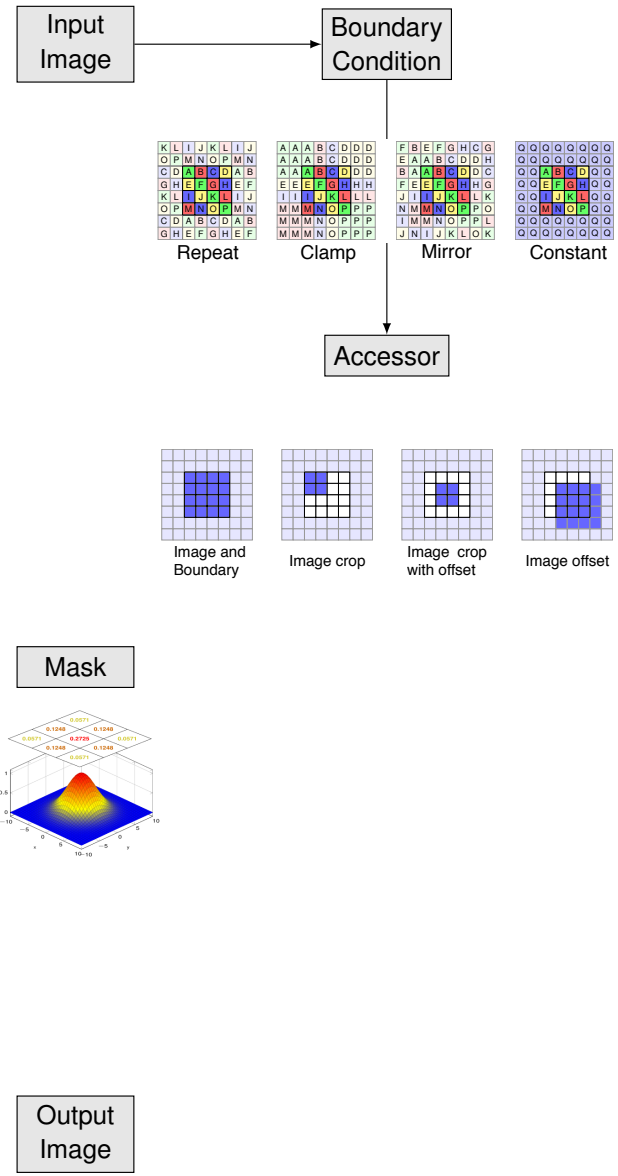
Mask



Output Image

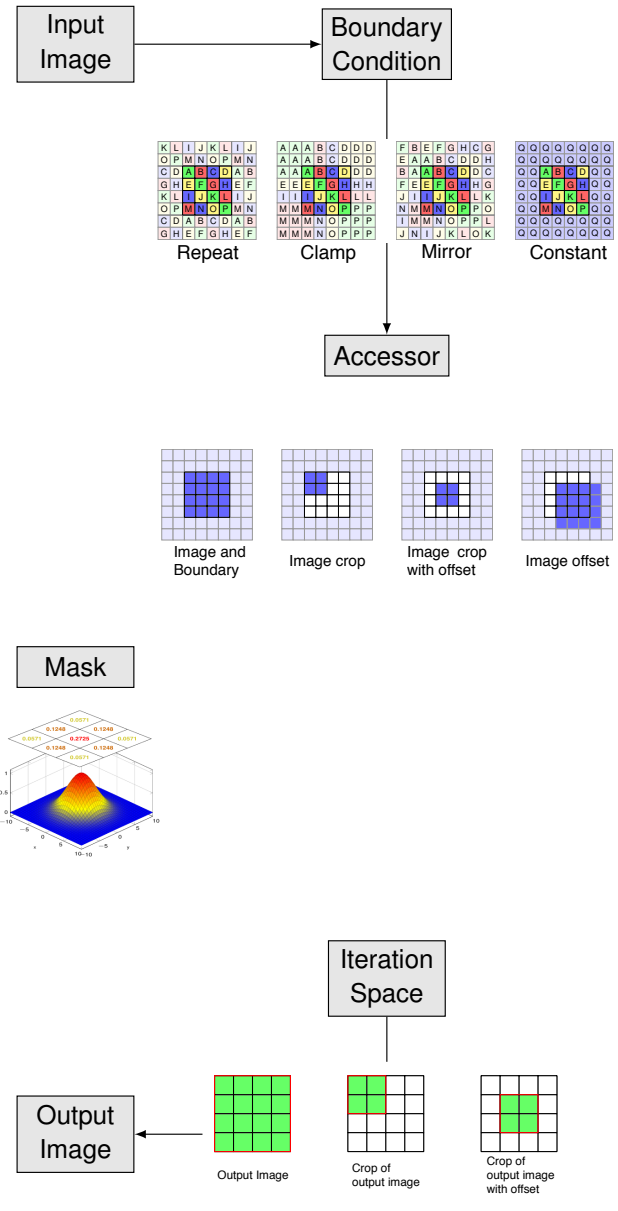
Example: Laplacian Operator

```
// 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();
```



