Halide

Dr Martin Johnson
School of Mathematical and Computational Sciences
Massey University
Auckland, New Zealand
M.J.Johnson@massey.ac.nz

What is Halide?

A Functional, Domain Specific Language (DSL) designed for Image Processing (but useful for many other things)

Developed at MIT and Google since 2012

- http://halide-lang.org
- Produces highly efficient, parallel, vectorized code
- Used extensively by Google Camera on Pixel phones, most Pixels have specialized Halide hardware (Pixel Visual/Neural Core).

Uses llvm, backends for:

- x86, ARMv7, ARMv8, CUDA, OpenGL Compute Shaders, OpenCL
- Windows, OSX, Linux, Android, iOS

What Makes Halide Different?

- The most significant difference between Halide and other languages is the separation of the processing pipeline into an **Algorithm** and a **Schedule**.
- The **Algorithm** describes the inputs, outputs, and sequence of operations in the pipeline.
  The **Schedule** defines how the pipeline will be mapped onto a particular processing architecture.
- Modifying the schedule is guaranteed not to change the result. This allows you to get the algorithm working and then improve performance.

Why Halide?

- Bounds inference
  - Halide can always work out the size of any intermediate buffers and iterate over them automatically.
- Automatic parallelization
  - Function evaluations can always be computed in parallel and there can never be undetectable race conditions.
- Multiple architectures
  - Use the same algorithm but a different schedule to support mobile, desktop and GPU programming.

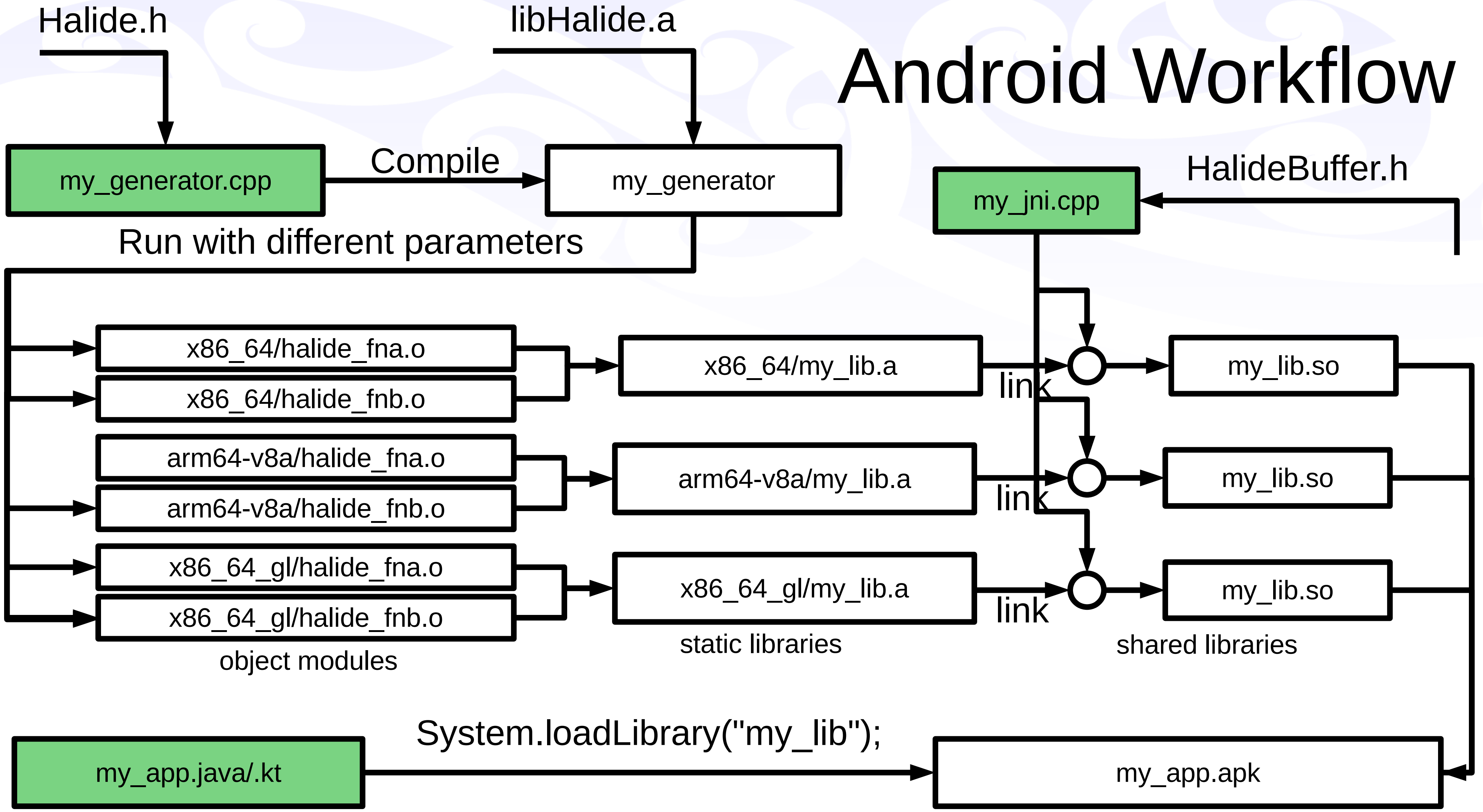
How to write Halide code

- Halide is usually written as a C++ generator, the generator class contains a **generate()** method for the algorithm and a **schedule()** method for the schedule with inputs and outputs defined as member variables.
- This generator can produce object modules for any supported architecture and OS by running it with different parameters e.g.

```
./halide_generator -g halide_bitmap -o arm64-v8a target=arm-64-android -e o,h
./halide_generator -g halide_vel_step -o x86_64 target=x86-64-android -e o,h
./halide_generator -g halide_vel_step -o x86_64_gl target=x86-64-android-openglcompute -e o,h
```

- For Android you will also need some native jni “glue” to call the Halide code and manage Buffers.

