ArrayFire Graphics

A tutorial

by Chris McClanahan, GPU Engineer

Overview

- Introduction to ArrayFire
- Introduction to ArrayFire Graphics
- Optical Flow
- Optical Flow GFX Example / Demo

ArrayFire Introduction

Matrix Types

f64
real double precision

f32
real single precision

array
container type

b8
boolean byte

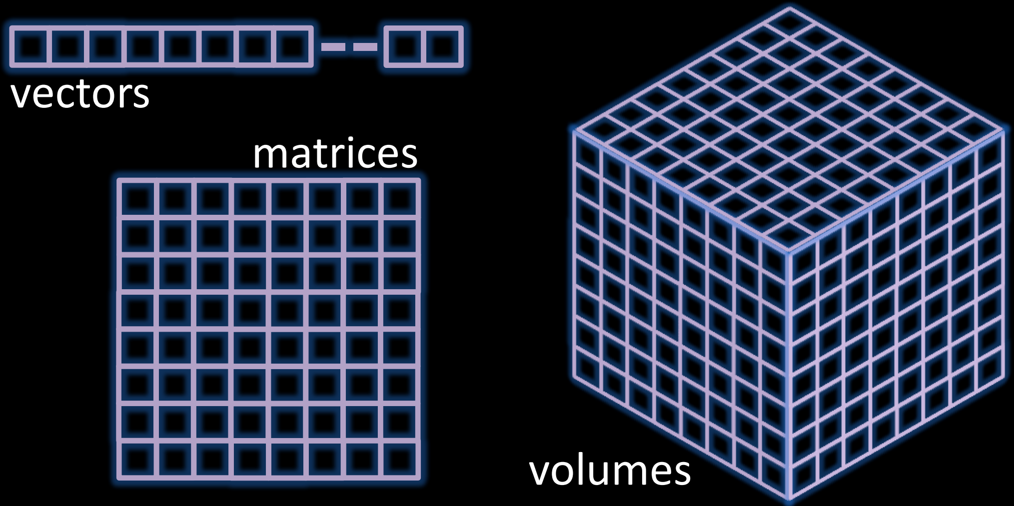
c32
complex single precision

c64
complex double precision

Matrix Types: ND Support

f64
real double precision

f32
real single precision



b8
boolean byte

... ND

c32
complex single precision

c64
complex double precision

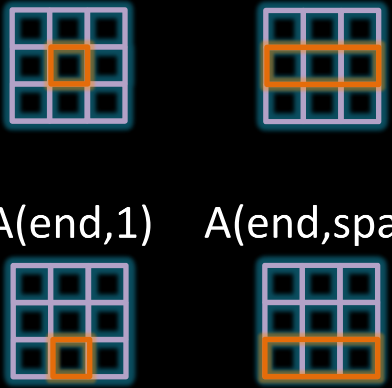
Matrix Types: Easy Manipulation

f64
real double precision

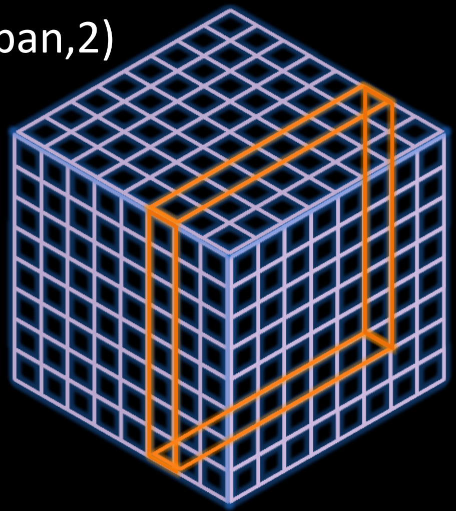
ArrayFire Keywords: **end, span**

A(1,1) A(1,span)

A(end,1) A(end,span)



