TAMPERE UNIVERSITY OF TECHNOLOGY

Rapid Customization of Image Processors Using Halide

Ville Korhonen, Pekka Jääskeläinen, Matias Koskela, Timo Viitanen, and Jarmo Takala

Tampere University of Technology, Finland

Motivation

- Exploit custom operations in computing platform at high abstraction level description
  - avoid error-prune low-level descriptions
  - maintain platform portability
- We demonstrate an dxperimental design flow
  - High-abstraction level descriptions: Halide
  - Processor customization: TCE toolset

Halide (MIT)

- Domain specific (image processing) functional parallel programming language developed at MIT
- Decoupling of algorithm and its schedule
  - The same algorithm easily optimized for different types of processors, only by modifying the schedule
- Algorithm part requires only little knowledge about parallel programming or parallel compute platforms

C/C++ vs. Halide

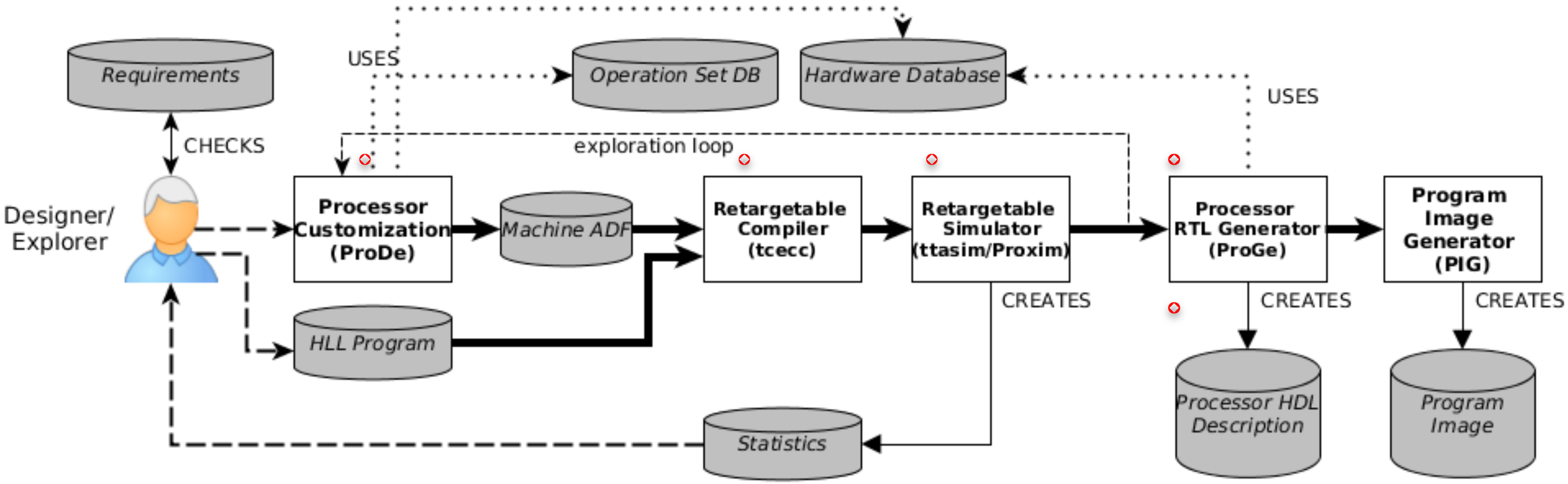
C++ function for “blur”

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate blurx array
    for (int xTile = 0; xTile < in.width; xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__mm128*)(inPtr-1));
          b = _mm_loadu_si128((__mm128*)(inPtr+1));
          c = _mm_loadu_si128((__mm128*)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          intPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          a = _mm_load_si128(blurxPtr+256/8);
          a = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(1, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Halide description for “blur”