Android Workflow



Parts of a Halide Algorithm

Variables – range is determined by bounds inference

```
Var x,y;
```

Expressions

```
Expr lum=77*red+150*green+29*blue;
```

Functions

```
Func sum_x;
sum_x(x,y)=sum(base(x * size + rTile, y))/size;
```

There lots of built in functions, e.g.

sum(), max(), clamp(), select(), mux(), lerp(), argmin()

Reduction Domains – iterate over a specified range

```
RDom rTile(0, size);
```

Tuples – simple data structure indexed with []

```
Tuple t={xmin,ymin};
```

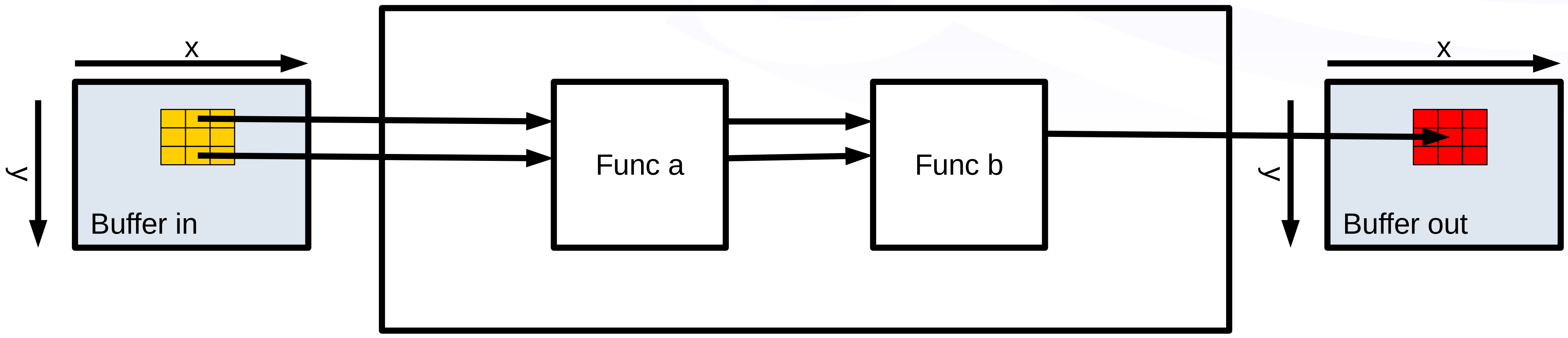
Limitations

- Not a Turing complete language
  - You can only write functions that are guaranteed to terminate
  - No recursion
  - No loops (except over Var bounds or RDoms)
  - No **if** statements
- Use RDom for fixed size iteration
- Use select() or mux() instead of **if**
- Use C++ calling Halide for iteration until a condition is met

Halide schedules

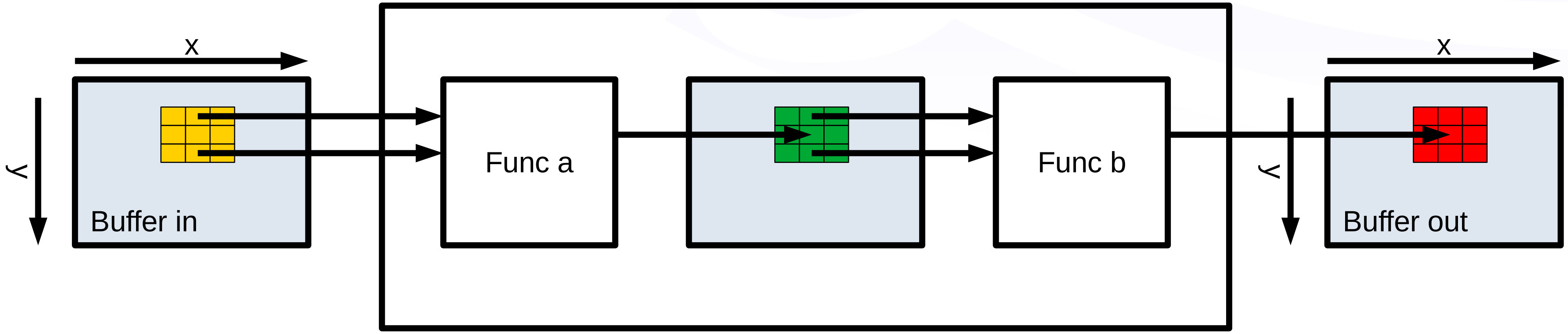
- The schedule specifies the order and manner in which the algorithm is computed using functions such as **parallel()**, **vectorize()**, **unroll()** and **reorder()**.
- The schedule can be used to specify when to compute the value of a function. **compute_root()** will compute and store all values for a function, **compute_at()** and **compute_with()** allow a value to be computed inside a loop for computing another function.
- The default schedule is **compute_inline()**, for all but the most trivial functions, this will not work well!

Halide schedules



Consider two Functions, **a** and **b**
Each uses 2 pixels to get the value for each output pixel
There are many ways to compute **a** followed by **b**.

Halide schedules

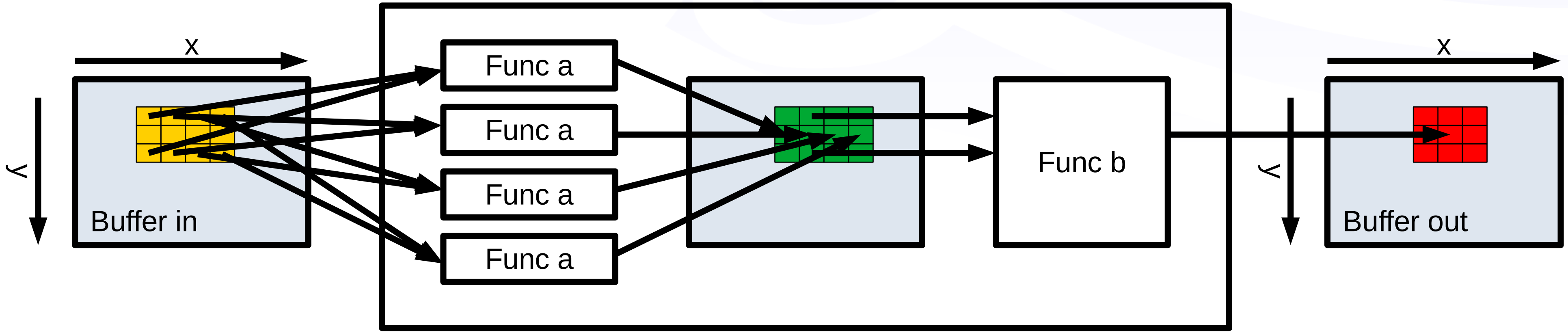


No parallelism

```
a.compute_root();
b.compute_root();
```

Needs a new buffer.