A(span,span,2)



f32
real single precision

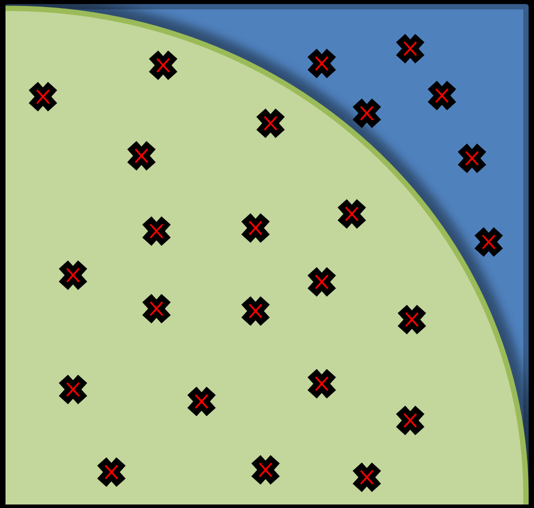
b8
boolean byte

c32
complex single precision

c64
complex double precision

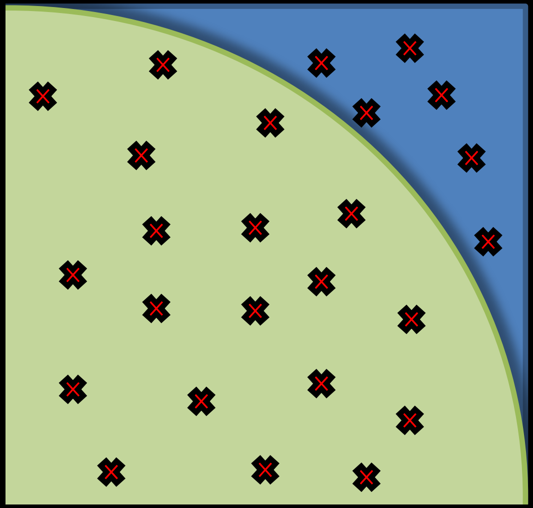
Easy GPU Acceleration in C++

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // 20 million random samples
    int n = 20e6;
    array x = randu(n,1), y = randu(n,1);
    // how many fell inside unit circle?
    float pi = 4 * sum<float>(sqrt(mul(x,x)+mul(y,y))<1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```



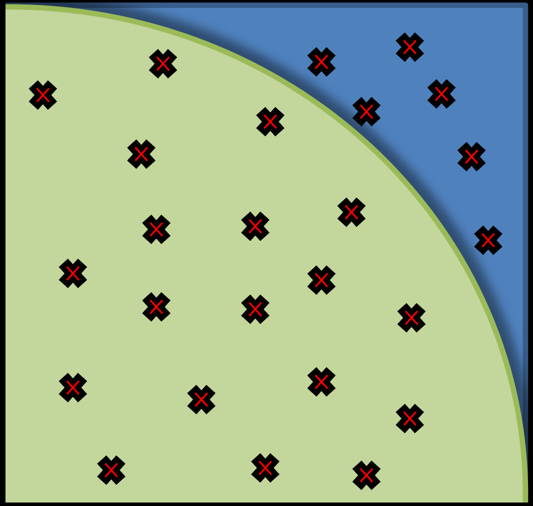
Easy GPU Acceleration in C++

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // 20 million random samples
    int n = 20e6;
    array x = randu(n,1), y = randu(n,1);
    // how many fell inside unit circle?
    float pi = 4 * sum<float>(sqrt(mul(x,x)+mul(y,y))<1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```



