Rendering fractal flames on the GPU

Matt Znoj   Michael Semeniuk
Nicolas Mejia   Steven Robertson

September 29, 2011

Flame overview

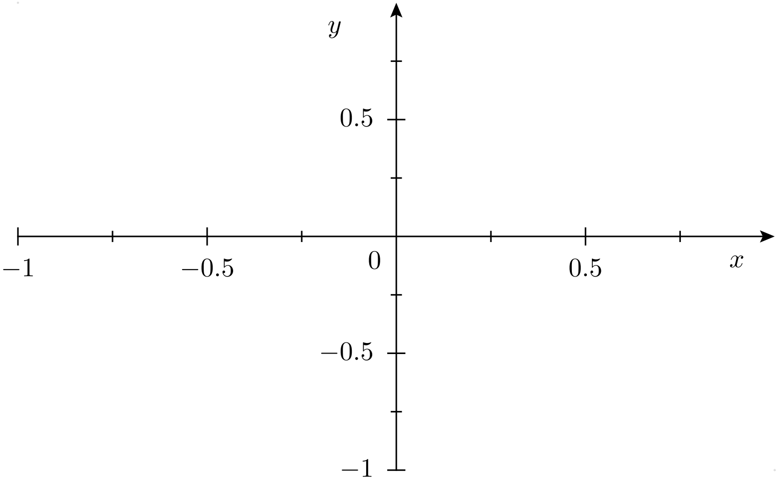
$$
\begin{aligned}
(x_{n+1}, y_{n+1}) &= (0.5x_n, & 0.5y_n) \\
(x_{n+1}, y_{n+1}) &= (0.5(x_n + 1), & 0.5y_n) \\
(x_{n+1}, y_{n+1}) &= (0.5x_n, & 0.5(y_n + 1))
\end{aligned}
$$

Flame overview

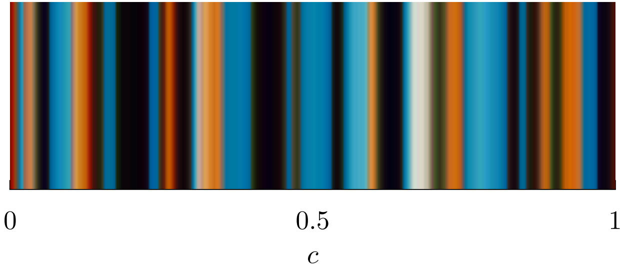
```
<flame name="electricsheep.244.04653" time="0" size="1024 1024"
center="-0.0304807 0.152945" scale="640" rotate="0"
supersample="4" filter="1" filter_shape="gaussian"
temporal_filter_type="box" temporal_filter_width="1"
quality="1000" passes="1" temporal_samples="1000" background="0 0
0" brightness="73.7913" gamma="4.28" vibrancy="1"
estimator_radius="14" estimator_minimum="0" estimator_curve="1"
gamma_threshold="0.01" palette_mode="linear"
interpolation_type="log" url="">
  <xform weight="0.122" color="0" symmetry="0" polar="0.003"
  coefs="-0.223176 1.64498 -1.64498 -0.223176 0.00173 -0.00881"/>
  <xform weight="1.829" color="1" symmetry="0" juliascope="1"
   juliascope_power="2" juliascope_dist="1"
   coefs="0.256909 0 0 0.256909 0 0" />
  <xform weight="0.458" color="0" symmetry="0" hyperbolic="2.051"
   coefs="-0.841582 -1.07778 1.07778 -0.841582 0 0" />
   <finalxform color="0" symmetry="1" perspective="1"
   perspective_angle="0.530779" perspective_dist="1.46989"
   coefs="2.01414 0 0 2.01414 0 0" />
   <!-- palette omitted -->
</flame>
```