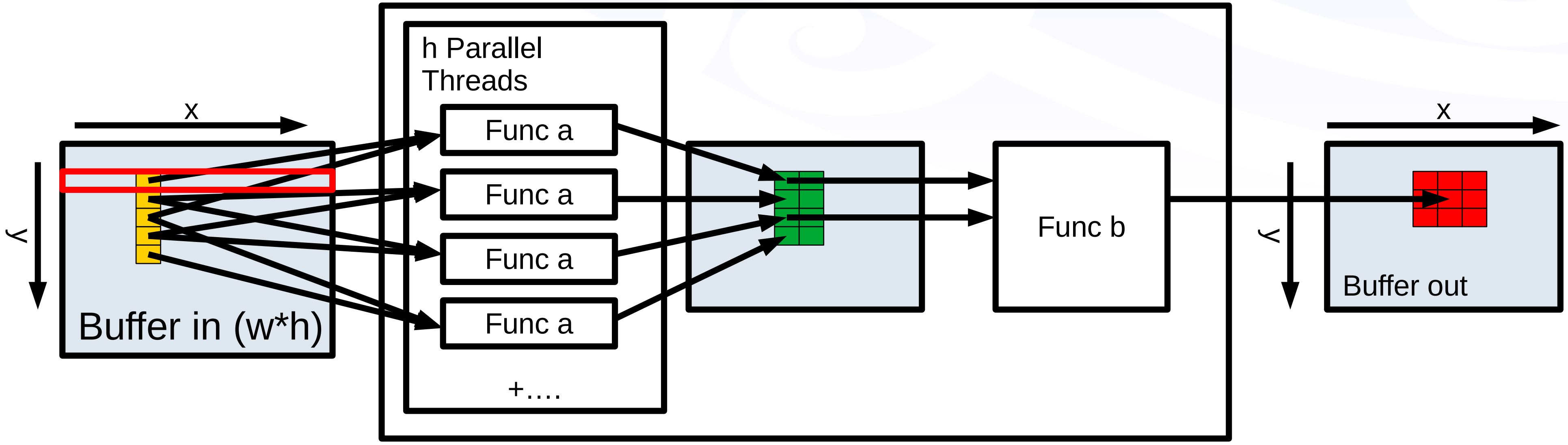
Halide schedules



```
a.compute_root().vectorize(x,4);
b.compute_root();
```

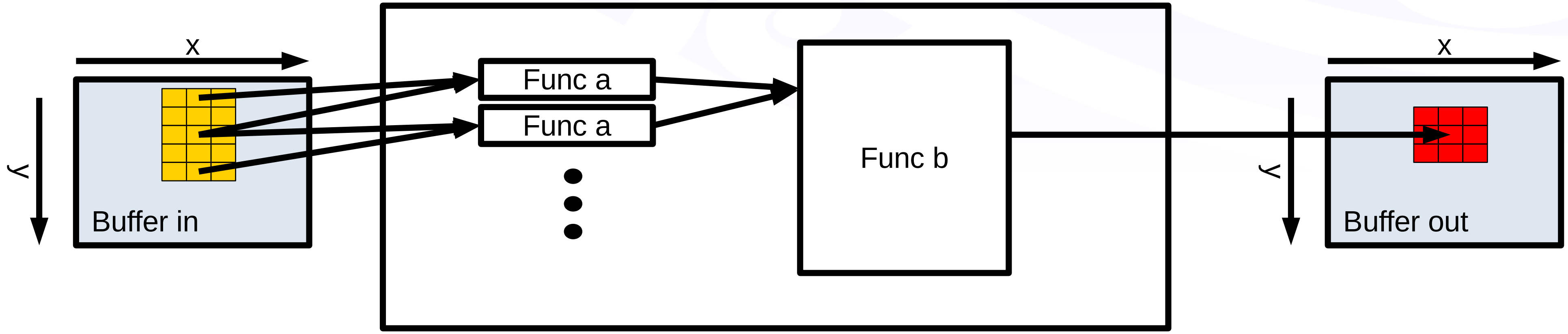
parallelism

Halide schedules



a.compute_root().parallel(y);
b.compute_root();

Halide schedules

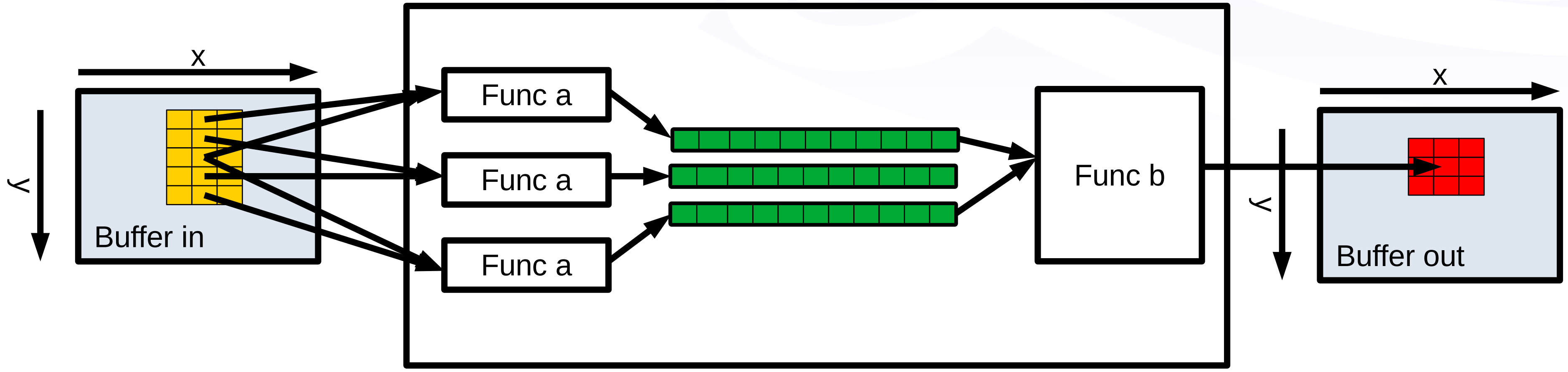


```
a.compute_inline();
b.compute_root();
```

