Adobe Image Foundation & Adobe PixelBender

Our use of LLVM

Charles F. Rose, III
1. August 2008

Motivation: GPU proliferation

- PC graphics cards were fixed function
  - Texture, lighting, transformation, depth, etc.
- Programmable GPUs took over
  - Tiny asm-like per-pixel programs
  - Per-vector programs
  - High level shading languages (GLSL, HLSL, CG)
  - Multi-pass frameworks (e.g. Effects)
  - GPGPU: CUDA, CTM, OpenCL, DX Compute Shader

Could we use GPUs to do image/video/etc. processing?

Games and beyond

- Games drove PC graphics card development
  - PC gaming made the GPU a successful product and drove innovation at a furious pace for the last decade
- Programmability gave GPUs wider use

- Adobe Image Foundation is a framework for performing data parallel image processing using all available computational resources.
- Adobe PixelBender is a language for writing hybrid GPU/CPU image processing algorithms.

Let’s have a look at PixelBender….

Identity filter

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}

// “sample” is a bilinear interpolation of the texture “src”,
// an operation performed in hardware on all modern GPUs
```

Identity filter

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}
```

Identity filter

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}
```

Identity filter

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}
```

Identity filter

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}
```

Identity filter

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}
```

Identity filter recap

- Programs are written to produce pixels
- Inputs and outputs are globals
- Lots of vector operations going on
- At first glance, it looks a lot like GLSL, but…
  - Has many additions tuned towards image processing
  - Has the concept of things which occur per-frame vs. those which occur per-pixel
  - The entire kernel lives in the PixelBender program, including things which don't run on the GPU
  - Kernel + setup all done in PixelBender program

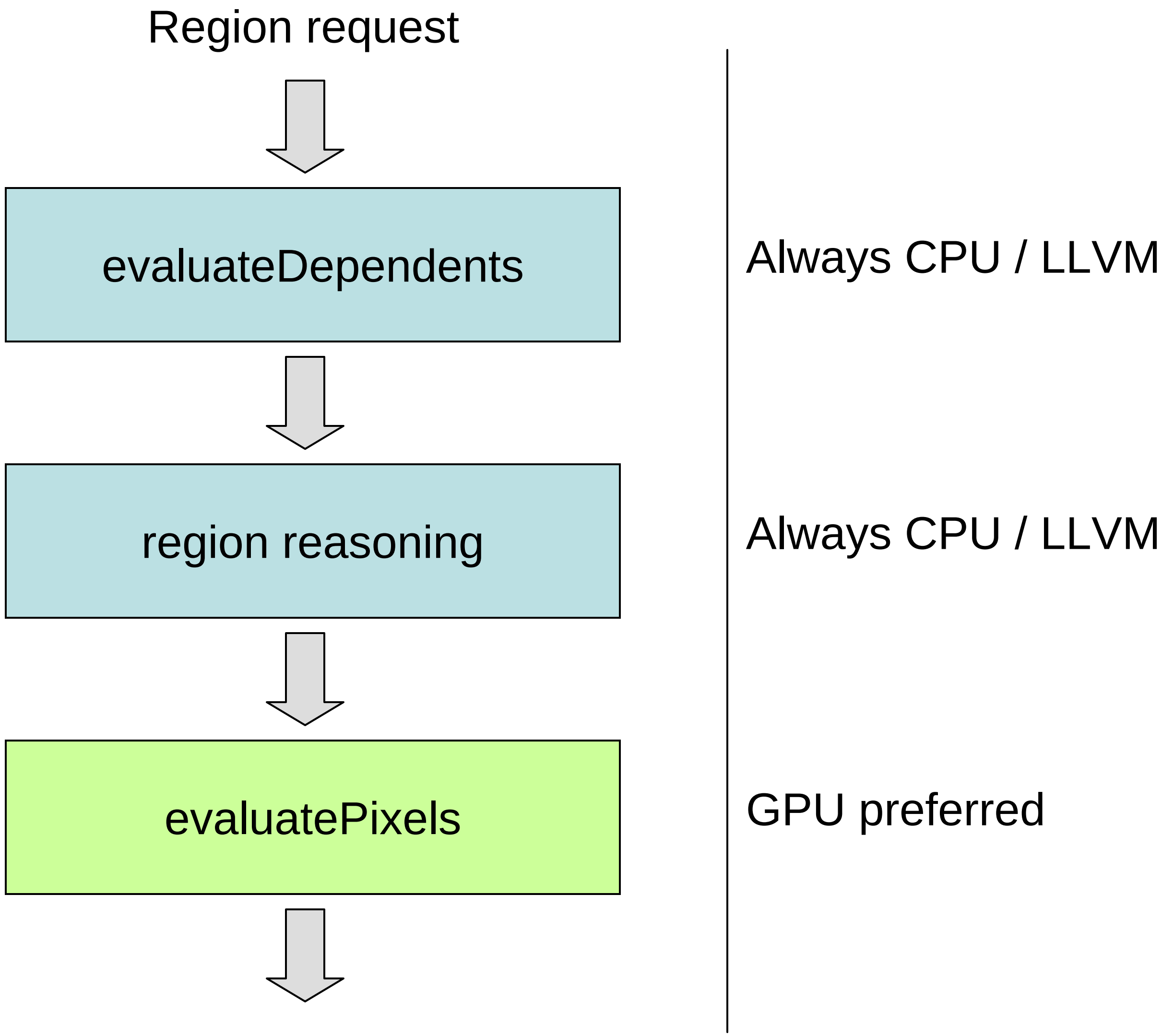
Negative filter

```
kernel Negative
{
    input image4 src;  output pixel4 dst;
    parameter bool isNegative;
    dependent float s;

    void evaluateDependents()
    {  // this happens once per frame
        s = isNegative ? 1.0 : 0.0;
    }

    void evaluatePixel()
    {  // this happens once per pixel
        float4 tmp = sample(src, outCoord());

        // a curious way to write dst = isNegative ? -tmp : tmp;

        dst = s * (1.0 - p) + (1.0 - s) * p;
        dst.a = tmp.a; // leave alpha alone
    }
}
```