Flame overview

Flame overview



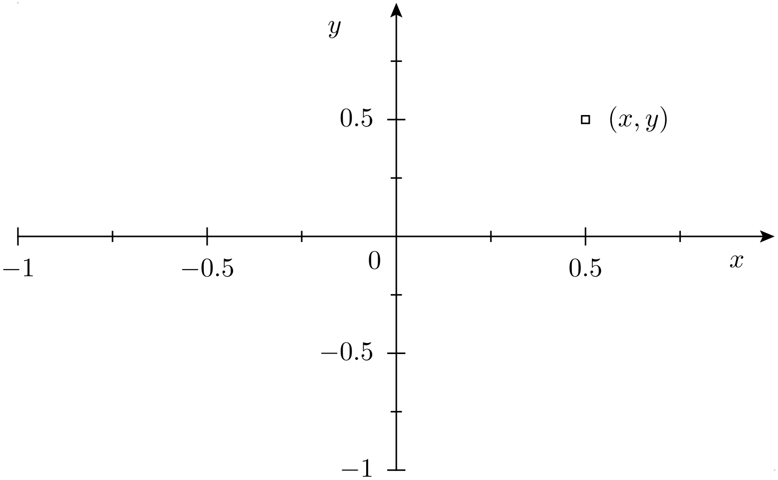
Flame overview



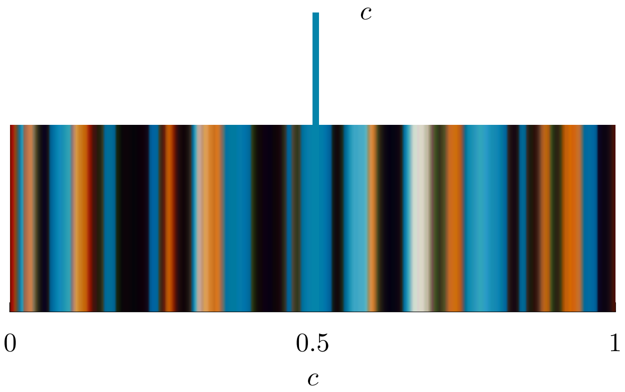
Flame overview

$$(x, y, c)$$

Flame overview



Flame overview



Flame overview

$$f_1(x, y, c)$$

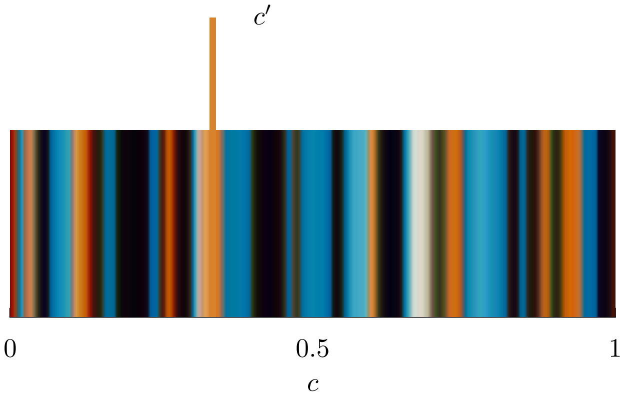
$$f_2(x, y, c)$$

$$f_3(x, y, c)$$

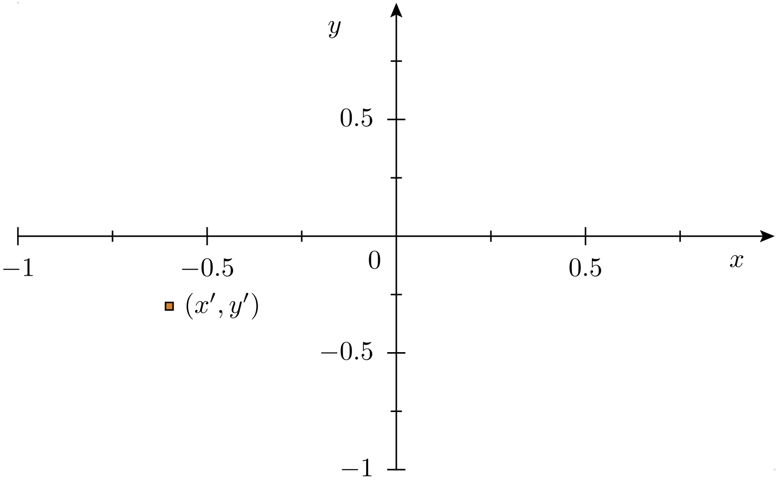
Flame overview

$$(x', y', c') = f_1(x, y, c)$$

Flame overview



Flame overview



Flame overview

40,000,000,000 iterations

The Prototype

The Prototype

Normalized to 16 Amazon EC2 compute units:

- 14.3M points per second
- 290 seconds per frame
- 116 hours per minute of video

The Prototype

Slow as balls.

The Prototype

Not complete.