unnecessary because this is the default

Extra evaluations of **a**
But no storage necessary

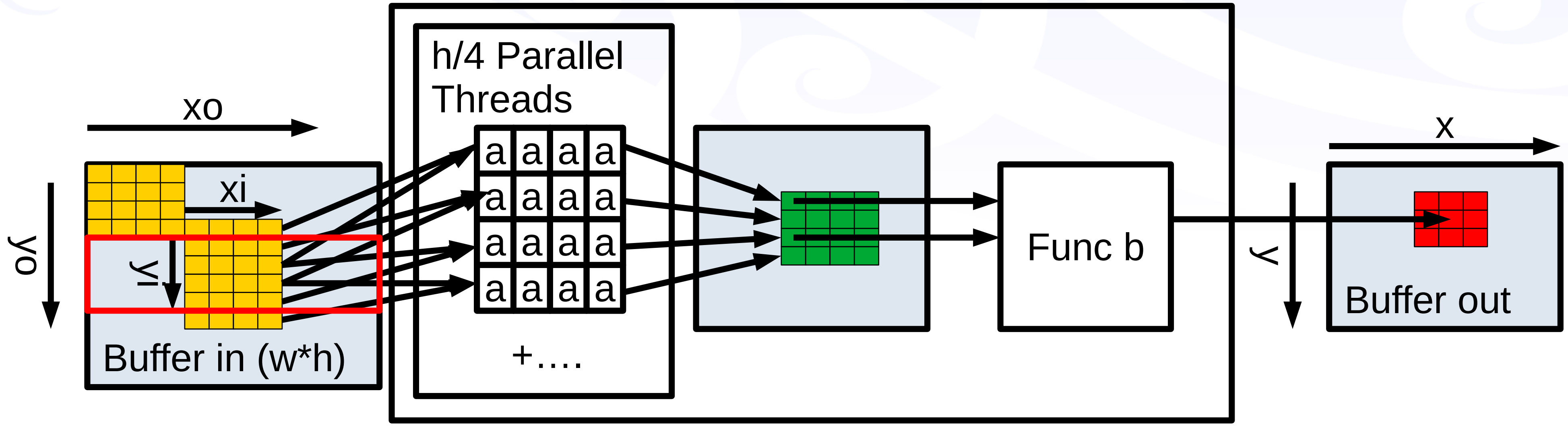
Halide schedules



Needs a 3 line buffer.

a.compute_at(b,y);
b.compute_root();

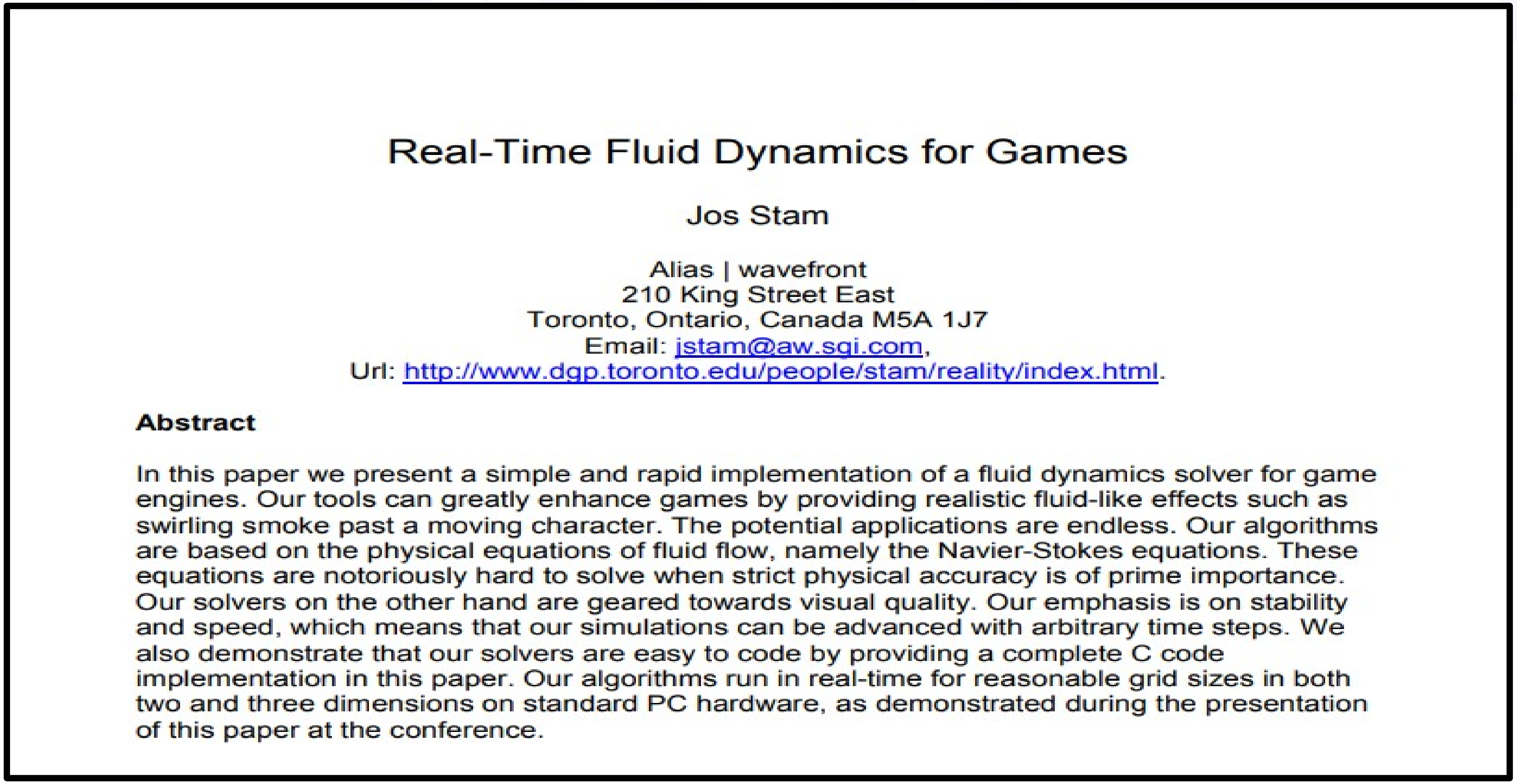
Halide schedules



```
a.compute_root().tile(x, y, xo, yo, xi, yi, 4, 4)
                  .parallel(yo).vectorize(xi);
b.compute_root();
```

Lots of parallelism!