Easy GPU Acceleration in C++

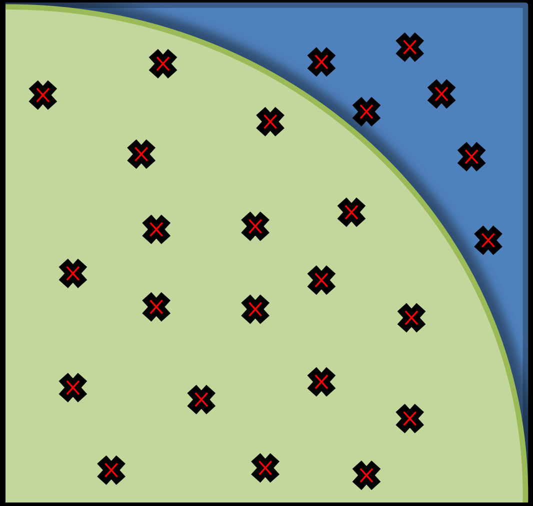
```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // 20 million random samples
    int n = 20e6;
    array x = randu(n,1), y = randu(n,1);
    // how many fell inside unit circle?
    float pi = 4 * sum<float>(sqrt(mul(x,x)+mul(y,y))<1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```



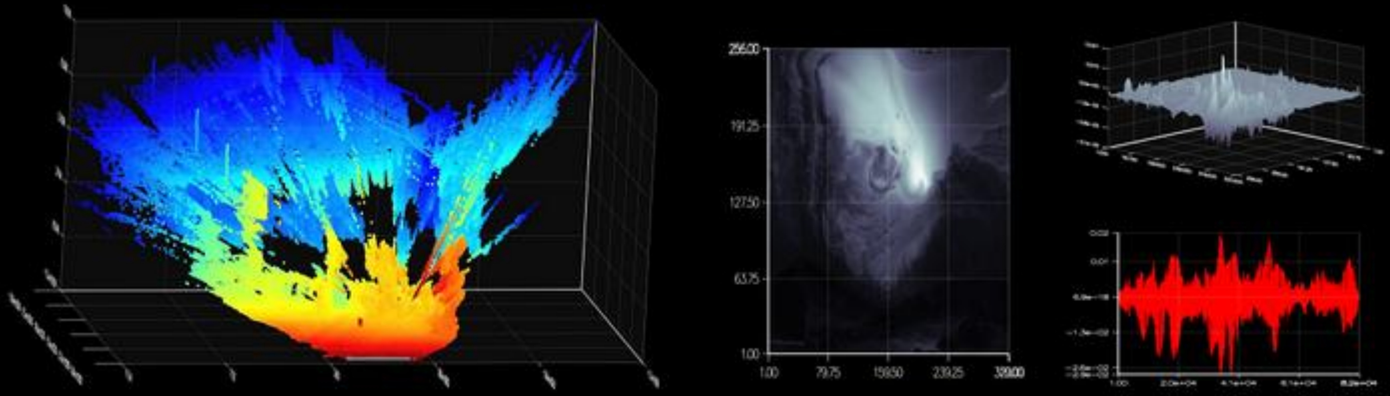
On GPU

Easy GPU Acceleration in C++

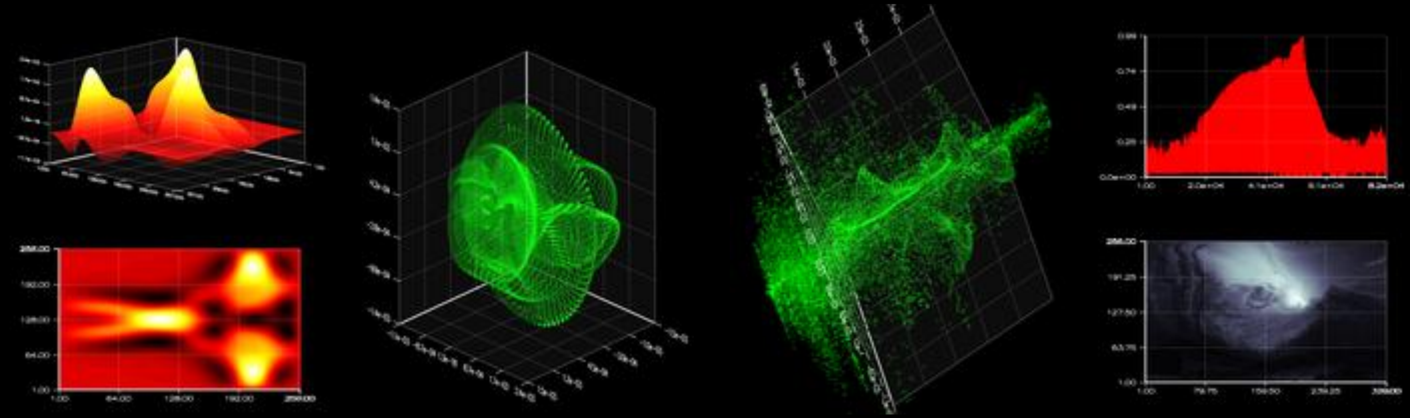
```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // 20 million random samples
    int n = 20e6;
    array x = randu(n,1), y = randu(n,1);
    // how many fell inside unit circle?
    float pi = 4 * sum<float>(sqrt(mul(x,x)+mul(y,y))<1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```