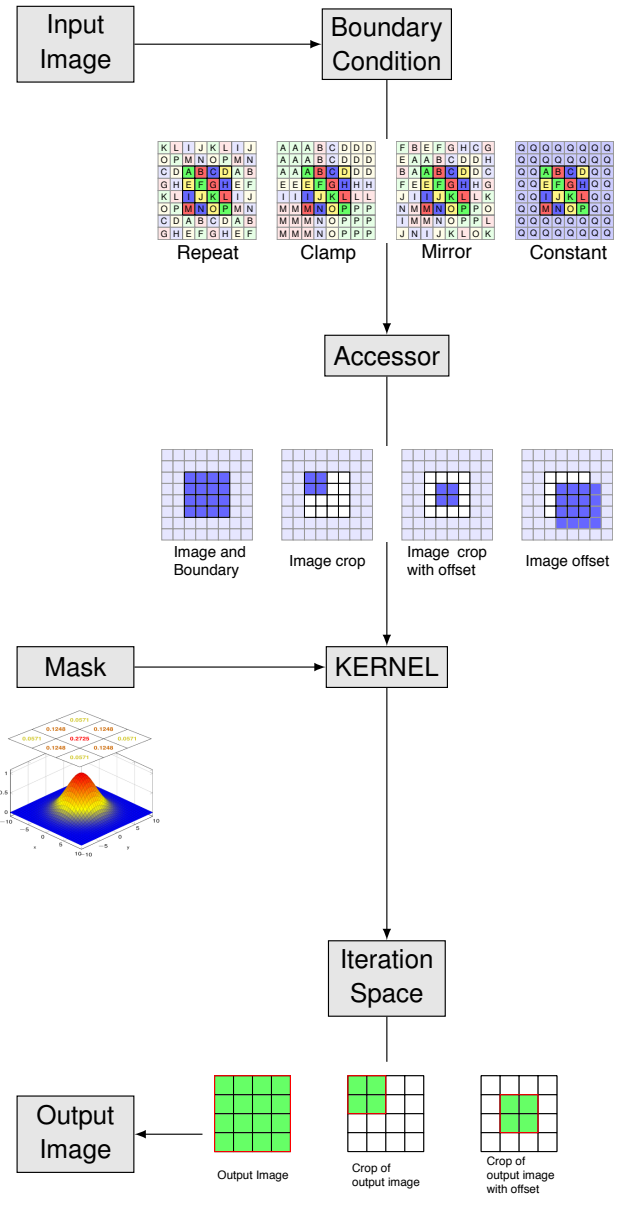
Example: Laplacian Operator

```
// 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();
```



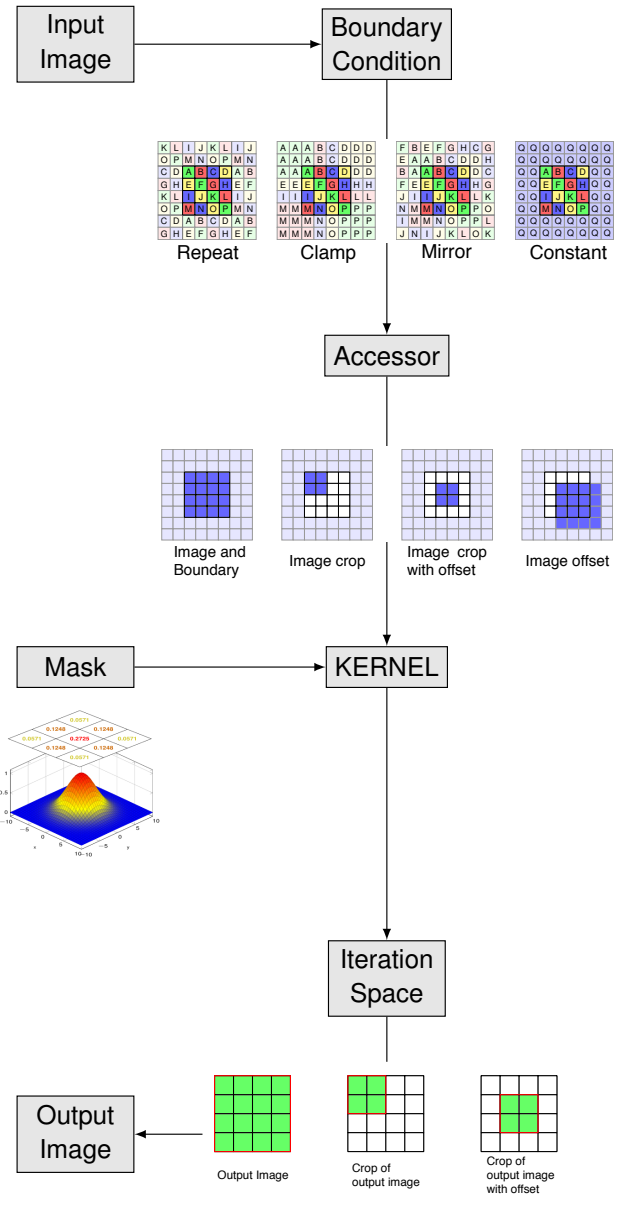
Example: Laplacian Operator

```
// 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();
```



