# Cuburn, a GPU-accelerated fractal flame renderer

Steven Robertson, Michael Semeniuk, Matthew Znoj, and Nicolas Mejia

School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816

***Abstract* — This paper describes cuburn, an implementation of a GPU-accelerated fractal flame renderer. The flame algorithm is an example of a process which is theoretically well-suited to the abstract computational model used on GPUs, but which is difficult to efficiently implement on current-generation platforms. We describe specific challenges and solutions, as well as possible generalizations which may prove useful in similar problem domains.**

## I. Introduction

The fractal flame algorithm is a method for visualizing the strange attractor described in a mathematical structure known as an iterated function system, or IFS. In broad terms, the algorithm plots a 2D histogram of the trajectories of a point cloud as it cycles through the function system. Due to the self-similarity inherent in well-defined iterated function systems, the resulting images possess considerable detail at every scale, a fact which is particularly evident when animations are produced via time-varying parameters on the IFS.

Capturing this multi-scale detail requires a considerable amount of computation. The contractive nature of IFS transform functions means that histogram energy is often concentrated in a small part of the image, leaving the lower-energy image regions susceptible to sampling noise. When rendering natural scenes using related techniques such as ray-tracing, strong filtering is often sufficient to suppress this noise in non-foreground regions, but the flame algorithm intentionally amplifies darker regions using dynamic range compression in order to highlight the multi-scale detail of an attractor. As a result, flames require a very large number of samples per pixel. Further, each sample on its own is often computationally complex. Unlike many other fractal visualization algorithms, flames incorporate a number of operations that are slow on many CPUs, such as trigonometric functions.

Despite requiring an atypically large number of challenging samples, the flame algorithm is still fundamentally similar to ray-tracing algorithms, and is thus a potential target for acceleration using graphics processing units. Nevertheless, past attempts at doing so have been challenged by specific implementation details which constrain performance or reduce image fidelity.

Cuburn overcomes these challenges by what is essentially a brute-force approach. We have abandoned best practices, compatibility guidelines, and even safety considerations in cuburn in order to get as close to the hardware as possible and wring out every last bit of performance we can. This has resulted in an implementation which, while considerably more complicated than its peers, is also faster, more complete, and more correct. It has also resulted in new insights about the hardware on which it runs that are generally applicable, including a few algorithms for accomplishing commonly-used tasks more quickly.

## II. The fractal flame algorithm

At its simplest, the fractal flame algorithm is a method for colorizing the result of running the *chaos game* on a particular *iterated function system*.

An iterated function system, or IFS, consists of two or more functions, each typically being contractive (having a derivative whose magnitude is below a certain constant across the area being rendered) and having fixed points or fixed point sets that differ from one another. The Sierpinski gasket is a commonly-used example, having the three sets of affine transforms presented in (1).

$$\begin{aligned} x_0 &= \frac{x}{2} & y_0 &= \frac{y}{2} \\ x_1 &= \frac{x+1}{2} & y_1 &= \frac{y}{2} \\ x_2 &= \frac{x}{2} & y_2 &= \frac{y+1}{2} \end{aligned} \qquad (1)$$

The chaos game is simple. Starting from any point in the set of points along the attractor, choose one equation in the IFS and apply it to the point to receive a new point. Repeat this process, recording each generated point, until a sufficient number of samples have been generated. The recorded point trajectories can be analyzed to reveal the shape of the attractor. The result, in the case of Sierpinski's gasket, is shown in Figure 1.

The flame algorithm expands and codifies this algorithm in a few important ways, some of which will be described below.

To select a valid starting point for iteration, select any point within the region of interest, and iterate without recording that trajectory until the error is sufficiently small. In fractal flame implementations, this process is called *fusing with the attractor*, one of many terms chosen, arguably poorly, by the original implementors and retained for consistency. Since it is difficult to determine distance to attractor locations without first having evaluated them, most implementations simply run a fixed, conservatively large number of fuse rounds before recording begins.