Project goals and specifications

flam3: simple and complete

Project goals and specifications

flam4: simple and fast

Project goals and specifications

Chaotica: complete and fast

Project goals and specifications

Chaotica: complete and less slow

Project goals and specifications

Our project: complete, fast, GPU

Project goals and specifications

Our project: complete, fast, GPU

(...project goals)

Project goals and specifications

- AWS EC2 instance type `cg1.4xlarge`
  - 2 NVIDIA Tesla M2050 GPUs
  - 33.5 EC2 compute units (2×8 cores)
- CPU: 15M iter/sec per processor
- GPU: 1030B FMA/sec / (100 instr/iter)
  $\leq$ 1B iter/sec

Project goals and specifications

Upper bound: 68.6× faster

Project goals and specifications

Practical target: 20× faster

Project goals and specifications

Project specifications

- Runs on NVIDIA Tesla M2050 (among others)
- Supports full 24,000-parameter genome feature space
- Runs at 1920×1080 with 4× ISAA
- Renders at 20× the rate of 16 EC2 CU

Implementation: Overview

Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is. (Rob Pike)

Premature optimization is the root of all evil... look carefully at the critical code; but only after that code has been identified. (Donald Knuth)

Implementation: Overview

Bottom-up design process

Implementation: Overview

Implementation: RNG

Step 1: Randomly select a transform function.

Implementation: RNG

A “good” RNG

Implementation: RNG

Cryptographic properties

Don’t care

Implementation: RNG

Statistical properties

Implementation: RNG

Period

Implementation: RNG

n-dimensional linear correlation

Implementation: RNG

Practical properties

Implementation: RNG

Speed

Implementation: RNG

Block size

Implementation: RNG

State size

Implementation: RNG

Implementation: RNG

Lots

Implementation: RNG

Approximately lots

Sidebar: GPU resource limitations

“Won’t fit”

Sidebar: GPU resource limitations

No stack

Sidebar: GPU resource limitations

No dynamic heap

Sidebar: GPU resource limitations

High latencies for global memory

Sidebar: GPU resource limitations

16 bytes/thread shared memory

Sidebar: GPU resource limitations

20 registers

Sidebar: GPU resource limitations

Register spilling

Implementation: RNG

Implementation: RNG

Early termination and selective restart

Implementation: RNG

Huge pool of random numbers

Implementation: RNG

Mersenne Twister

Implementation: RNG

ISAAC

Implementation: RNG

Linear Congruential Generator

Implementation: RNG

Multiply-With-Carry

Implementation: RNG

Multiply-With-Carry

- Batch size: 1
- State: 8 bytes (12 for independent multiplier)

Implementation: RNG

Multiply-With-Carry

- Batch size: 1
- State: 8 bytes (12 for independent multiplier)
- Passes the Diehard tests

Implementation: RNG

- Seeding strategy to avoid cross-correlation
- Multiplier selection and distribution
- Spectral properties under shared multiplier

Implementation: Variations, Code Gen

Step 2: Apply the transform function.

Implementation: Variations, Code Gen

Linear variation

$$x'' = w \cdot x'$$

$$y'' = w \cdot y'$$

Implementation: Variations, Code Gen

Flux variation

$$x'' = w \cdot \left(2 + flux\_spread \cdot \sqrt{\frac{\sqrt{y'^2 + (x' + w)^2}}{\sqrt{y'^2 + (x' - w)^2}}} + \cos\left(\arctan \frac{y'}{x' - w} - \arctan \frac{y'}{x' + w}\right)\right)$$

$$y'' = w \cdot \left(2 + flux\_spread \cdot \sqrt{\frac{\sqrt{y'^2 + (x' + w)^2}}{\sqrt{y'^2 + (x' - w)^2}}} + \sin\left(\arctan \frac{y'}{x' - w} - \arctan \frac{y'}{x' + w}\right)\right)$$

Implementation: Variations, Code Gen

```
...
if(genome.weights[VAR_LINEAR]) {
  double w = genome.weights[VAR_LINEAR];
  xf += w * xt;
  yf += w * yt;
}

if(genome.weights[VAR_SINE])
...
```

Implementation: Variations, Code Gen

Common subexpression elimination

```
// Original code: fx += sin(x) * sin(x);
double tmp000 = sin(x);
fx += tmp000 * tmp000;
```