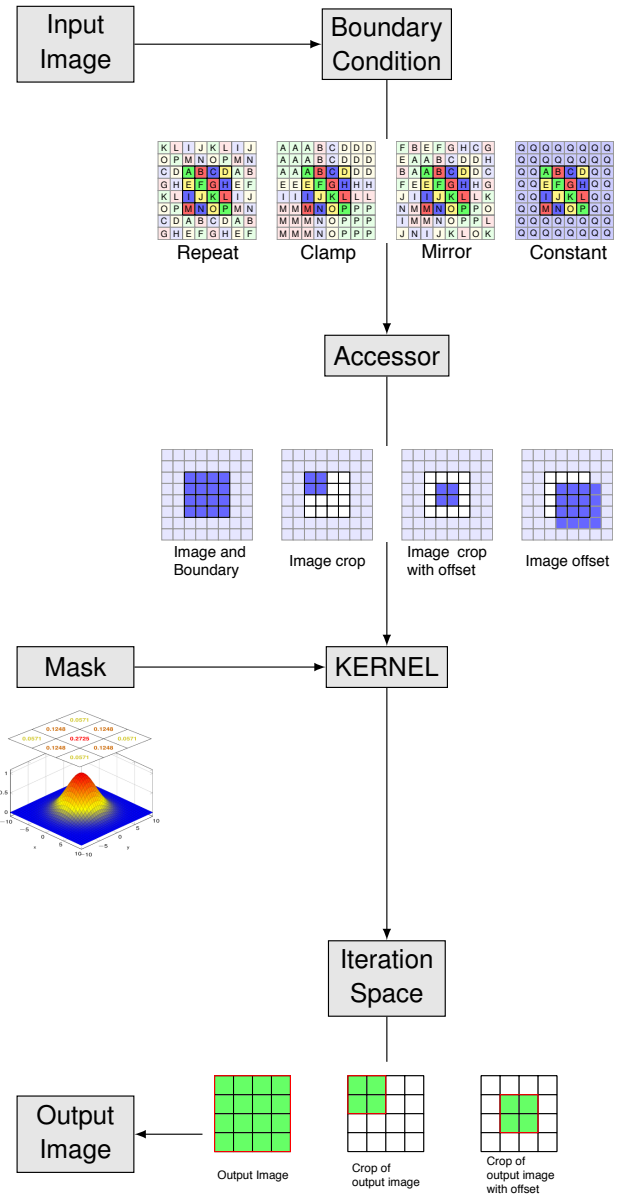
Example: Laplacian Operator

```
// 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 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) {
      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);
    }
};
```

Convolution call

Extensions to HIPA^{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^{cc}

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

```
#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;
```

Automatic Bit Width Reduction: An extension to HIPA^{cc}

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

```
#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

```
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^{cc} vs Handwritten Examples provided by Altera

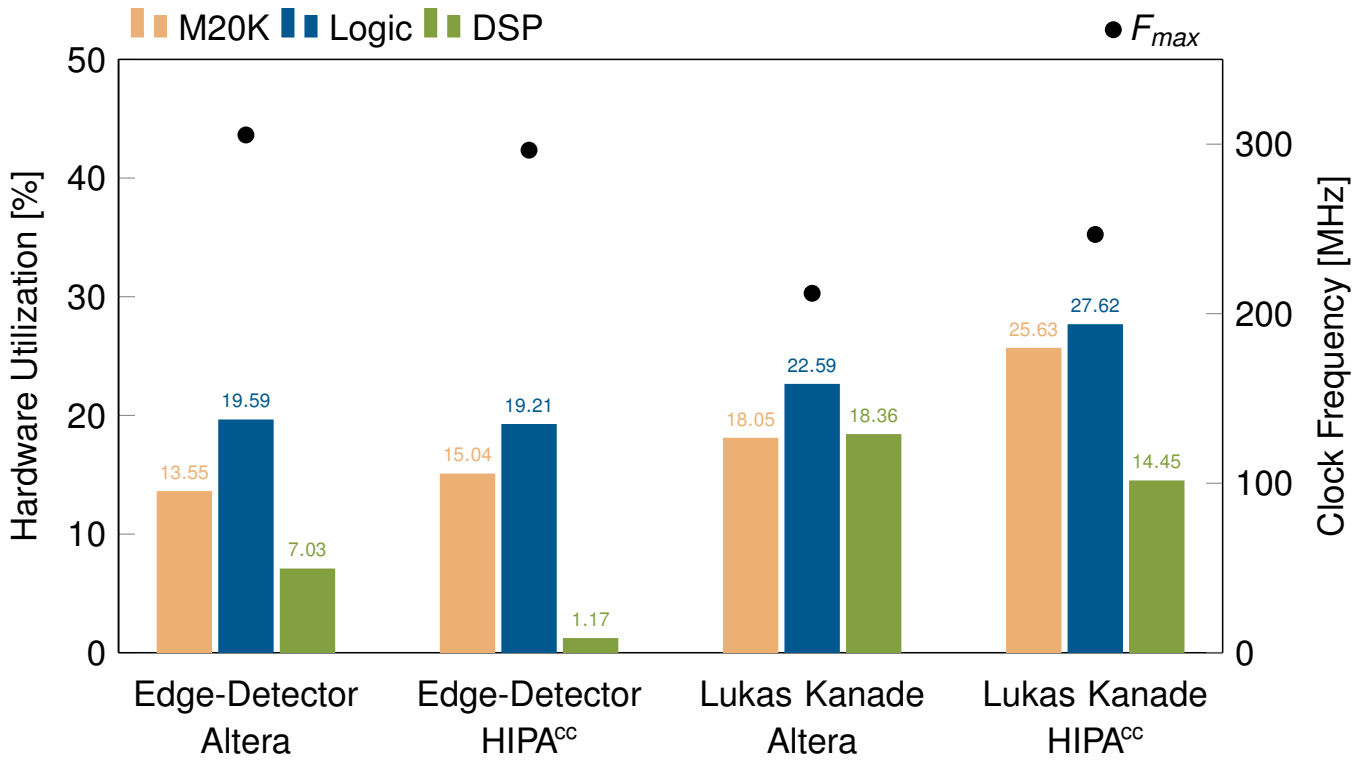


Figure: HIPA^{cc} vs Altera handwritten examples for 1024 $\times$ 1024 image size.