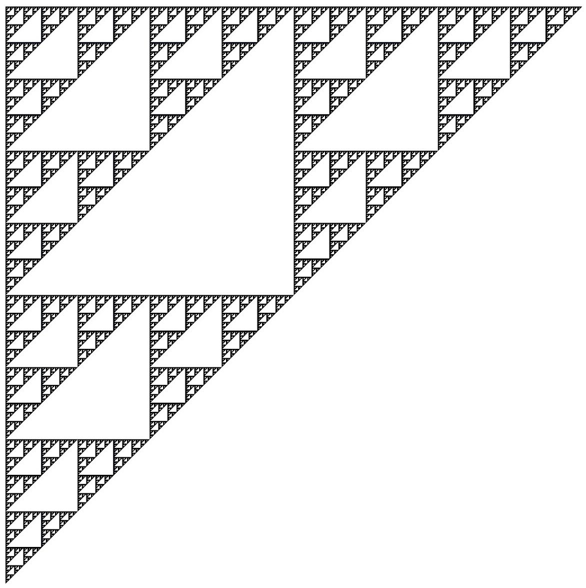


Figure 1. Sierpinski's gasket, as rendered by cuburn.

IFS transform functions in flames are considerably more expressive than the traditional affine transformations employed in fractal compression and other uses of IFS. While each of these *xforms* — another historically-motivated misnomer — does begin with an affine transform, that transformed value is then passed through any number of *variations*, a predefined set of functions which range from a simple identity function to a parameterized implementation of the Möbius transform. The weighted sum of the output coordinates of these variations are then passed through a second affine transform to produce the final transformed coordinates.

The method of selecting which xform to apply at each round is similarly augmented. Each xform is assigned a density value, which affects how often it is selected. Some flames also make use of *chaos*, which is in fact not chaotic but a relatively orderly Markov process depending only on the immediately previously selected xform.

Xforms are also augmented with a separate 1D transformation which is applied to the *color coordinate*, an indepently-tracked value with a valid domain of [0,1]. The color coordinate is used to record the point trajectories: as each point is generated, the $(x, y)$ coordinates are used to identify a memory cell in a 2D accumulation buffer, while the color is used to select a tristimulus value (traditionally RGB, though any additive color space will do) from a palette defined by the flame. The tristimulus value is then added to the appropriate cell in the accumulation buffer, and the *accumulation density* for that cell is incremented. In this way, a sort of histogram is constructed from the chaos game point trajectories.

After a prespecified number of iterations, the values in the accumulation buffer are read, and the tristimulus values are scaled logarithmically in proportion to the density of that cell. This is followed by *density estimation*, which has nothing to do with estimation, and is more accurately termed "noise filtering in extremely low density areas via a variable-width Gaussian filter proportional to accumulation cell density", although admittedly that's less catchy. A small amount of postprocessing handles such things as color values which exceed the representable range, and then an image is created.

Fractal flame images can be beautiful on their own, but are at their best when in motion. Animation of a fractal flame is accomplished by defining transform values as functions of time, either implicitly (as in the standard, frame-based XML *genome* format) or explicitly (as used in cuburn's more powerful animation-focused JSON representation). Each frame of an animation is generated by computing thousands of *control points* (which are not control points but rather the result of interpolating between the actual control points) covering the frame's display time, and then running the chaos game for each of these control points to obtain the appearance of smooth motion.

## III. Implementation overview

To render a flame, cuburn first loads a genome file and analyzes it to determine which features the genome makes use of, as well as estimates of certain constants such as the size of auxiliary buffers that may be required during the rendering process. The feature set and constant estimates are supplied to the code generator, which combines annotated code snippets to produce kernels that will execute on the compute device. Estimated values are fixed based on the results of code generation and corrected constants are injected into the kernels before uploading to the device.

Device buffers are then allocated and initialized, including a copy of the genome packed into a runtime-fixed layout determined in the previous step. For each frame of the animation, an interpolation kernel is applied to the genome data to produce a control point for each temporal step of the frame.

For most flames, the next step is to perform the chaos game. A kernel is launched which performs a portion of the requisite iterations. After each iteration, the current position is examined, and — if within the region of interest — the linear accumulator pixel address and color coordinate are packed in a 32-bit integer and appended to a log. The log is then sorted based on a key extracted from between 8 and 10 bits of the log entries, to obtain a degree of linear memory locality for histogram accumulation. An accumulation kernel reads the sorted log in multiple passes, storing information in shared memory before dispatching it to a device buffer. This process of iterate, sort, and accumulate is repeated until a sufficient number of samples have been collected in the accumulation buffer.

A density estimation kernel is launched, which performs log-scaling of the accumulated points, antialiasing postprocessing, and selective blurring to provide a depth-of-field effect and correct a limited amount of sampling noise. A final kernel applies gamma and brightness to the image and handles pixels whose values exceed the full range of representable values in the output format. The output buffer is then copied to the host asynchronously for compression.