On GPU



ArrayFire Graphics Introduction

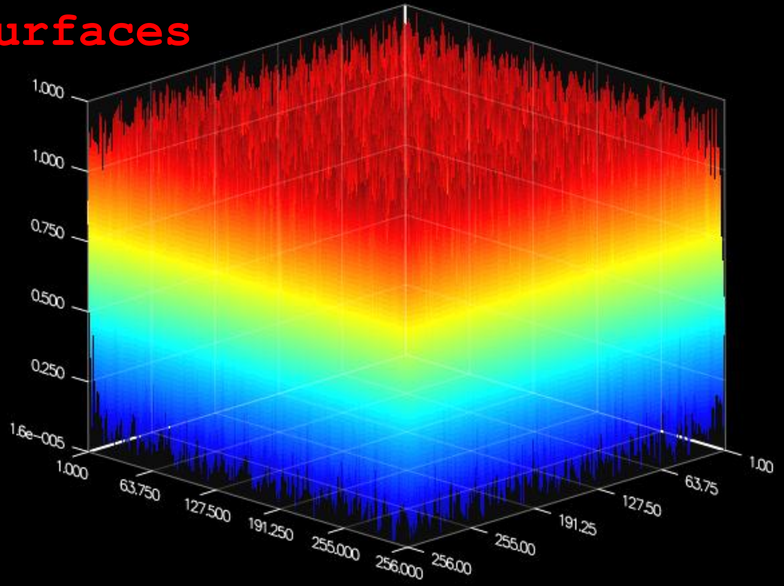


Coupled Compute and GFX

- Graphics were designed to be as unobtrusive as possible on GPU compute
  - Graphics rendering completed in separate worker thread
  - Most graphics commands from compute thread non-blocking and lazy
  - Graphics commands designed to be as simplistic as possible

ArrayFire Graphics Example

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // Infinite number of random 3d surfaces
    const int n = 256;
    while (1) {
        array x = randu(n,n);
        // 3d surface plot
        plot3d(x);
    }
    return 0;
}
```



DEMO

ArrayFire Graphics Example

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // Infinite number of random 3d surfaces
    const int n = 256;
    while (1) {
        array x = randu(n,n);
        // 3d surface plot
        plot3d(x);
    }
    return 0;
}
```