FPGA vs GPU

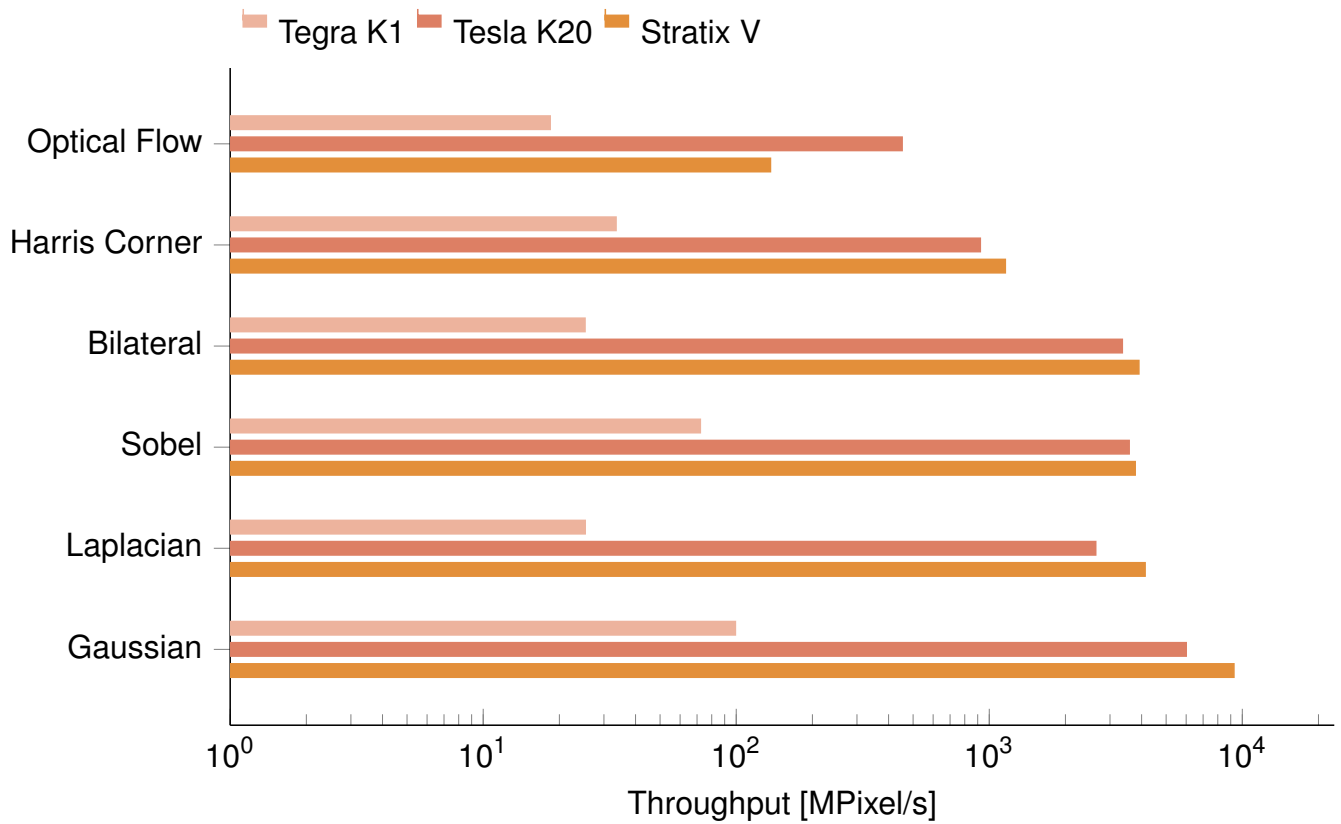


Figure: Comparison of throughput for the Nvidia Tegra K1, Nvidia Tesla K20, and Altera Stratix V.

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^{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

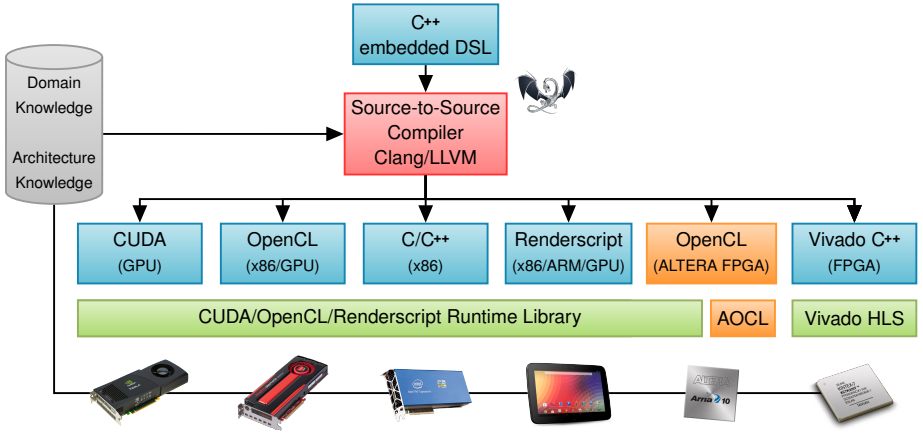
HIPA^{cc} DSL code serves as baseline implementation $\Rightarrow$ Test bench

Develop the algorithm first, decide the target architecture afterwards!

Questions?

Thanks for listening.

Any questions?