Per-frame functions

- **evaluateDependents**
  - General purpose function which sets all “dependent” variables using the parameters.
- Region reasoning / **needed** & **changed**
  - The needed and changed functions are used to calculate how much of an image is needed in order to produce a desired output region.
  - This is particularly useful when kernels are chained together in a series.
- **generated**
  - How much of the output image is produced by this kernel. This is particularly useful for kernels which take no input, such as a Mandelbrot set generator.

Order of macro operations for a single kernel



When do we use LLVM for evaluatePixel?

- Old graphics card
  - Loops, branches, break/continue, int, & bool are only on newer cards
  - Instruction count limits low on older cards
- Card with wonky driver
  - Bad drivers seen on most graphics cards, from all vendors, and on both Mac and PC
- Higher numerical accuracy needed
  - CPU still sets the standard
  - Final render
- Consistency required
  - Want all frames to have the same floating point behavior in a video stream, for example
  - Parameter changes can throw a shader off the card because of instruction count limits.

Adobe

Adobe PixelBender details

PixelBender in detail – Types

float
int
bool
pixel1

int2	float2	bool2	pixel2
int3	float3	bool3	pixel3
int4	float4	bool4	pixel4

float2x2
float3x3
float4x4

PixelBender in detail – Types

float

int

bool

pixel1

int2	float2	bool2	pixel2
int3	float3	bool3	pixel3
int4	float4	bool4	pixel4

float2x2

float3x3

float4x4

PixelBender in detail – Types

float
int
bool
pixel1

int2	float2	bool2	pixel2
int3	float3	bool3	pixel3
int4	float4	bool4	pixel4

float2x2
float3x3
float4x4

PixelBender in detail – Operators

- Scalars + - * /

PixelBender in detail – Operators

- Scalars + - * /
- Vectors + - * / componentwise

```
float2 a, b, c;

a = b + c;                a[ 0 ] = b[ 0 ] + c[ 0 ];
                                   a[ 1 ] = b[ 1 ] + c[ 1 ];
```