Implementation: Variations, Code Gen

Implementation: Variations, Code Gen

Bitmasked conditional cascade

```
if (genome.xf[1].var_mask & 0xf) {
    if (genome.xf[1].var_mask & 0x7) {
        if (genome.xf[1].var_mask & 0x3) {
            if (genome.xf[1].weights[VAR_LINEAR]) {
                ...
            }
            if (genome.xf[1].weights[VAR_SINE]) {
                ...
            }
...
```

Implementation: Variations, Code Gen

Stack-based data structures

```
int next_xf_id = POPi();
if (next_xf_id == VAR_LINEAR) {
    float w = POPf();
    fx += w * tx;
    fy += w * ty;
    next_xf_id = POPi();
}
if (next_xf_id == VAR_SINE) {
...
```

Implementation: Variations, Code Gen

Implementation: Variations, Code Gen

$$\sum_{0 \le k \le n} \binom{n}{k} = 2^n = 633825300114114700748351602688$$

Implementation: Variations, Code Gen

```
if (genome.xf[1].weights[VAR_LINEAR]) {
    float w = genome.xf[1].weights[VAR_LINEAR];
    fx += w * tx;
    fy += w * ty;
}
if (genome.xf[1].weights[VAR_SINE]) {
    ...
}
...
```

Implementation: Variations, Code Gen

```
float w = genome.xf[1].weights[VAR_LINEAR];
fx += w * tx;
fy += w * ty;
float w = genome.xf[1].weights[VAR_SINE];
...
```

Implementation: Variations, Code Gen

```
float w = xf1_weights_linear;
fx += w * tx;
fy += w * ty;
float w = xf1_weights_sine;
...
```

Implementation: Variations, Code Gen

Implementation: Variations, Code Gen

Mission accomplished!

Implementation: Chaos, Point Swapping

haha j/k

Implementation: Chaos, Point Swapping

GPUs are vector machines

Implementation: Chaos, Point Swapping

Divergent branch

Implementation: Chaos, Point Swapping

This kills performance.

Implementation: Chaos, Point Swapping

But...

Implementation: Chaos, Point Swapping

We choose at runtime.

Implementation: Dynamic Tuning

Implementation: Dynamic Tuning

"I bet pervasive use of runtime code generation results in its own set of subtle problems."

Implementation: Dynamic Tuning

Memory Management Unit

Implementation: Dynamic Tuning

Thread block

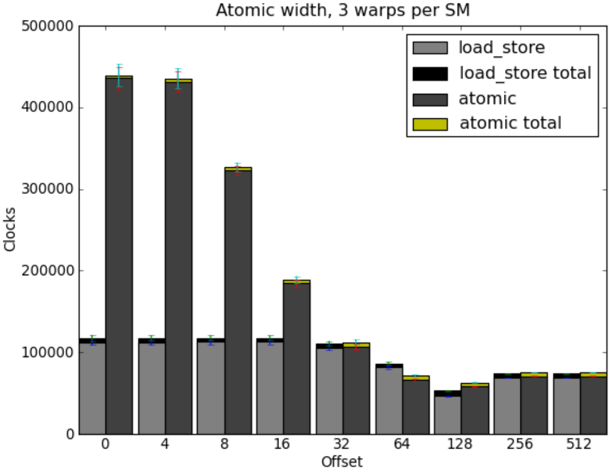
Implementation: Dynamic Tuning

Contiguous shared memory region

Implementation: Dynamic Tuning

- Number of registers
- Grid dimensions
- Block dimensions
- Parameter sizes
- Shared memory size
- Local memory size
- Texture references

Implementation: Dynamic Tuning



Implementation: Dynamic Tuning

DynaTune™

Implementation: Dynamic Tuning

(we're kidding about the name)

Implementation: System

C++

Implementation: System

Implementation: System

Compute platform

Implementation: System

Implementation: System

OpenCL

Implementation: System

Not that different

Administrivia

We’re “done”!