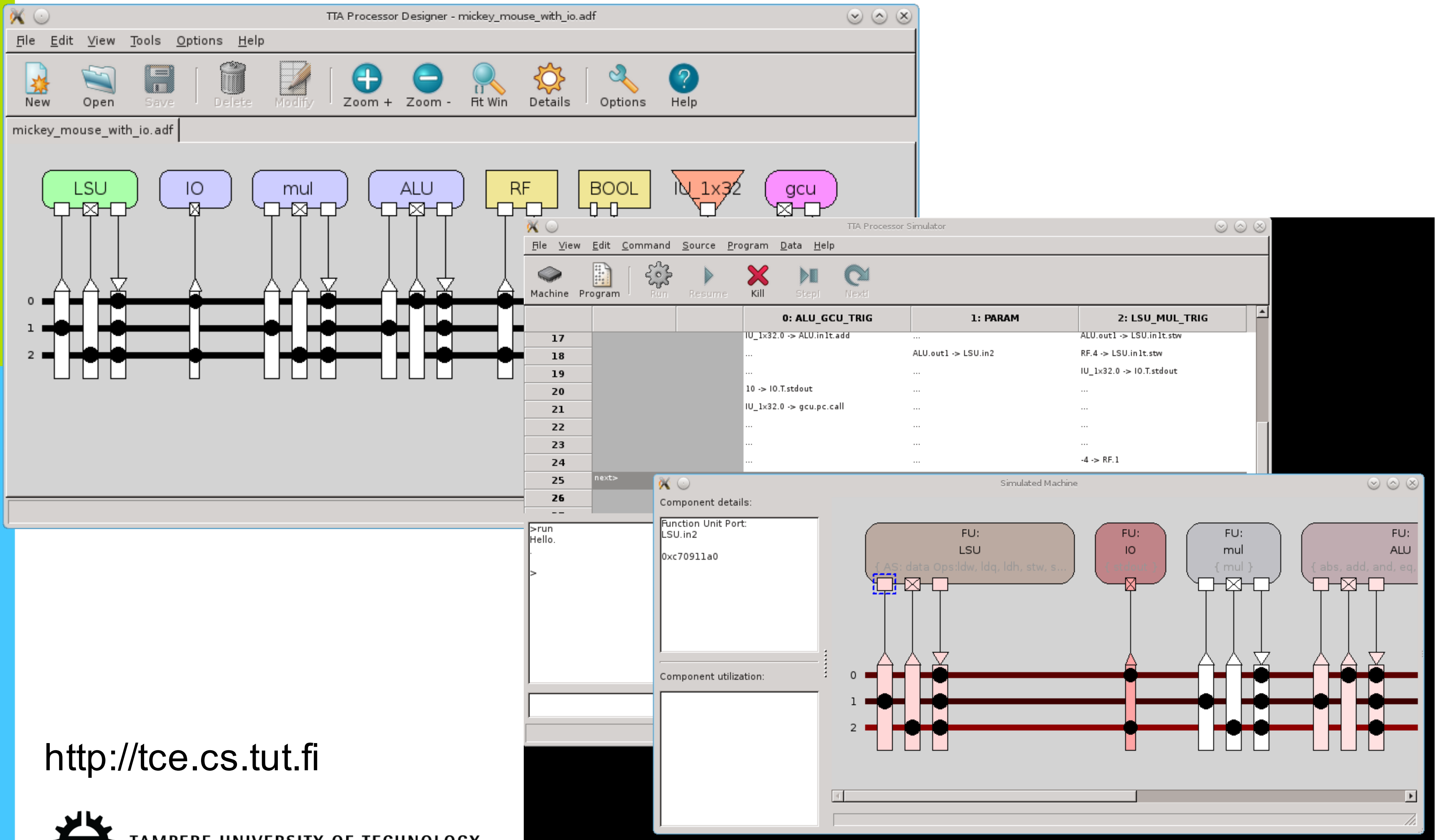
```
// The algorithm
blur_x(x, y) = (input(x, y)
               + input(x+1, y)
               + input(x+2, y))/3;
blur_y(x, y) = (blur_x(x, y)
               + blur_x(x, y+1)
               + blur_x(x, y+2))/3;

// How to schedule it
blur_y.split(y, y, yi, 8).parallel(y)
  .vectorize(x, 8);
blur_x.store_at(blur_y, y).compute_at(blur_y, yi)
  .vectorize(x, 8);
```

TTA-Based Co-Design Environment (TCE) (Tampere Univ. Tech.)