Example – Fluid Simulation Paper (GDC 2003)



Real-Time Fluid Dynamics for Games

Jos Stam

Alias | wavefront
210 King Street East
Toronto, Ontario, Canada M5A 1J7
Email: jstam@aw.sgi.com,
Url: http://www.dgp.toronto.edu/people/stam/reality/index.html.

Abstract

In this paper we present a simple and rapid implementation of a fluid dynamics solver for game engines. Our tools can greatly enhance games by providing realistic fluid-like effects such as swirling smoke past a moving character. The potential applications are endless. Our algorithms are based on the physical equations of fluid flow, namely the Navier-Stokes equations. These equations are notoriously hard to solve when strict physical accuracy is of prime importance. Our solvers on the other hand are geared towards visual quality. Our emphasis is on stability and speed, which means that our simulations can be advanced with arbitrary time steps. We also demonstrate that our solvers are easy to code by providing a complete C code implementation in this paper. Our algorithms run in real-time for reasonable grid sizes in both two and three dimensions on standard PC hardware, as demonstrated during the presentation of this paper at the conference.

Real-Time Fluid Dynamics for Games – Jos Stam 2003

- Used by many games to display fire or smoke.
  - Solves the Navier-Stokes equations for incompressible fluids.
  - Contains a full C implementation of the algorithm.
  - Uses 3 x 2D arrays of floats - density(d), x-velocity(u), y-velocity(v)
- 3 main functions:
  - diffuse() - diffusion of density and velocity
  - advect() - change density and velocity
  - project() - make the velocity field obey the equations
- Called by dens_step() and vel_step()

Android App

- https://github.com/dzo/FluidSimulation
  - MainActivity.java - Runs the simulation in BG thread
  - Simuation.java – Interface to the simulation
  - JavaSimulation.java – Implementation in Java
  - NativeSimulation.java – native declarations
  - SettingsActivity.java – Preference Settings
  - SimulationView.java – Custom View for drawing the simulation
  - jnisimulation.cpp – C Implementation and Halide glue
  - halide_generator.cpp – Generator for Halide code

C Implementation of advect()

```c
void advect(float *d, float *d0, float *u, float *v, float dt) {
    int i, j, i0, j0, i1, j1;
    float x, y, s0, t0, s1, t1, dt0;

    dt0 = dt * NH;
    FOR_EACH_CELL
            x = i - dt0 * u[IX(i, j)];
            y = j - dt0 * v[IX(i, j)];
            if (x < 0.5f) x = 0.5f;
            if (x > NH + 0.5f) x = NH + 0.5f;
            i0 = (int) x;
            i1 = i0 + 1;
            if (y < 0.5f) y = 0.5f;
            if (y > NW + 0.5f) y = NW + 0.5f;
            j0 = (int) y;
            j1 = j0 + 1;
            s1 = x - i0;
            s0 = 1 - s1;
            t1 = y - j0;
            t0 = 1 - t1;
            d[IX(i, j)] = s0 * (t0 * d0[IX(i0, j0)] + t1 * d0[IX(i0, j1)]) +
                          s1 * (t0 * d0[IX(i1, j0)] + t1 * d0[IX(i1, j1)]);
    END_FOR
}
```

Java Implementation of advect()

```
void advect(float[][] d, float[][] d0, float[][] u, float[][] v, float dt) {
    int i0, j0, i1, j1;
    float x, y, s0, t0, s1, t1, dth ;
    dth = dt * height;
    for (int i = 1; i <= height; i++) {
        for (int j = 1; j <= width; j++) {
            x = i - dth * u[i][j];
            y = j - dth * v[i][j];
            if (x < 0.5f) x = 0.5f;
            if (x > height + 0.5f) x = height + 0.5f;
            i0 = (int) x;
            i1 = i0 + 1;
            if (y < 0.5f) y = 0.5f;
            if (y > width+ 0.5f) y = width + 0.5f;
            j0 = (int) y;
            j1 = j0 + 1;
            s1 = x - i0;
            s0 = 1 - s1;
            t1 = y - j0;
            t0 = 1 - t1;
            d[i][j] = s0 * (t0 * d0[i0][j0] + t1 * d0[i0][j1])
                         + s1 * (t0 * d0[i1][j0] + t1 * d0[i1][j1]);
        }
    }
}
```

Halide Implementation of advect()

```
Func advect (Func d0, Func u, Func v, Expr dt, Expr w, Expr h) {
    Func advected{"advected"};

    Expr dt0 = dt*h;
    Expr xx = clamp(x-dt0*v(x,y), 0.5f, w+0.5f);
    Expr yy = clamp(y-dt0*u(x,y), 0.5f, h+0.5f);
    Expr i0=cast<int>(xx);
    Expr j0=cast<int>(yy);
    Expr i1=i0+1;
    Expr j1=j0+1;
    Expr s1 = xx-i0;
    Expr t1 = yy-j0;
    advected(x,y) = lerp(lerp(d0(i0,j0),d0(i0,j1),t1),
                         lerp(d0(i1,j0),d0(i1,j1),t1),s1);
    return advected;
}
```

Halide VelStep Inputs,Outputs and Funcs

```
class VelStepGenerator : public
Halide::Generator<VelStepGenerator> {
public:
    Input <Buffer<float>> u{"u", 2};
    Input <Buffer<float>> v{"v", 2};
    Input <Buffer<float>> u0{"u0", 2};
    Input <Buffer<float>> v0{"v0", 2};
    Input<float> visc{"visc"};
    Input<float> dt{"dt"};
    Output<Buffer<float>> outputu{"outputu", 2};
    Output<Buffer<float>> outputv{"outputv", 2};

    Func uu{"uu"}, vv{"vv"}, diffU{"diffU"}, diffV{"diffV"};
    Func au0{"au0"}, av0{"av0"}, adU{"adU"}, adV{"adV"};
```