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

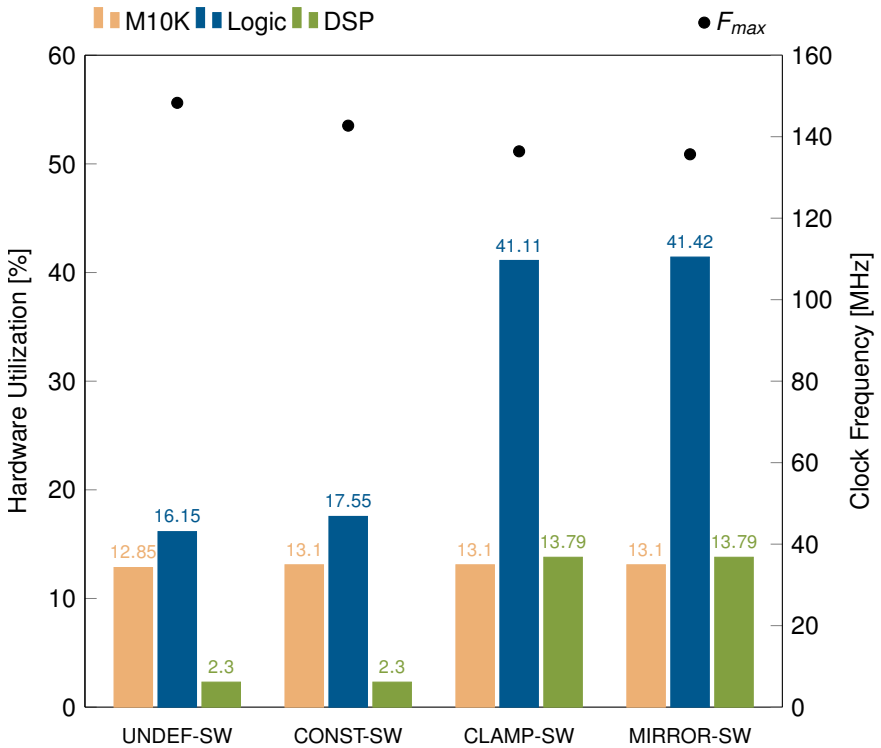
Title FPGA-Based Accelerator Design from a Domain-Specific Language

Speaker M. Akif Özkan, akif.oezkan@fau.de

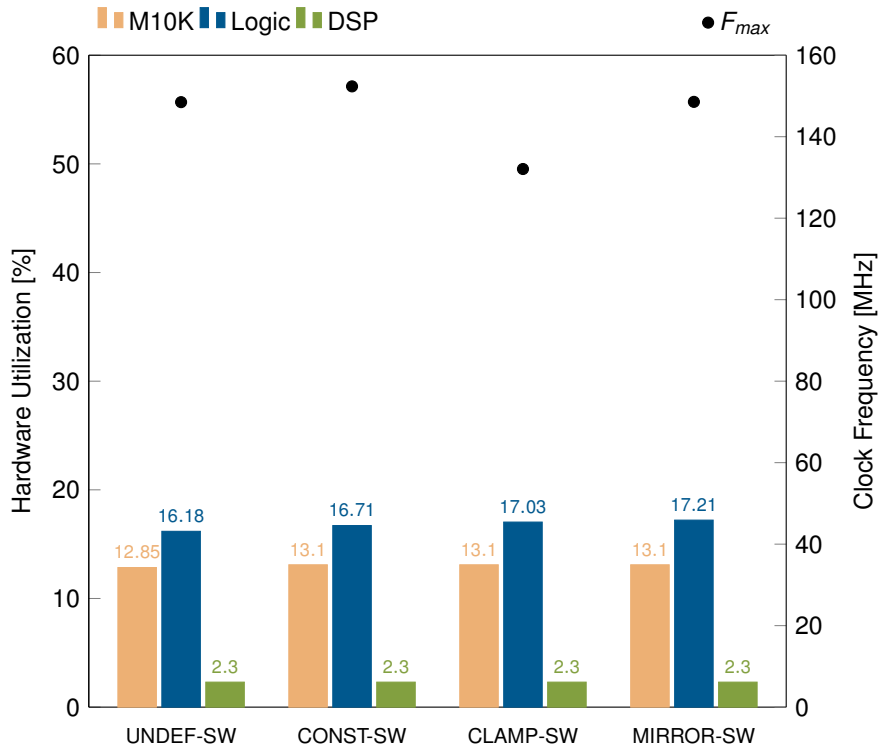
Backup Slides

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 5×5 Gaussian blur on a 1024×1024 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

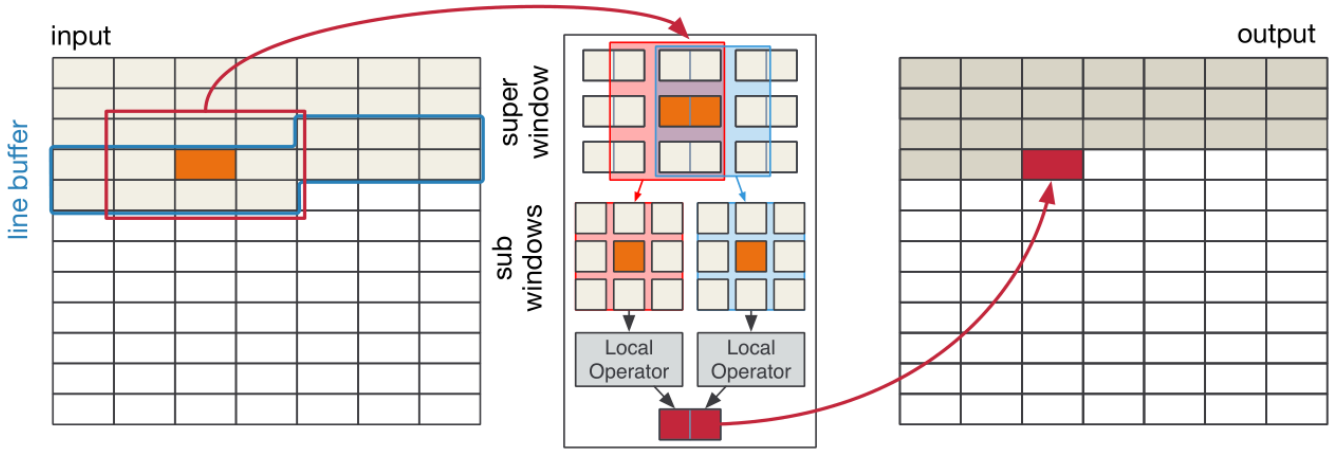
Table: Automatic bit width reduction on local operators of size 3×3 with image size 1024×1024.