- Framework for customizing processor architectures and retargetable compiler
  - Exploits transport triggered architecture (TTA) template
  - Supports C/C++ and OpenCL
- A research platform for customized processors, compilation techniques etc.
- MIT-licensed, available at http://tce.cs.tut.fi

TCE Screenshots



http://tce.cs.tut.fi

TAMPERE UNIVERSITY OF TECHNOLOGY

Custom Operations

- Custom operation (special instruction) is an optimized atomic operation that usually wraps up the behavior of multiple basic operations.
- Simple custom operations automatically invoked by compiler (LLVM)
  - E.g., multiply-accumulate, auto-increment in addressing

Custom Operations

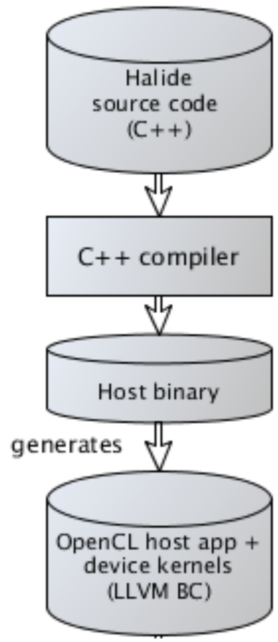
- More complex custom operations need to be created
  - Custom op created as a simulation model in TCE (for implementation the corresponding HDL has to be created)
  - TCE generates intrinsics for the custom op and corresponding function wrapper for Halide
  - User indicates the use of custom op with extern declaration

Halide-OpenCL Flow

Halide:

- Application compiled to executable binary for the target
- Application wraps the algorithm and the schedule into an OpenCL application presented in LLVM IR (OpenCL kernels included as global strings)
- Compiles LLVM IR for the target

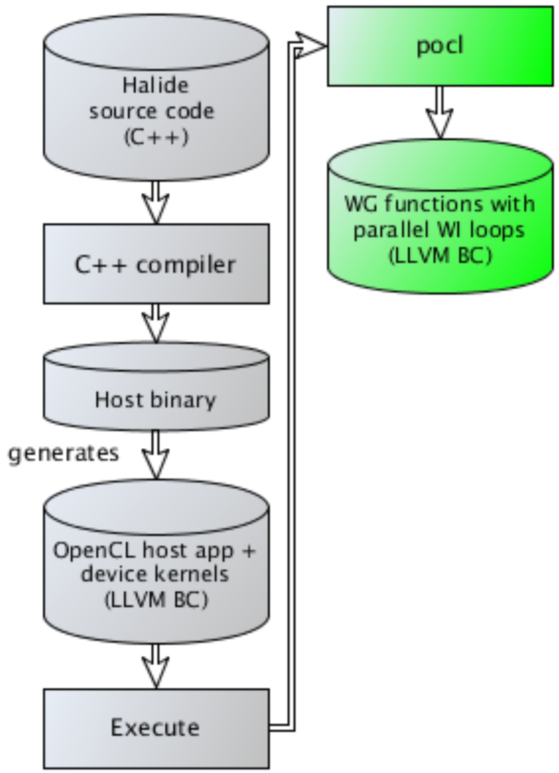
(offline and online compilation possible)



Halide-OpenCL-TCE Flow

OpenCL:

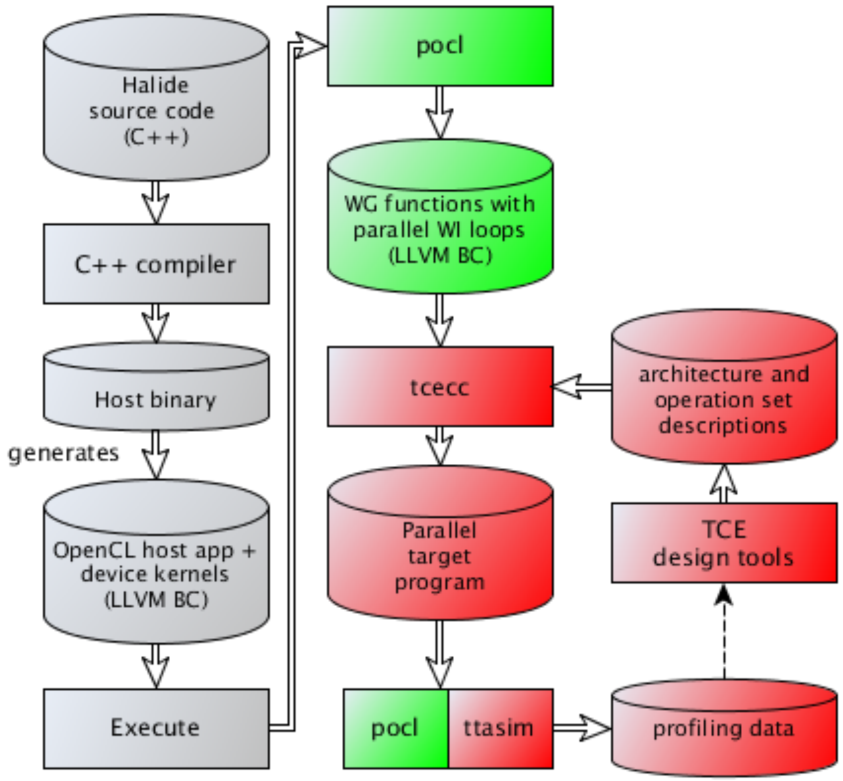
- OpenCL application uses pocl through the OpenCL API
  - pocl is our open source implementation of OpenCL standard: http://pocl.sourceforge.net/
- OpenCL takes care of platform portability, parallelism on higher level and heterogeneity.
- pocl compiles the plain text kernels to Workgroup functions
- pocl gathers execution times of the kernels for profiling purposes



Halide-OpenCL-TCE Flow

TCE:

- Processor design tools
- Custom operation definitions
- Simulation models for custom ops
- Function wrapping for custom ops
- Retargetable compiler
  - adapts to changes in the processor architecture on the fly
- Profiling tools for evaluating performance



Experiment Case 1: Blur

- Blurs image by calculating weighted average over adjacent pixels
- Starting point
  - minimalistic scalar integer TTA processor
- Accelerated with a custom op
  - weighted average

Case 1: Blur

```
// Take a color 8-bit input
Image<uint8_t> input = load<uint8_t>("rgb.png");

// Upgrade it to 16-bit, so we can do math without it overflowing.
Func input_16("input_16");
input_16(x, y, c) = cast<uint16_t>(input(x, y, c));

// Blur it horizontally:
Func blur_x("blur_x");
blur_x(x, y, c) = (input_16(x-1, y, c)
                  + 2*input_16(x, y, c)
                  + input_16(x+1, y, c))/4;

// Blur it vertically:
Func blur_y("blur_y");
blur_y(x, y, c) = (blur_x(x, y-1, c)
                  + 2*blur_x(x, y, c)
                  + blur_x(x, y+1, c))/4;
```

Potential custom operation ("*2" and "/4" are only rewiring inside custom op) — for blur_x

Potential custom operation — for blur_y

Weighted average custom op

```
HalideExtern_3 (uint8_t, _tce_wavg3, uint8_t, uint8_t, uint8_t);

int main(int argc, char **argv) {
    // First we'll declare some Vars to use below.
    Var x("x"), y("y"), c("c");

    // Take a color 8-bit input
    Image<uint8_t> input = load<uint8_t>("rgb.png");

    // Blur it horizontally:
    Func blur_x("blur_x");
    blur_x(x, y, c) = _tce_wavg3(input(x-1, y, c),
                                 input(x, y, c),
                                 input(x+1, y, c));
    // Blur it vertically:
    Func blur_y("blur_y");
    blur_y(x, y, c) = _tce_wavg3(blur_x(x, y-1, c),
                                 blur_x(x, y, c),
                                 blur_x(x, y+1, c));
```