PixelBender in detail – Operators

- Scalars + - * /
- Vectors + - * / componentwise

```
float2 a, b, c;

a = b + c;                    a[ 0 ] = b[ 0 ] + c[ 0 ];
                              a[ 1 ] = b[ 1 ] + c[ 1 ];
```

- Matrices + - / componentwise

PixelBender in detail – Operators

- Scalars + - * /
- Vectors + - * / componentwise

```
float2 a, b, c;

a = b + c;                    a[ 0 ] = b[ 0 ] + c[ 0 ];
                              a[ 1 ] = b[ 1 ] + c[ 1 ];
```

- Matrices + - / componentwise
- Matrices * linear transform multiplication
- Vector / matrix * linear transform multiplication

(For componentwise matrix multiply use matrixCompMult)

PixelBender in detail – Functions

sin	log	clamp	any	sample
cos	log2	mix	all	sampleLinear
tan	sqrt	smoothStep	not	sampleNearest
asin	abs	matrixCompMult	nowhere	
acos	sign	inverseSqrt	everywhere	lessThan
atan	floor		transform	lessThanEqual
atan	ceil	length	union	greaterThan
radians	fract	distance	intersect	greaterThanEqual
degrees	mod	dot	outset	equal
pow	min	cross	inset	notEqual
exp	max		bounds	
exp2	step		isEmpty	

PixelBender in detail – Functions

sin	log	clamp	any	sample
cos	log2	mix	all	sampleLinear
tan	sqrt	smoothStep	not	sampleNearest
asin	abs	matrixCompMult	nowhere	
acos	sign	inverseSqrt	everywhere	lessThan
atan	floor		transform	lessThanEqual
atan	ceil	length	union	greaterThan
radians	fract	distance	intersect	greaterThanEqual
degrees	mod	dot	outset	equal
pow	min	cross	inset	notEqual
exp	max		bounds	
exp2	step		isEmpty	

PixelBender in detail – Functions

sin	log	clamp	any	sample
cos	log2	mix	all	sampleLinear
tan	sqrt	smoothStep	not	sampleNearest
asin	abs	matrixCompMult	nowhere	
acos	sign	inverseSqrt	everywhere	lessThan
atan	floor		transform	lessThanEqual
atan	ceil	length	union	greaterThan
radians	fract	distance	intersect	greaterThanEqual
degrees	mod	dot	outset	equal
pow	min	cross	inset	notEqual
exp	max		bounds	
exp2	step		isEmpty	

PixelBender in detail – Functions

sin	log	clamp	any	sample
cos	log2	mix	all	sampleLinear
tan	sqrt	smoothStep	not	sampleNearest
asin	abs	matrixCompMult	nowhere	
acos	sign	inverseSqrt	everywhere	lessThan
atan	floor		transform	lessThanEqual
atan	ceil	length	union	greaterThan
radians	fract	distance	intersect	greaterThanEqual
degrees	mod	dot	outset	equal
pow	min	cross	inset	notEqual
exp	max		bounds	
exp2	step		isEmpty	

PixelBender in detail – Functions

sin	log	clamp	any	sample
cos	log2	mix	all	sampleLinear
tan	sqrt	smoothStep	not	sampleNearest
asin	abs	matrixCompMult	nowhere	
acos	sign	inverseSqrt	everywhere	lessThan
atan	floor		transform	lessThanEqual
atan	ceil	length	union	greaterThan
radians	fract	distance	intersect	greaterThanEqual
degrees	mod	dot	outset	equal
pow	min	cross	inset	notEqual
exp	max		bounds	
exp2	step		isEmpty	

PixelBender in detail – Functions

sin	log	clamp	any	sample
cos	log2	mix	all	sampleLinear
tan	sqrt	smoothStep	not	sampleNearest
asin	abs	matrixCompMult	nowhere	
acos	sign	inverseSqrt	everywhere	lessThan
atan	floor		transform	lessThanEqual
atan	ceil	length	union	greaterThan
radians	fract	distance	intersect	greaterThanEqual
degrees	mod	dot	outset	equal
pow	min	cross	inset	notEqual
exp	max		bounds	
exp2	step		isEmpty	