Halide generate() for VelStep

```
void generate() {
    Expr w = u.width() - 2;
    Expr h = u.height() - 2;
    uu(x,y) = u(x,y) + u0(x,y) * dt;
    vv(x,y) = v(x,y) + v0(x,y) * dt;
    diffU(x,y) = diffuse(u0, uu, visc, dt, w, h);
    diffV(x,y) = diffuse(v0, vv, visc, dt, w, h);
    project(diffU, diffV, au0, av0, w, h);
    adU(x,y) = advect(au0, au0, av0, dt, w, h);
    adV(x,y) = advect(av0, au0, av0, dt, w, h);
    project(adU, adV, outputu, outputv, w, h);
}
```

Halide schedule() for VelStep

```
void schedule() {
    if (auto_schedule) {
        u.set_estimates({{0, 512},{0, 786}});
        v.set_estimates({{0, 512},{0, 786}});
        u0.set_estimates({{0, 512},{0, 786}});
        v0.set_estimates({{0, 512},{0, 786}});
        outputu.set_estimates({{0, 512},{0, 786}});
        outputv.set_estimates({{0, 512},{0, 786}});
        dt.set_estimate(0.1);
        visc.set_estimate(0.00001);
    } else {
        good_schedule({uu, vv, au0, av0, adU, adV, outputu, outputv});
        if(!gpu) {
            adU.compute_with(adV, x);
            uu.compute_with(vv, x);
            au0.compute_with(av0, x);
            outputu.compute_with(outputv, x);
        }
    }
}
```

good_schedule()

```
void good_schedule(vector <Func> v) {
    if(auto_sch)
        return;
    for(Func f:v) {
        if (gpu)
            f.compute_root().gpu_tile(x,y, xo,yo,xi, yi, 16,16);
        else
            f.compute_root().tile(x, y, xi, yi, 16, 16)
                            .parallel(y).vectorize(xi,16);
    }
}
```

JNI code

```
JNIEXPORT void JNICALL
Java_com_example_martin_simulation_NativeSimulation_dens_1step(JNIEnv *env, jobject
thiz, jfloat diff, jfloat dt) {
    if(halide>1) {
        dens0_h.set_host_dirty();
        halide_dens_step(dens_h, dens0_h, u_h, v_h, diff, dt, dens_h);
    }
    else
        dens_step(width, height, dens, dens_prev, u, v, diff, dt);
}
JNIEXPORT void JNICALL
Java_com_example_martin_simulation_NativeSimulation_vel_1step(JNIEnv *env, jobject
thiz, jfloat visc, jfloat dt) {
   if(halide>1) {
       u0_h.set_host_dirty();
       v0_h.set_host_dirty();
       halide_vel_step(u_h, v_h, u0_h, v0_h, visc, dt, u_h, v_h);
   }
   else
       vel_step(width, height, u, v, u_prev, v_prev, visc, dt);
}
```

Profiler – generate using -profile

```
halide_vel_step
 total time: 688.737427 ms  samples: 651  runs: 373  time/run: 1.846481 ms
 average threads used: 6.978495
 heap allocations: 312574  peak heap usage: 2986112 bytes
  halide_malloc:         0.076ms   (4%)    threads: 10.074
  halide_free:           0.068ms   (3%)    threads: 9.625
  vv:                    0.110ms   (5%)    threads: 3.200  peak: 589824   num: 373       avg: 589824
  uu:                    0.000ms   (0%)    threads: 0.000  peak: 589824   num: 373       avg: 589824
  f2:                    0.090ms   (4%)    threads: 4.718  peak: 595984   num: 373       avg: 595984
  f1:                    0.054ms   (2%)    threads: 10.157 peak: 335232   num: 9325      avg: 27936
  f5:                    0.127ms   (6%)    threads: 4.022  peak: 595984   num: 373       avg: 595984
  f4:                    0.045ms   (2%)    threads: 10.125 peak: 335232   num: 9325      avg: 27936
  div:                   0.116ms   (6%)    threads: 4.926  peak: 589824   num: 373       avg: 589824
  fy:                    0.139ms   (7%)    threads: 5.612  peak: 602176   num: 373       avg: 602176
  fx:                    0.082ms   (4%)    threads: 10.379 peak: 19008    num: 144724    avg: 1584
  av0:                   0.128ms   (6%)    threads: 6.088  peak: 595984   num: 373       avg: 595984
  au0:                   0.000ms   (0%)    threads: 0.000  peak: 595984   num: 373       avg: 595984
  adV:                   0.317ms   (17%)   threads: 8.437  peak: 595984   num: 373       avg: 595984
  adU:                   0.000ms   (0%)    threads: 0.000  peak: 595984   num: 373       avg: 595984
  div$1:                 0.065ms   (3%)    threads: 3.869  peak: 589824   num: 373       avg: 589824
  fy$1:                  0.213ms   (11%)   threads: 6.680  peak: 602176   num: 373       avg: 602176
  fx$1:                  0.093ms   (5%)    threads: 10.696 peak: 19008    num: 144724    avg: 1584
  outputu:               0.000ms   (0%)    threads: 0.000
  outputv:               0.116ms   (6%)    threads: 6.951
```