## IV. Runtime code generation

The diversity of variations available in the flam3 and Apophysis implementations of the flame algorithm are a boon to fractal flame artists, but a curse upon implementors. While the selection of variation functions is somewhat standardized — by the simple expedient of declaring that whatever the latest version of flam3 supports is the standard — the functions themselves vary widely, from simple polynomial expressions to those that require thousands of CPU instructions and as many as ten parameters from the host.

On the CPU, this is implemented by use of a cascade of `if`/`else` conditions, which check whether or not a variation is used and, if so, apply it. A smart optimizing compiler can convert this cascade to a jump table, and modern desktop CPUs can use branch prediction to accelerate the process in either form. GPUs cannot perform relative jumps as used in a jump table, nor do they have branch predictors. Further, both the load and comparison instructions execute at less than full throughput per clock — comparison being half of full throughput, load being at most a quarter, and typically less — such that this conditional cascade, which must be executed billions of times per frame, would itself require more GPU cycles than all other components of the iteration kernel combined. Certain optimizations which are critical to attaining high throughput are also impossible to perform across basic block boundaries, such as instruction reordering, common subexpression elimination, and register renaming.

To work around this issue, we constructed a system to perform runtime code generation. Rather than distributing a compiled code object containing the kernels intended for GPU exeecution, we distribute an annotated repository of CUDA code snippets embedded into our Python library. When a flame is loaded for rendering, its genome is analyzed to determine which segments of device code are needed. These segments are gathered and joined into a consistent CUDA `.cu` file, which is then passed to PyCUDA's caching `nvcc` frontend to obtain a GPU code object in ELF format that can be loaded by the CUDA driver and copied to the device.

By selecting source code segments at runtime, we can eliminate the conditional cascade by building separate inlined functions for each xform that include the exact set of variation functions. We can also remove the conditional guards around those variations, allowing them to be merged into a single basic block and optimized more effectively. While this technique adds a considerable amount of complexity to the host side, the improved performance and flexibility in device code cannot be obtained otherwise.

While initially considered primarily to solve the variation problem described above, the runtime code generation system has found use in simplifying or improving many parts of cuburn's device code. In some cases, these improvements are small, yielding execution time changes of one or two percentage points; in others, the flexibility it offers has proved critical to reaching our performance targets. One such case is the method for interpolating and accessing control point parameters.

## V. Control point interpolation and access

As described in Section II, fractal flames are animated by allowing parameters to vary over time. For smooth, continuous motion, each animation frame is comprised of points generated by thousands of parameter sets, each sampled from the continuous distribution of parameter values at thousands of of time, smoothly covering the frame's display time. For each of these temporal samples, a distinct copy of the parameter set data structure is required.

CUDA GPUs have a provision for loading parameter sets from memory, known as constant memory. This is device memory which is shadowed by a small read-only cache at each GPU core with its own narrow address space configured to map to global memory by the host. Accesses to constant memory can be inlined into dependent operations with no overhead, not requiring a separate load instruction or temporary register, but only if that access follows certain restrictions, chief among these that the access must use a fixed offset from the start of the memory space. If a non-fixed index is used, the code makes use of the normal memory data path, which is considerably slower.

In order to run chaos game iterations on thousands of temporal samples, we need to be able to load data from a particular parameter set. Doing so with constant memory requires either performing a separate kernel launch, with corresponding constant address space configuration, for each temporal sample, or using indexing to select a temporal sample at runtime. The former method leads to ineffecient load balancing, and the latter forces constant memory accesses to take the slow path.

The most common alternative to constant memory is shared memory, which can be described as an L1 cache under programmer control. Static-offset lookups from shared memory are not quite as fast as inline constant memory lookups, but are faster than indexed lookups. However, another problem presents itself: when represented as a complete data structure, the full parameter set exceeds the maximum available 48KB of memory and far outstrips the 2KB maximum size required to obtain sufficient occupancy to hide pipeline latency on current-generation hardware.

To retain the benefits of static-offset lookups without requiring a static data structure, we augmented the runtime code generator with a data packer. This tool allows you to write familiar property accessors in code templates, such as `cp.xform[0].variations.linear.weight`. The code generator identifies such accessors, and replaces them with a fixed-offset access to shared memory. Each access and offset are tracked, and after all code necessary to render the genome has been processed, they are used to create a device function which will perform the requisite interpolation for each value and store it into a global array. Upon invocation, each iteration kernel may then cooperatively load that data into shared memory and use it directly.