Adobe

Recap: overall shape of PixelBender

- Matrix, vector and intrinsic heavy language
- No recursion
- No pointers
- Limited use of arrays
- No user defined structures

- It's a shader language optimized to run on GPU
- Per-frame operations for handling image-processing specific semantics

PixelBender -> LLVM

- evaluatePixel
- mainLoop
  - Loops over the pixels
  - Translates requests for images on a theoretical “real” image plane to pixel coordinates
  - Calls evaluatePixel and setPixel
- mainLoopExternal
  - All functions have external signature of void foo(void**)
- Callbacks for many intrinsics
  - PixelBender, like GLSL, has a host of mathematical intrinsics that operate on vector and scalar values

Identity filter (reminder)

```
kernel Identity
{
    input image4 src;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sample(src, outCoord());
    }
}
```

evaluatePixel in LLVM-IR

```
define void @evaluatePixel(<4 x float>* %dst, IMAGE* %src, <2 x float> %_OutCoord,
i32* %_executionStatus) {

Entry_evaluatePixel:
    %sampledPixelPtrRaw = alloca <4 x float>, align 16
    br label %Body_evaluatePixel

Body_evaluatePixel:        ; preds = %Entry_evaluatePixel
    %_OutCoordElem = extractelement <2 x float> %_OutCoord, i32 0
    %_OutCoordElem1 = extractelement <2 x float> %_OutCoord, i32 1
    %sampledPixelPtrAsFloatPtr = bitcast <4 x float>* %sampledPixelPtrRaw to float*
    call void @_AIF_sampleLinear(float %_OutCoordElem,
        float %_OutCoordElem1, float* %sampledPixelPtrAsFloatPtr, IMAGE* %src)
    %sampledPixelPtr = load <4 x float>* %sampledPixelPtrRaw, align 1
    store <4 x float> %sampledPixelPtr, <4 x float>* %dst, align 1
    br label %Exit_evaluatePixel

  Exit_evaluatePixel:        ; preds = %Body_evaluatePixel
    ret void
  }
```

IMAGE is an LLVM struct type with a bunch of elements

Adobe

main and _mainExternal for identity filter

```
define void @main(<4 x float>* %_regionToGenerate,
                  IMAGE* %dst, IMAGE* %src,  i32* %_executionStatus)

define void @_external_main({ <4 x float>**,  IMAGE**,  IMAGE**,
                              i32** }* %boxedParameterBox)
```

In C pseudo-code, our LLVM function _externalMain:

```
void _externalMain(ParameterBox* bPB) {
   main(*(bPB->_regionToGenerate),*(bPB->_dst),
           *(bPB->_src), *(bPB->_executionStatus));
}
```

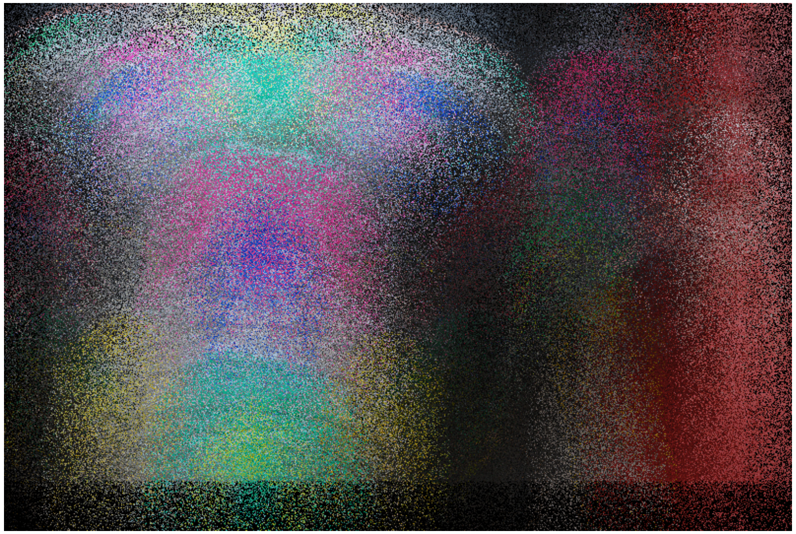
- From within our runtime, _external_main has the signature void _externalMain(void** boxedParameterBox).
  - Our external parameters are passed into main as an array of void* to the actual parameters.
- We can bypass the JIT with this