Filter	Type	II	ALUTs	Registers	Logic (%)	M10K	DSP	Freq [MHz]
Sobel	normal	2	8552	12433	19.76	84	4	131.25
	reduced	2	8230	12098	**19.11**	84	3	**152.09**
Gaussian	normal	2	8593	12423	19.86	84	4	132.50
	reduced	2	8439	11843	**19.26**	83	3	**153.11**

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

[1] Moritz Schmid et al. “Loop Coarsening in C-based High-Level Synthesis”. In: *Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. (Toronto, Canada). IEEE, July 27–29, 2015, pp. 166–173

Kernel Vectorization

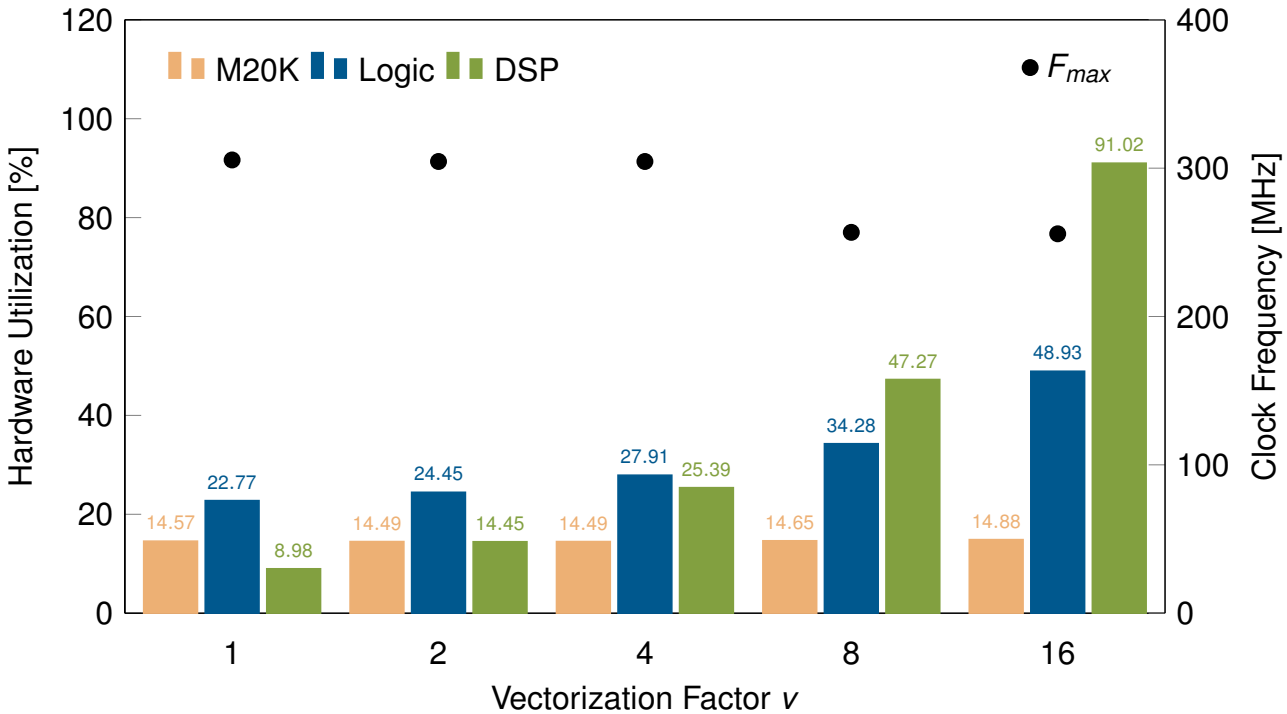


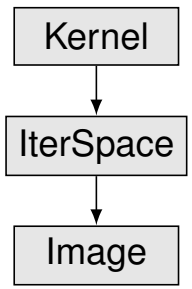
Figure: Vectorization of a 3×3 bilateral filter with clamping on an image of size 1024×1024.

Generating the Streaming Pipeline

Generating the Streaming Pipeline

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

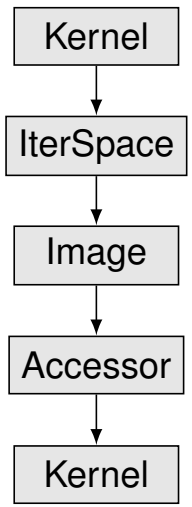
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

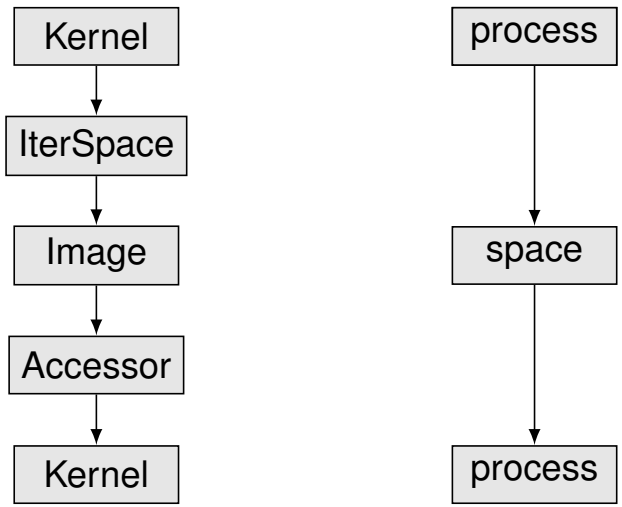
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