Custom op declaration

Custom op

Same custom op again

Case 2: Bilateral grid

- Edge preserving blur
- Starting point:
  - minimalistic scalar integer + float TTA processor
- Accelerated with custom ops:
  - Blur
  - 3D linear interpolation

Case 2: Bilateral grid

```
// Blur the grid using a five-tap filter
Func blurx("blurx"), blury("blury"), blurz("blurz");

blurz(x, y, z, c) = (histogram(x, y, z-2, c) +
                     histogram(x, y, z-1, c)*4 +
                     histogram(x, y, z  , c)*6 +
                     histogram(x, y, z+1, c)*4 +
                     histogram(x, y, z+2, c));
blurx(x, y, z, c) = (blurz(x-2, y, z, c) +
                     blurz(x-1, y, z, c)*4 +
                     blurz(x  , y, z, c)*6 +
                     blurz(x+1, y, z, c)*4 +
                     blurz(x+2, y, z, c));
blury(x, y, z, c) = (blurx(x, y-2, z, c) +
                     blurx(x, y-1, z, c)*4 +
                     blurx(x, y  , z, c)*6 +
                     blurx(x, y+1, z, c)*4 +
                     blurx(x, y+2, z, c));
```

Potential custom op

Case 2: Bilateral grid

```
HalideExtern_5(float, _tce_wavg5f, float, float, float, float, float);

int main(int argc, char **argv) {
    •
    •
    •
    // Blur the grid using a five-tap filter
    Func blurx("blurx"), blury("blury"), blurz("blurz");
    blurz(x, y, z, c) = _tce_wavg5f(histogram(x, y, z-2, c),
                                    histogram(x, y, z-1, c),
                                    histogram(x, y, z  , c),
                                    histogram(x, y, z+1, c),
                                    histogram(x, y, z+2, c));
    blurx(x, y, z, c) = _tce_wavg5f(blurz(x-2, y, z, c),
                                    blurz(x-1, y, z, c),
                                    blurz(x  , y, z, c),
                                    blurz(x+1, y, z, c),
                                    blurz(x+2, y, z, c));
    blury(x, y, z, c) = _tce_wavg5f(blurx(x, y-2, z, c),
                                    blurx(x, y-1, z, c),
                                    blurx(x, y  , z, c),
                                    blurx(x, y+1, z, c),
                                    blurx(x, y+2, z, c));
```

Linear interpolation

Original 3D linear interpolation implemented using Halide builtin function "lerp" (1D linear interpolation)