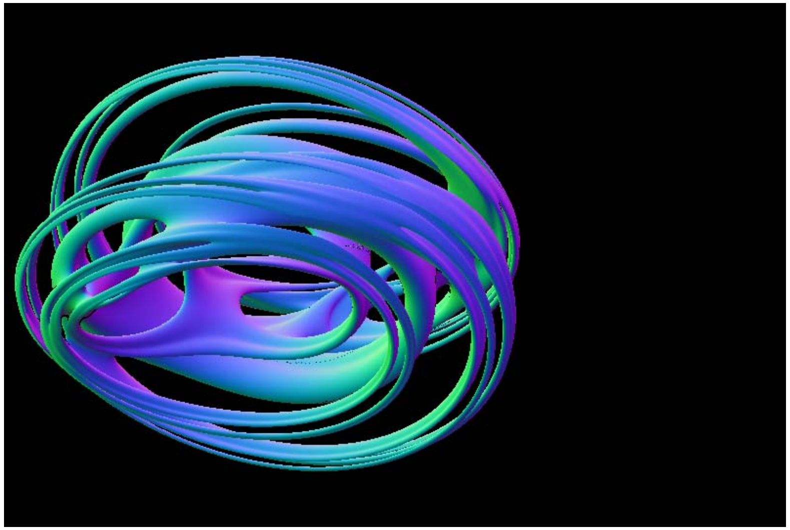
Some filters:

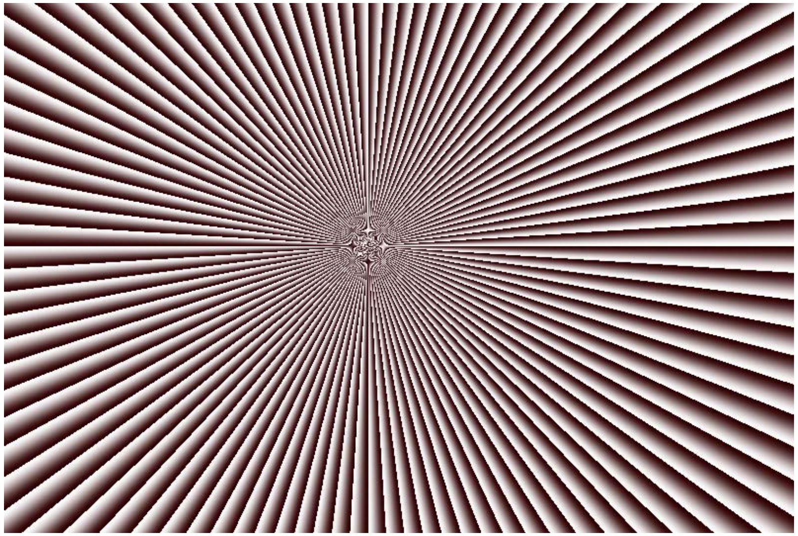





Some filters from the field….

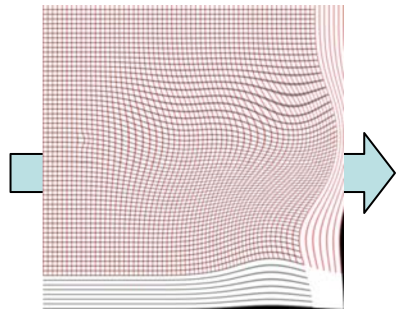




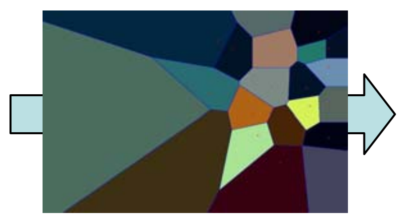


*(1) Tubeview Petri Leskinen, Jan 2008 (2) Fuzz by Tyler Glaiel (3) Julia Set by Luca Deltodesco (4) Radial Mario Klingemann