GPU generates random numbers in **compute thread**

ArrayFire Graphics Example

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // Infinite number of random 3d surfaces
    const int n = 256;
    while (1) {
        array x = randu(n,n);
        // 3d surface plot
        plot3d(x);
    }
    return 0;
}
```

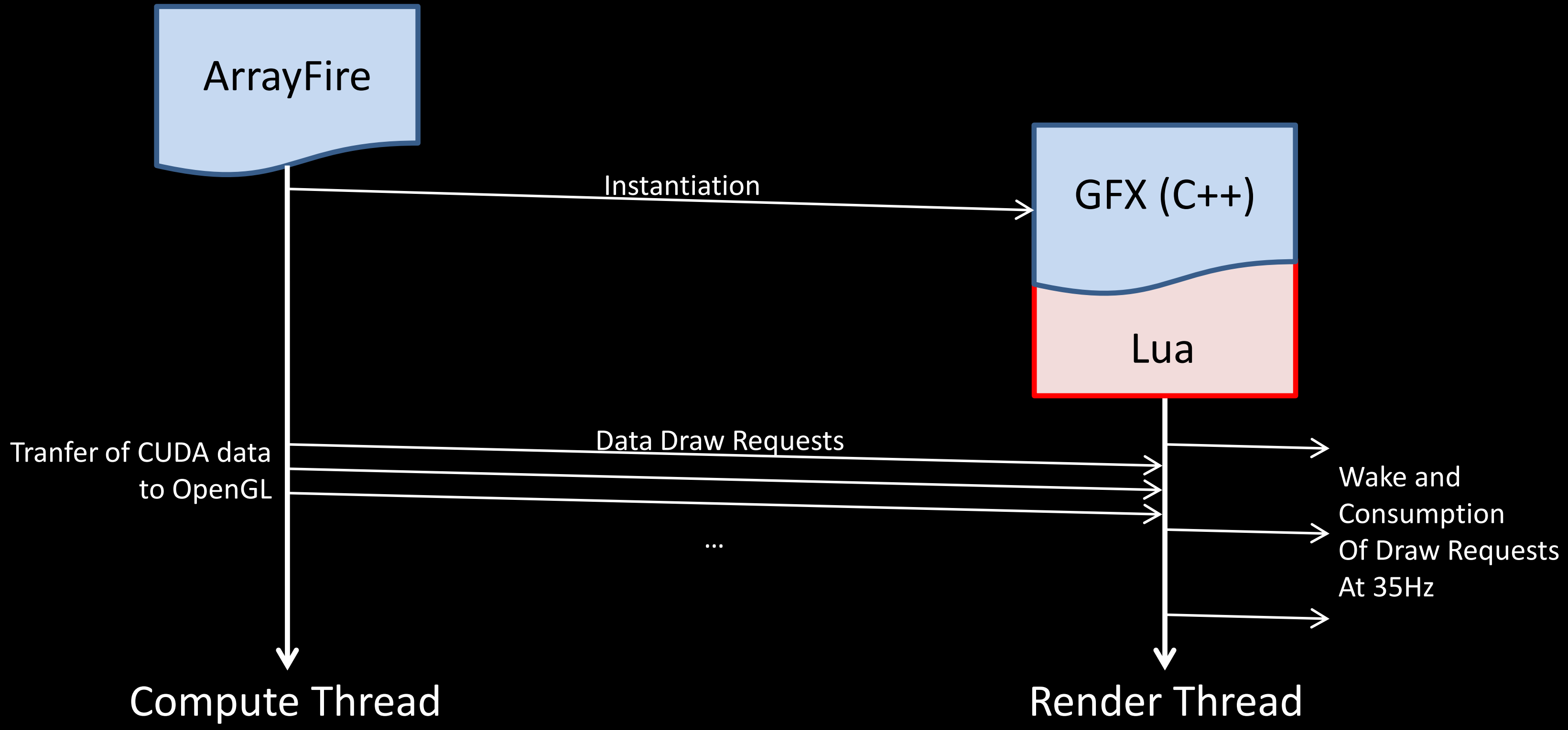
Data from x is transferred to OpenGL and drawing is queued in newly created **render thread**

ArrayFire Graphics Example

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // Infinite number of random 3d surfaces
    const int n = 256;
    while (1) {
        array x = randu(n,n);
        // 3d surface plot
        plot3d(x);
    }
    return 0;
}
```