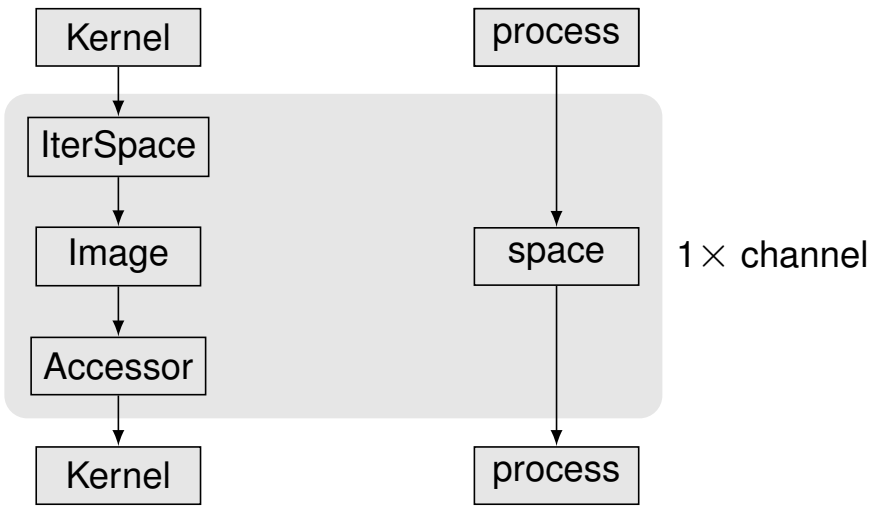
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

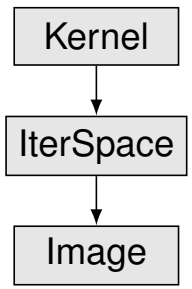
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

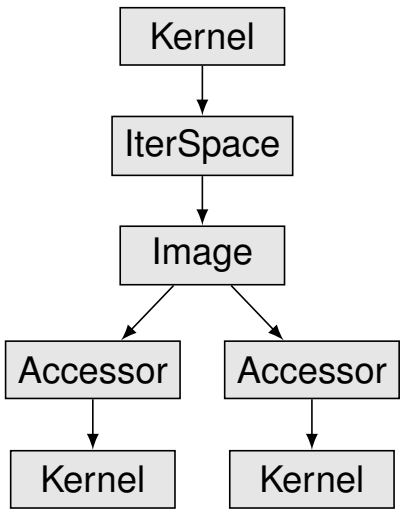
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

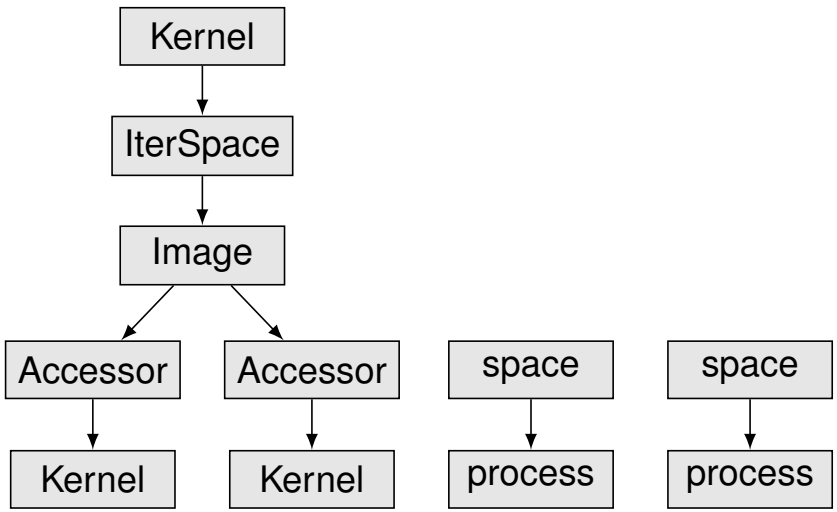
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

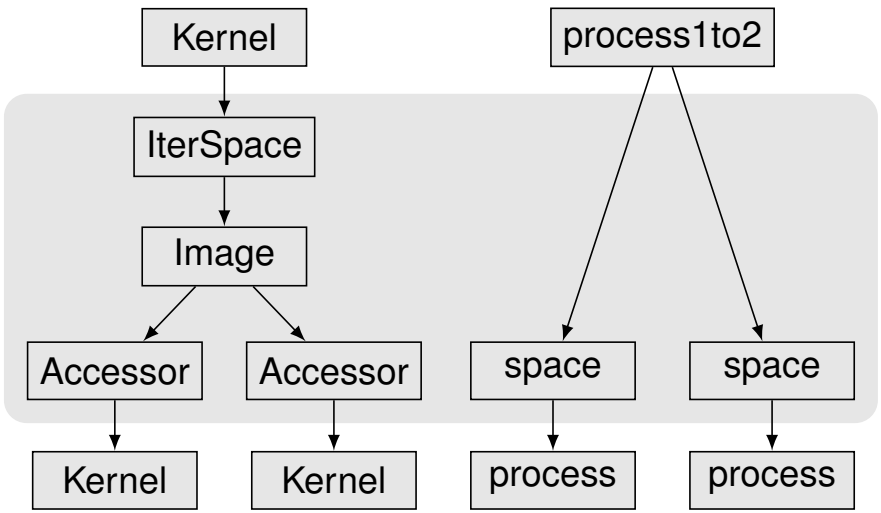
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