Administrivia

Task	Completeness
Research	100%
Planning	100%
Prototyping	100%
Discovering how wrong our plan was	100%
Performance optimizations	100% (of goal)
Feature parity	100%
Image quality testing	100%

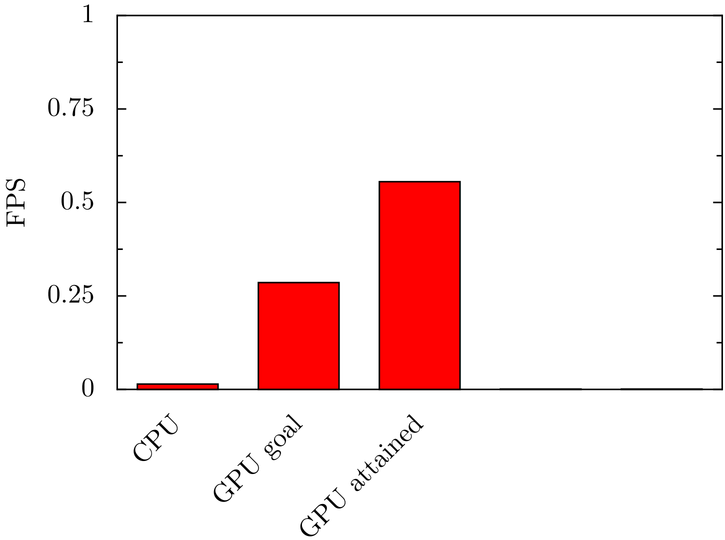
Administrivia

- OpenCL support
- Multi-card rendering
- Sorted writeback
- Machine-learning tuner
- Parallel MWC research

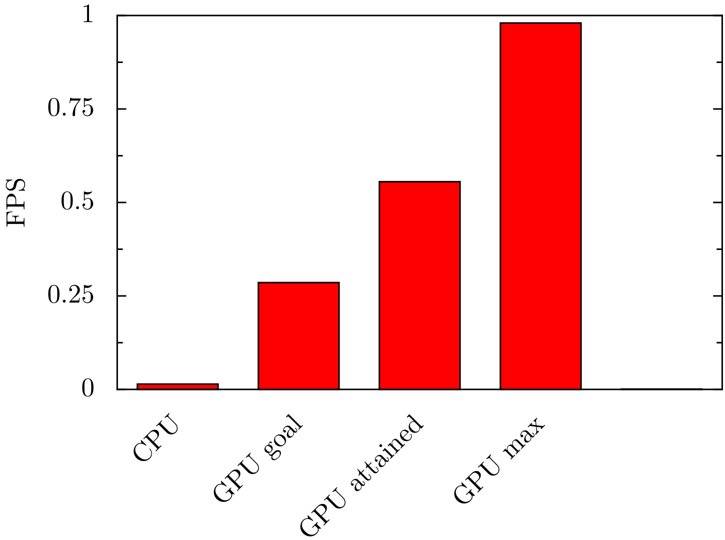
Administrivia

Not *done* done

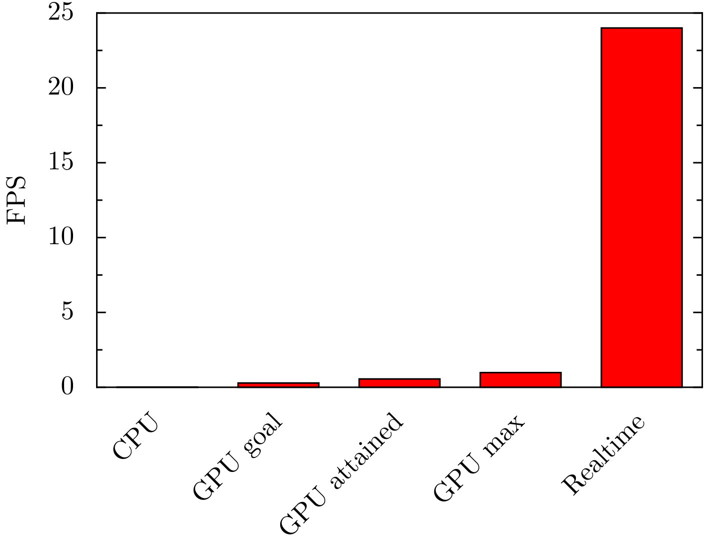
Administrivia



Administrivia



Administrivia



Administrivia

Item	Cost
AWS GPU instance (3 hours)	$6.30

Administrivia

Item	Cost
AWS GPU instance (3 hours)	$6.30
Matt's laptop (Lenovo ThinkPad W520)	$1691.00
Mike's laptop (Alienware M17XR3)	$1749.00
Steve's card (NVIDIA GTX 460 OC)	$229.99

Administrivia

Administrivia

Matt	Filtering, image enhancement, AA
Mike	Colorspace, tonemapping, writeback
Nick	RNG, `malloc()` emulation
Steve	Prototype, language tools

Administrivia

Questions?