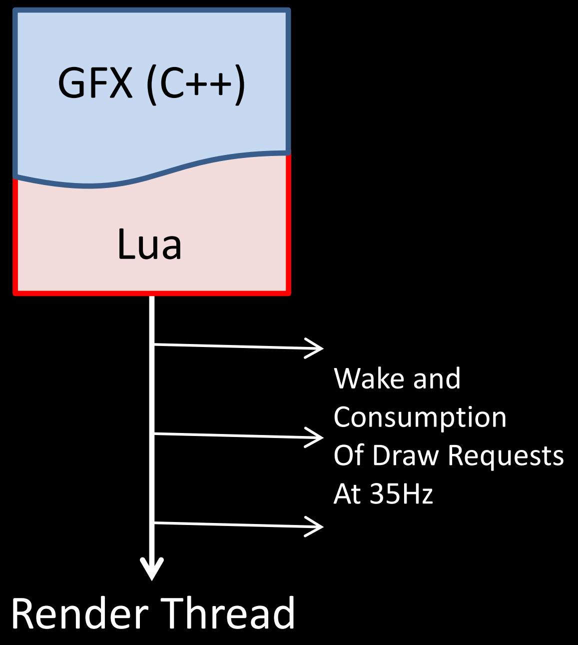
Drawing done in **render thread** at 35Hz; some data dropped

ArrayFire Graphics Implementation



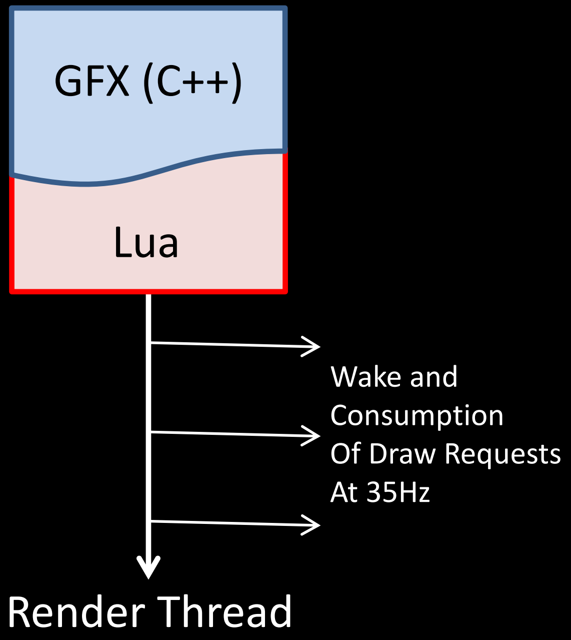
ArrayFire Graphics Implementation

- Render thread hybrid C++/Lua
  - C++
    - Wake / Sleep Loop
    - OpenGL Memory Management
  - Lua
    - Port of OpenGL API to Lua (independent AccelerEyes effort)
    - All graphics primitives / drawing logic



ArrayFire Graphics Implementation

- Lua interface to be opened up at a later date to end users
  - Custom Graphics Primitives
  - Modification of Existing GFX system
  - Easily couple visualization with compute in a platform independent environment



Graphics Commands

- Available primitives (non-blocking)
  - `plot3d`: 3d surface plotting (2d data)
  - `plot`: 2d line plotting
  - `imgplot`: intensity image visualization
  - `arrows`: quiver plot for vector fields
  - `points`: scatter plot
  - `volume`: volume rendering for 3d data
  - `rgbplot`: color image visualization

DEMO

Graphics Commands

- Utility Commands (blocking unless otherwise stated)
  - `keep_on / keep_off`
  - `subfigure`
  - `palette`
  - `clearfig`
  - `draw (blocking)`
  - `figure`
  - `title`
  - `close`

DEMO

Optical Flow (Horn-Schunck)

Horn-Schunck Background

- Compute apparent motion between two images
- Minimize the following functional,

$$E = \iint \left[(I_x u + I_y v + I_t)^2 + \alpha^2(\|\nabla u\|^2 + \|\nabla v\|^2)\right] \mathrm{d}x\mathrm{d}y$$

- By solving,

$$\frac{\partial L}{\partial u} - \frac{\partial}{\partial x}\frac{\partial L}{\partial u_x} - \frac{\partial}{\partial y}\frac{\partial L}{\partial u_y} = 0 \qquad \frac{\partial L}{\partial v} - \frac{\partial}{\partial x}\frac{\partial L}{\partial v_x} - \frac{\partial}{\partial y}\frac{\partial L}{\partial v_y} = 0$$

Horn-Schunck Background

- Can be accomplished with ArrayFire
  - Relatively small amount of code
  - Iterative Implementation on GPU via gradient descent
  - Easily couple compute with visualization via basic graphics commands