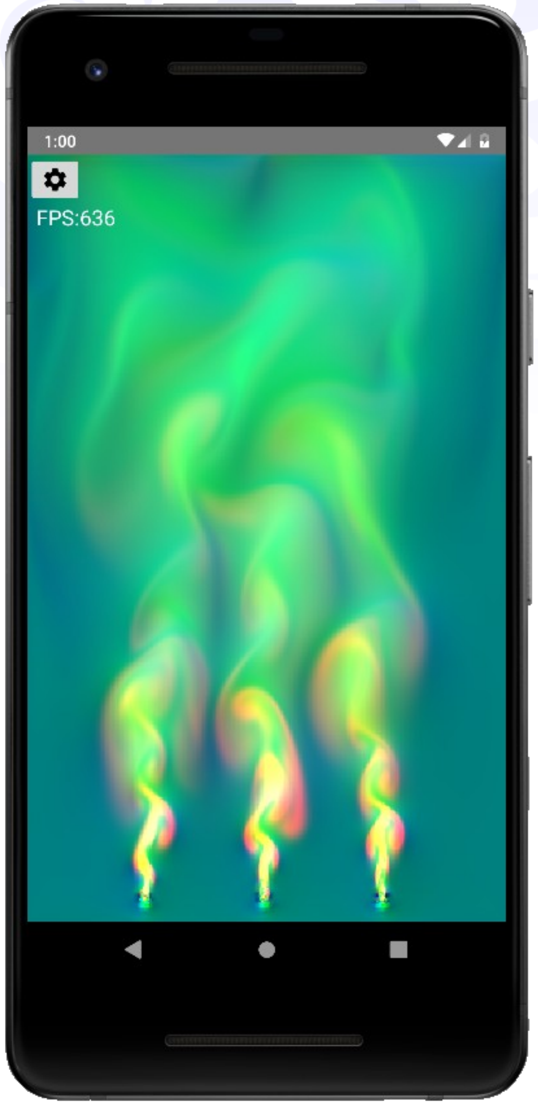
Makefile

```
include Makefile.inc
GENERATOR_SRC=$(HALIDE_SRC_PATH)/tools/GenGen.cpp
GENERATOR_OPTS_RT=-no_asserts-no_bounds_query
SCHEDULE=auto_schedule=false
EMIT=-e o,h,stmt
all:  arm64-v8a/libnavierstokes_halide.a x86_64/libnavierstokes_halide.a \
x86_64_gl/libnavierstokes_halide.a arm64-v8a_gl/libnavierstokes_halide.a \
arm64-v8a_cl/libnavierstokes_halide.a \
define haliderules
$(1)/libnavierstokes_halide.a : $(1)/halide_dens_step.o $(1)/halide_vel_step.o \
$(1)/halide_bitmap.o
    ar r  $(1)/libnavierstokes_halide.a $$^
$(1)/%.o: halide_generator
    ./$$< -g $$(*F) -o $(1) target=$(2)$(GENERATOR_OPTS_RT) $(EMIT) $(SCHEDULE)
endef
.SECONDARY: halide_generator
$(eval $(call haliderules,arm64-v8a,arm-64-android))
$(eval $(call haliderules,arm64-v8a_gl,arm-64-android-openglcompute))
$(eval $(call haliderules,arm64-v8a_cl,arm-64-android-opencl))
$(eval $(call haliderules,x86_64,x86-64-android))
$(eval $(call haliderules,x86_64_gl,x86-64-android-openglcompute))

%:: %.cpp
    $(CXX) $(CXXFLAGS) $< $(GENERATOR_SRC) -g $(LIB_HALIDE) -o $@ -lpthread \
           -fno-rtti -ldl -lz -lncurses -rdynamic -O3 $(LDFLAGS) $(LLVM_SHARED_LIBS)
```

Demo

- Android Emulator on Linux
- 4 core (of 6) Ryzen 3600 4.2GHz
- 270x432 grid



Android Performance

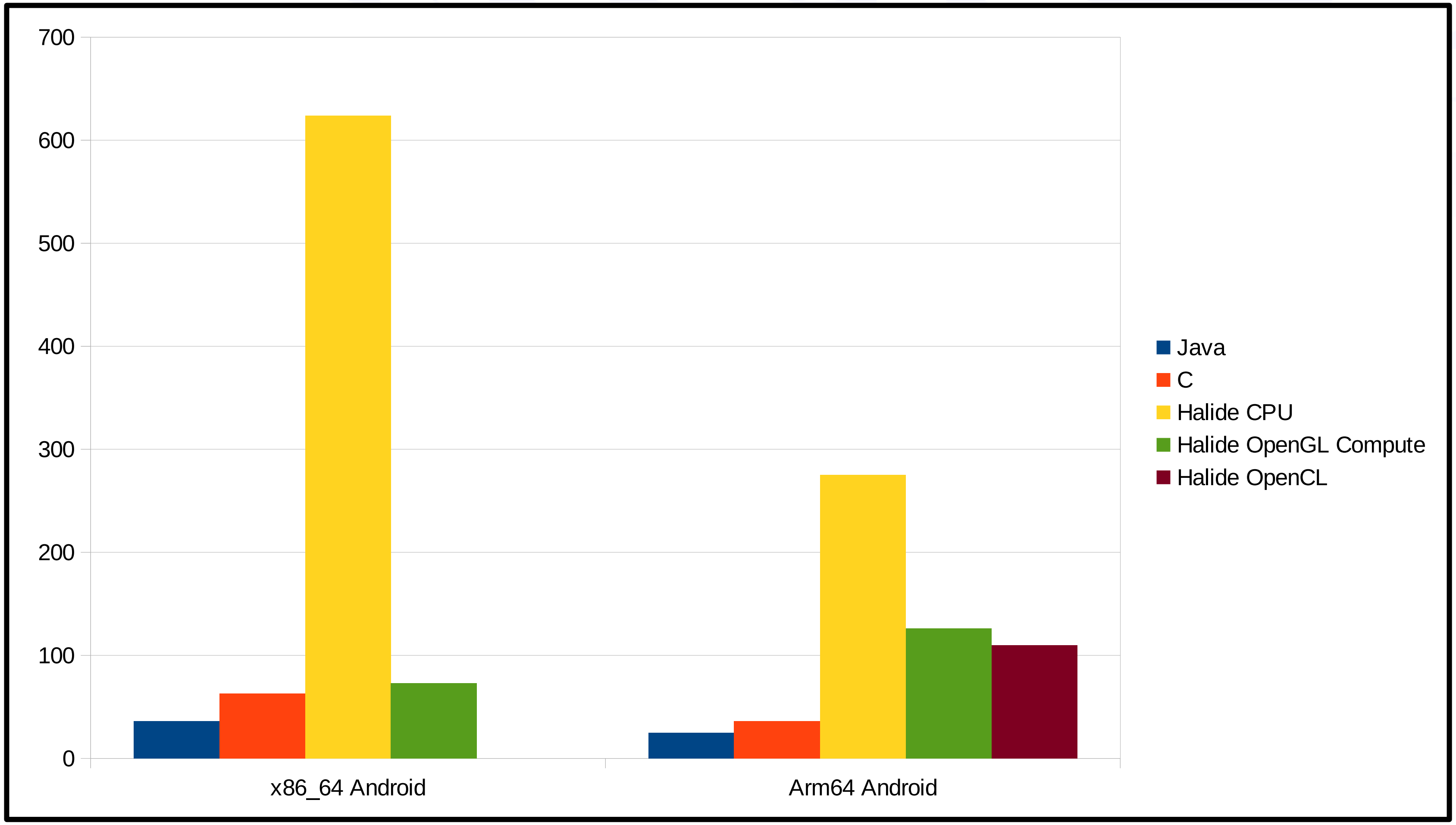
- x86_64 (emulator - 4 core Ryzen 3600 4.2GHz) (270x432 grid)