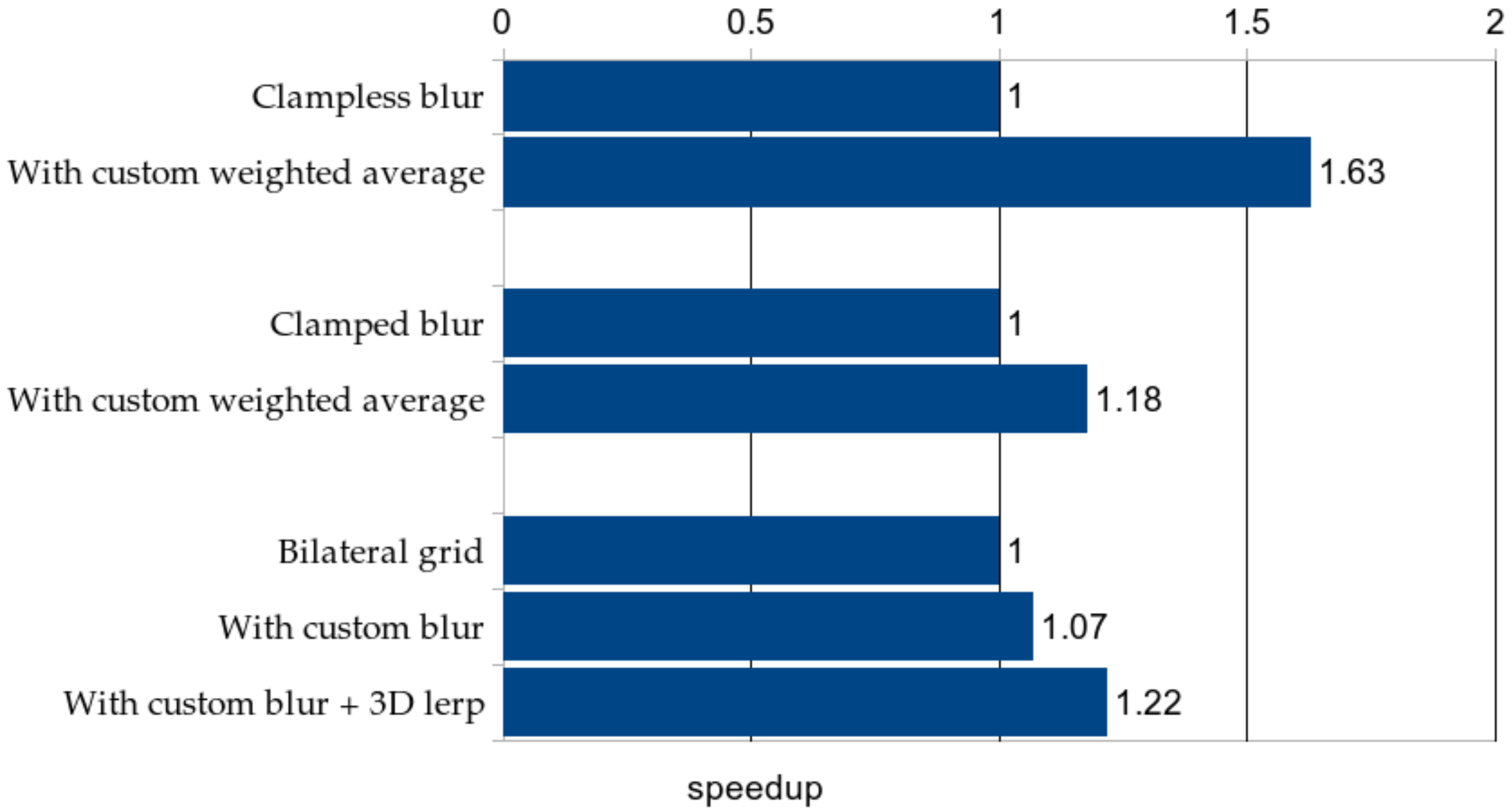
```
Func interpolated("interpolated");
interpolated(x, y, c) =
    lerp(lerp(lerp(blury(xi, yi, zi, c), blury(xi+1, yi, zi, c), xf),
              lerp(blury(xi, yi+1, zi, c), blury(xi+1, yi+1, zi, c), xf), yf),
         lerp(lerp(blury(xi, yi, zi+1, c), blury(xi+1, yi, zi+1, c), xf),
              lerp(blury(xi, yi+1, zi+1, c), blury(xi+1, yi+1, zi+1, c), xf), yf), zf);
```

7 calls to lerp

Replaced with 3D lerp custom operation:

```
HalideExtern_11(float, _tce_lerpf3d, float, float, float, float,
                float, float, float, float, float, float, float);

interpolated(x, y, c) =
  _tce_lerpf3d(blury(xi, yi, zi, c), blury(xi+1, yi, zi, c),
               blury(xi, yi+1, zi, c), blury(xi+1, yi+1, zi, c),
               blury(xi, yi, zi+1, c), blury(xi+1, yi, zi+1, c),
               blury(xi, yi+1, zi+1, c), blury(xi+1, yi+1, zi+1, c),
               xf, yf, zf);
```

Results



- Clamped blur: address computations add overhead
- Bilateral grid: histogram computations dominate

Conclusion

- We described a tool flow from high abstraction level to customized processors
- Custom operations for increasing performance and/or energy efficiency
- Custom operations can be used directly from Halide descriptions
- Tool flow supports multicore custom processors; vector support in progress