ArrayFire Implementation

- Main algorithm loop:

```
array u = zeros(I1.dims()), v = zeros(I1.dims());
while (true) {
        iter++;
        array u_ = convolve(u, avg_kernel, afConvSame);
        array v_ = convolve(v, avg_kernel, afConvSame);

        const float alphasq = 0.1f;
        array num = mul(Ix, u_) + mul(Iy, v_) + It;
        array den = alphasq + mul(Ix, Ix) + mul(Iy, Iy);

        array rsc = 0.01f *  num;
        u = u_ - mul(Ix, rsc) / den;
        v = v_ - mul(Iy, rsc) / den;
}
```

ArrayFire Implementation

- Main algorithm loop:

```
array u = zeros(I1.dims()), v = zeros(I1.dims());
while (true) {
        iter++;
        array u_ = convolve(u, avg_kernel, afConvSame);
        array v_ = convolve(v, avg_kernel, afConvSame);

        const float alphasq = 0.1f;
        array num = mul(Ix, u_) + mul(Iy, v_) + It;
        array den = alphasq + mul(Ix, Ix) + mul(Iy, Iy);

        array rsc = 0.01f *  num;
        u = u_ - mul(Ix, rsc) / den;
        v = v_ - mul(Iy, rsc) / den;
}
```

Initial flow field we are solving for (on GPU)

ArrayFire Implementation

- Main algorithm loop:

```
array u = zeros(I1.dims()), v = zeros(I1.dims());
while (true) {
        iter++;
        array u_ = convolve(u, avg_kernel, afConvSame);
        array v_ = convolve(v, avg_kernel, afConvSame);

        const float alphasq = 0.1f;
        array num = mul(Ix, u_) + mul(Iy, v_) + It;
        array den = alphasq + mul(Ix, Ix) + mul(Iy, Iy);

        array rsc = 0.01f *  num;
        u = u_ - mul(Ix, rsc) / den;
        v = v_ - mul(Iy, rsc) / den;
}
```

Blur current field (on GPU)

ArrayFire Implementation

- Main algorithm loop:

```
array u = zeros(I1.dims()), v = zeros(I1.dims());
while (true) {
        iter++;
        array u_ = convolve(u, avg_kernel, afConvSame);
        array v_ = convolve(v, avg_kernel, afConvSame);

        const float alphasq = 0.1f;
        array num = mul(Ix, u_) + mul(Iy, v_) + It;
        array den = alphasq + mul(Ix, Ix) + mul(Iy, Iy);

        array rsc = 0.01f *  num;
        u = u_ - mul(Ix, rsc) / den;
        v = v_ - mul(Iy, rsc) / den;
}
```

Direction for gradient descent (on GPU). `Ix`, `Iy`, `It` are image derivatives computed by `diffs()` routine. (See code)

ArrayFire Implementation

- Main algorithm loop:

```
array u = zeros(I1.dims()), v = zeros(I1.dims());
while (true) {
        iter++;
        array u_ = convolve(u, avg_kernel, afConvSame);
        array v_ = convolve(v, avg_kernel, afConvSame);

        const float alphasq = 0.1f;
        array num = mul(Ix, u_) + mul(Iy, v_) + It;
        array den = alphasq + mul(Ix, Ix) + mul(Iy, Iy);

        array rsc = 0.01f *  num;
        u = u_ - mul(Ix, rsc) / den;
        v = v_ - mul(Iy, rsc) / den;
}
```

Scale and apply gradient to field iteratively (on GPU)