– Java:	36fps
– C:	63fps
– Halide CPU:	624fps (250fps with autoschedule)
– Halide OpenGL:	73fps (not reliable)

- ARM64 (Snapdragon 870 – 8 core 1x3.2GHz, 3x2.4GHz, 4x1.8GHz) (270x547 grid)

– Java	25fps
– C:	36fps
– Halide CPU:	275fps (125fps with autoschedule)
– Halide OpenGL:	126fps
– Halide OpenCL:	110fps (fluctuates a lot)

Android Performance



Desktop Performance

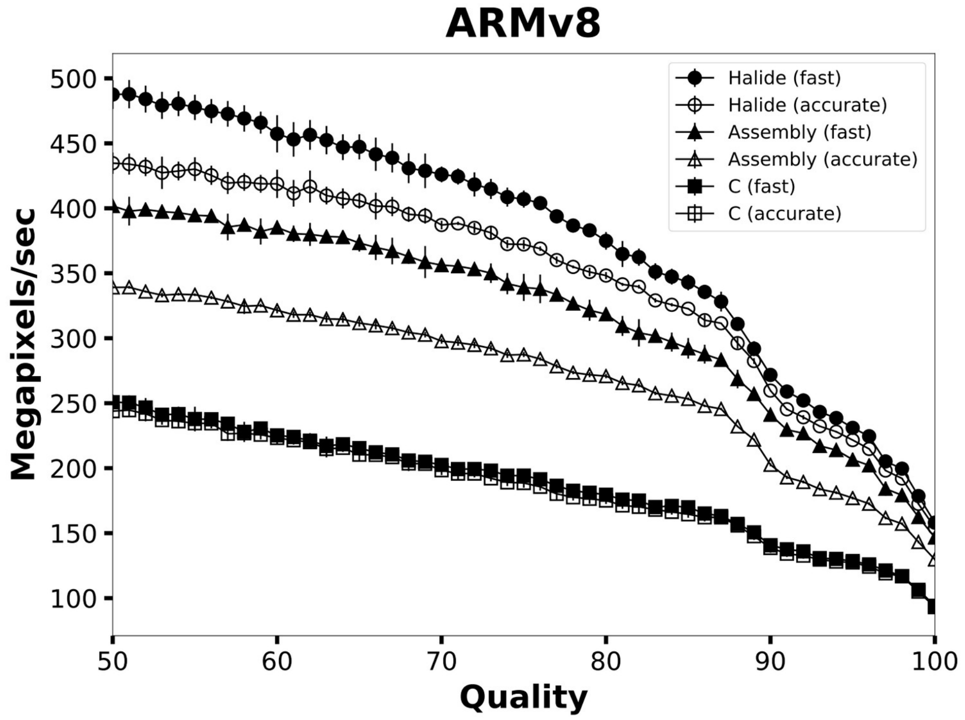
- 6 core (12 thread) Ryzen 3600 4.2GHz, 384x384 grid, GTX1650Super GPU

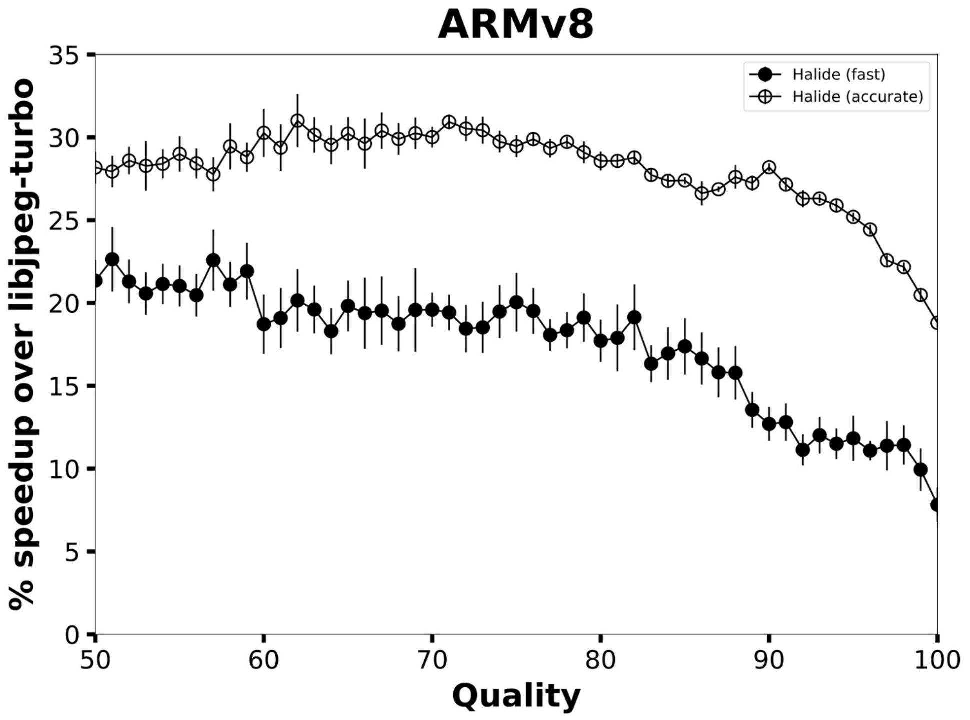
– C	41fps
– Halide CPU	466fps
– Halide OpenGL Compute	1706fps
– Halide OpenCL	788fps
– Halide Cuda	2155fps

Power Consumption (ARM64 Android)

	Fps	Current Consumption	Fps multiple	Current multiple
Java	25fps	180mA	1.0	1.0
C	36fps	210mA	1.44	1.16
Halide CPU	275fps	780mA	11	4.33
Halide OpenGL	126fps	240mA	5.04	1.33
Halide OpenCL	110fps	300mA	4.4	1.66

Another Example: JPEG Decoding





Conclusions

- C++ is not significantly faster than Java/Kotlin with ART
- Halide can give you an order of magnitude performance increase over Java/C++
- Writing Halide is easy and fun!
- Halide CPU gives best mobile performance, but uses lots of power
- Halide with OpenGL gives good Performance and Power Consumption