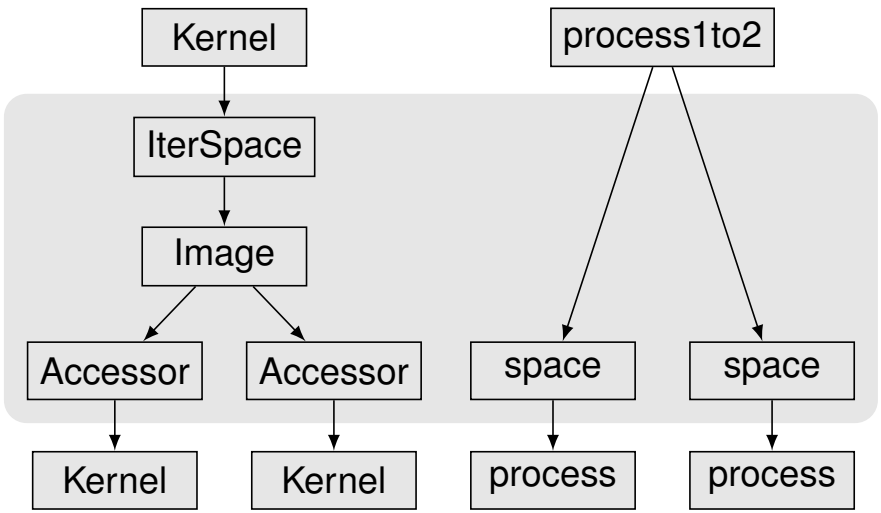
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Generating the Streaming Pipeline

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

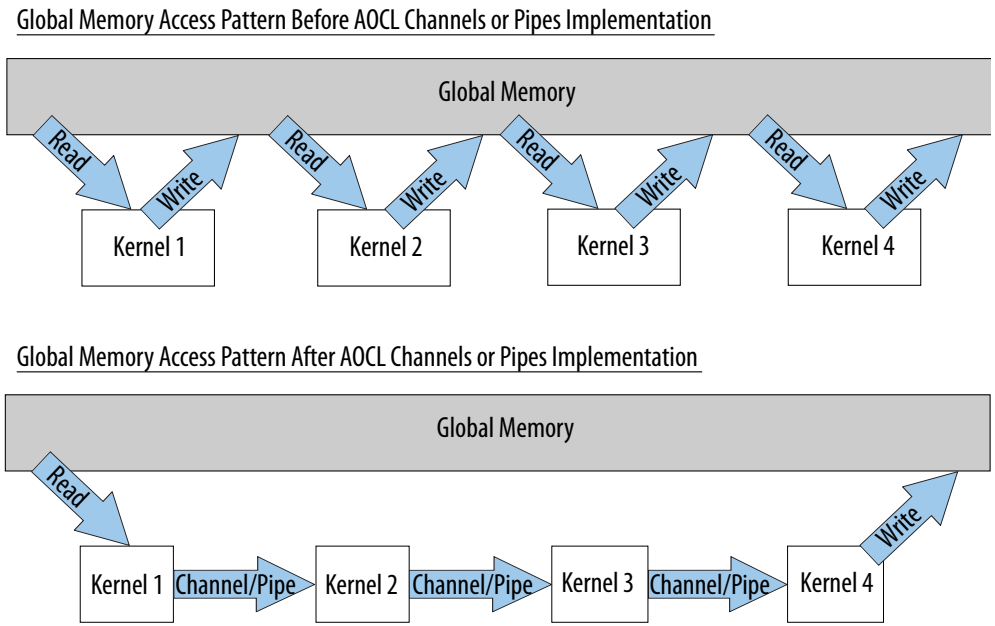
- model as combination of *processes* and *spaces*
- create unique channel objects for each *space*
- identify memory reuse and utilize processes accordingly
- build dependency graph
- traverse in depth-first search starting from output *spaces*



Altera OpenCL System Level Interfaces

Altera OpenCL System Level Interfaces

Channels Data sharing between kernels



Altera, SDK. for OpenCL: Best Practices Guide, Altera, 2016.

Altera OpenCL System Level Interfaces

Channels Data sharing between kernels

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

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

- Channels use considerable amount of logic!

Altera OpenCL System Level Interfaces

Channels Data sharing between kernels

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

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

- 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.

Kernel	Type	II	ALUTs	Registers	LU (%)	BRAM	DSP	Freq [MHz]
Copy Kernel	single	1	6821	7812	14.04	43	0	152.89
Mean Filter	single	1	8248	9415	16.78	51	0	151.52
2 Mean Filter	separate	1	12104	15132	25.36	100	0	145.78
2 Mean Filter	channel	1	10439	12466	21.47	60	0	154.14
2 Mean Filter	fused	1	9480	11715	19.47	64	0	122.47
3 Mean Filter	channel	1	12829	15451	26.56	67	0	145.57
3 Mean Filter	fused	1	10899	13651	22.24	79	0	125.00
Luma+Mean	single	1	8244	9694	16.84	66	2	132.56
Luma+Mean	channel	1	11281	13653	23.75	64	3	148.75
Luma+Mean	fused	1	9635	11846	20.09	72	2	123.53
Harris corner	channel	1	43127	54334	89.87	186	9	142.52
Harris corner	fused	1	33263	43319	70.87	177	9	107.35

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