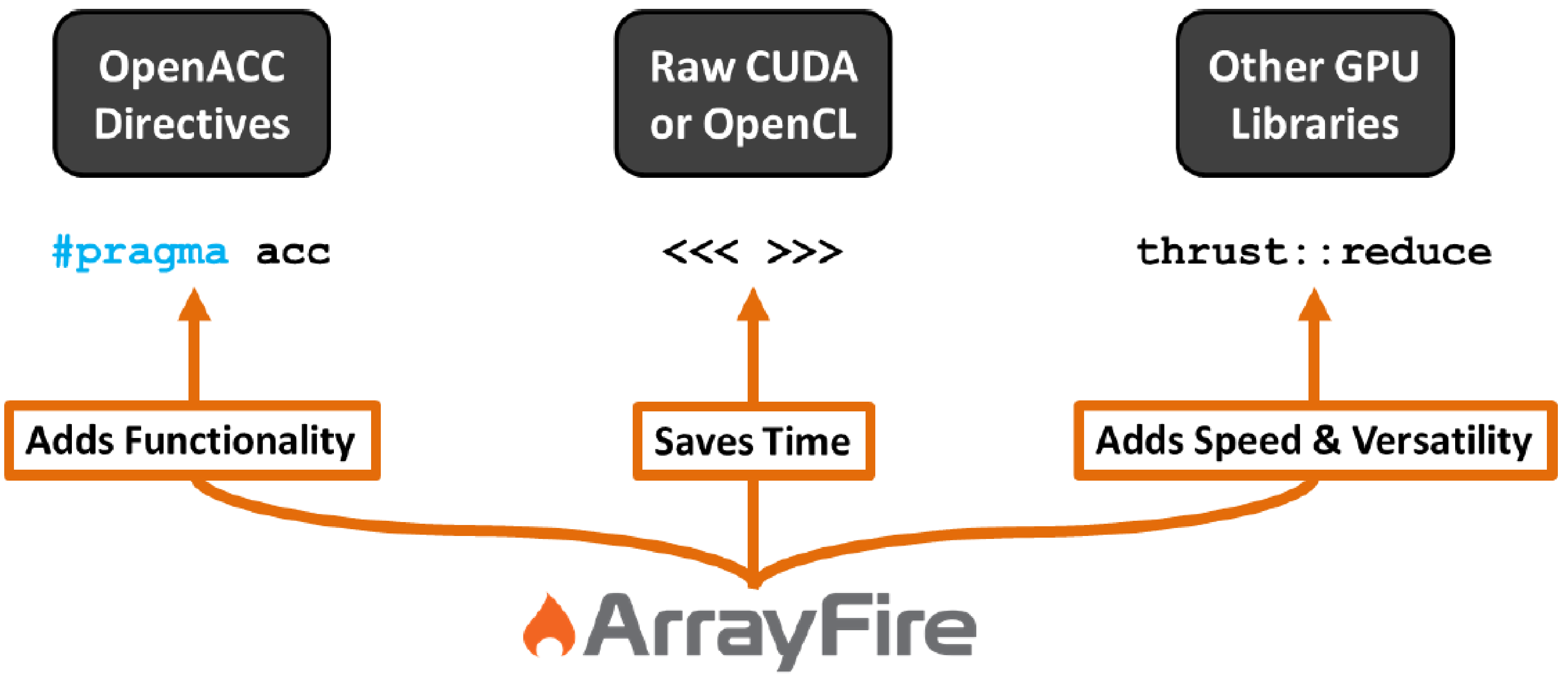
Graphics Integration

```
array u = zeros(I1.dims()), v = zeros(I1.dims());
while (true) {
        iter++;
        array u_ = convolve(u, avg_kernel, afConvSame);
        array v_ = convolve(v, avg_kernel, afConvSame);

        const float alphasq = 0.1f;
        array num = mul(Ix, u_) + mul(Iy, v_) + It;
        array den = alphasq + mul(Ix, Ix) + mul(Iy, Iy);

        array rsc = 0.01f *  num;
        u = u_ - mul(Ix, rsc) / den;
        v = v_ - mul(Iy, rsc) / den;

        subfigure(2,2,1); imgplot(I1);
        subfigure(2,2,3); imgplot(I2);
        subfigure(2,2,2); imgplot(u);
        subfigure(2,2,4); imgplot(v);
}
```

DEMO

Create figures, and draw

ArrayFire + Other GPU Code



ArrayFire

ACCELEREYES

Thank You