A couple of mine…


Radial basis function image warping. RBF=2D BSpline


Calculate Voronoi diagram implicitly and tint by pixel at cell center

Some results

- Per-frame calculations via LLVM compiled programs are plenty fast
- Per-pixel calculations via LLVM are a lot slower than a modern GPU
  - This isn't surprising
  - Cores are easy to use for per-pixel operations
- Things slowing us down:
  - Not using LLVM as well as we should
  - SSE usage limited
  - Callbacks not optimal
  - Security & numerical error trapping

Numerical troubles caused by heterogeneity

- What does x / 0.0 mean?
  - Inf on CPU. 0.0 on GPU
- What does i / 0 mean?
  - SEH on Windows
  - Mach exception on MacTel
  - 0 on MacPPC
- Intrinsics present differently on CPU and GPU
  - pow(x, y) | y < 0.0
- GLSL is officially unspecified as to the behavior
  - Filter writers have come to rely on this funky go-to-zero math
  - They weren't happy when we did not reproduce this behavior on the CPU

Numerical instabilities / hanging the CPU

- Numerically unstable calculations used for loop terminations in kernels are unwise but legal
- On GPU, termination by good fortune or fiat….
  - Inf and Nan just become zero, which will likely propagate through calculations “better”
  - GPU driver will nuke kernel from orbit if it runs too long
- No such protection on CPU….
  - Occasional callbacks can mitigate problem somewhat
  - Threading, out-of-process, etc., can also be used to give calling program safe place from which to terminate

Security issues

```
Parameter int selector;
void evaluatePixel()
{
   float4 localVector;

   localVector[selector] = …
```

- Shader programs present a fairly sandboxed execution model and no direct access to functions which effect system calls, but….

- Indirect array access can cause trouble
- Stack trashing or changing function return address
- At present, we check bounds on all indirect array/vector accesses.
  - Ideally, we'd like to skip that if we can analyze and know something about the index. In this case, since it comes from outside, it's pretty unconstrained

Some challenges we've faced….

- Most of my customers run Windows
  - Visual studio and/or Intel compiler are Adobe's compilers for Windows
  - VS not integrated into LLVM build and testing regimen
  - Stack alignment / SSE
  - Win64
- LLVM can crash
  - asserts and *NULL have bitten me many times.
  - LLVM doesn't fare well in low or out of memory conditions (*NULL)
  - Having LLVM live in-process requires a lot of testing
- API instability / checkin velocity

Challenges, continued….

- Platform specifics leak into IR
  - More or less need use vectors which actually exist on target architecture
  - Writing back end becomes much more complex: vectors and vector ops vs. arrays and scalar ops.
- Lack of intrinsics / types which would make my life easier
  - Matrix type <4 x 4 x float>?
  - Matrix multiply intrinsic
  - Sin/cos/etc. Currently we call back to scalar library functions, forcing de-vectorization
- LLVM is big.
  - On Mac: ~27 MB release / ~270 MB debug

Things we want to do

- Stuff I can’t talk about:
  - Which point products will ship AIF1.0
  - AIF specific optimizations
- Construction of better LLVM-IR
  - Some of our operations create “wordy” IR
  - Not being optimal with our use of loads/stores/etc.
- Work on stability issues
  - Error reporting
  - Better use of SSE
- Consumer to producer
  - Suggestions on how we can help would be great

Conclusions

- LLVM fits our needs nicely
- Some mismatch between our language and LLVM-IR
- Cross-platform has been a sticky wicket
- Security and stability concerns with JITted code on host CPU

Questions?

- http://labs.adobe.com/wiki/index.php/Pixel_Bender_Toolkit
- Search for
  - PixelBender
  - AIF
  - Adobe Image Foundation
  - PixelBender exchange
    - Community site for sharing PixelBender filters