Each direct property access as described above triggers the device-memory allocation and copying of the original Catmull-Rom spline knots from the genome file for that property. In some cases, it can also be useful to store something other than a directly interpolated value. To this end, the data packer also allows the insertion of precalculated values, including arbitrary device code to perform the calculation during interpolation. The device code can access one or more interpolated parameters, which are themselves tracked and copied from the genome in the same manner as direct parameter accesses. This feature is used both for precalculating variation parameters (where storing the precalculated version as a genome parameter directly would be either redundant or unstable under interpolation), as well as for calculating the camera transform with grid-shift antialiasing enabled (described in Section IX).

The generated function which prepares a control point on the device performs Catmull-Rom interpolation many times. To control code size, it is important to implement this operation in as few lines as possible. One step in variable-period Catmull-Rom interpolation involves finding the largest knot index whose time is strictly smaller than the time of interpolation. To implement this, we wrote a binary search (given in Figure 2) that requires exactly $3 \log N + 1$ instructions. We suspect this is not a novel algorithm, but we have not seen it elsewhere.

```
    ld.global.u32   rv,     [rt+0x100];
    setp.ge.u32     p,      rv,     rn;
@p  add.u32         rt,     rt,     0x100;
```

Figure 2. One round of cuburn's unrolled binary search.

## VI. Warp divergence and trajectory convergence

A CUDA compute core is essentially a 32-lane vector machine. Each lane has its own registers, and independent predication allows a lane to forego executing a particular instruction, but two lanes in the same processor cannot execute a different instruction simultaneously.

When a vector unit, called a *warp* in CUDA, reaches a branch, the compute core checks whether all vector units took the same branch. If they did, execution proceeds as normal. If the vector units diverged, independent execution is emulated by evaluating one side of the branch with all non-participating lanes disabled, then doing the same with the other side of the branch. If branches are equal in execution length, a single divergent branch takes twice as long as a normal branch to execute, plus a bit of overhead for resynchronization. Divergence is therefore to be avoided.

In the chaos game, each iteration requires random selection of a transform function to apply. If this is done for each point, the resulting threads will be divergent. With the 12 xform maximum supported by flam3, the result is that warp divergence will cause iteration to take up to 12 times as long as a work-optimal strategy. In cuburn, instead of selecting a unique xform per-point, we do so per-warp, with a shared memory buffer. This avoids warp divergence in the iteration function and its corresponding performance penalty.

On its own, however, this presents another problem. In well-behaved flames that produce visually interesting images, xforms tend to be contractive on average. Most have a small set of fixed points, such that from any initial point, the results of repeatedly applying an xform to its own output will converge upon one of these points. This property is what creates the strange attractors when the xforms are used together in an IFS: the densities across the image are the product of how each xform's fixed points attract points at the others'. However, that means that any two points which are within "pulling range" of an xform's attractor will end up closer together after passing through that xform. If all points in a warp go through the same xforms in the same order, they are very likely to wind up indistinguishably close to one another. This behavior is not necessarily incorrect, in terms of the resulting output image, but it reduces coverage in the low-density image regions which need it most.

For this reason, we also incorporate point swapping into the iteration loop. After every iteration, each thread uses shared memory to exchange its point with a thread in a different warp. The scheme by which threads are exchanged is predetermined, and follows a pattern which avoids shared memory bank conflicts on reading and writing. While this does not provide as strong a guarantee that individual points will not converge upon the same trajectories as an entirely random configuration, theoretical and experimental results show that it is sufficient to ensure trajectories do not converge to a single pixel.

## VII. Accumulation strategies

After each iteration, the color value of a point within the bounds of the accumulation buffer — a 2D region of memory slightly larger than the final image, with gutters on all sides for proper filter behavior at edges — must be added to its corresponding histogram bin. Cuburn supports three modes for performing this task: *direct*, *atomic*, and *deferred*.

The direct approach is, appropriately, the most straightforward. After an iteration, a texture lookup is performed to obtain tristimulus values for the current color, in either RGB or BT.704 YCbCr format. The contents of the appropriate histogram bin — four IEEE 754 single-precision floating point values, containing the linear-scaled tristimulus values and total density for a given accumulator pixel — are loaded into registers, added with the current point's color, and written back to device memory. Atomic mode is much the same, except instead of using a read-modify-write cycle, it relies on hardware atomic units to perform this task. This guarantees consistency, at a cost of nearly an order of magnitude in performance.

Two problems with the direct approach exist. The first is that it is typically a bottleneck for the iteration function. GPUs have an incredible amount of aggregate memory bandwidth, but the memory subsystem is optimized for serial throughput at the expense of random-access latency. Since the point trajectories of well-behaved flames will traverse the image area, the comparatively small data caches are unlikely to hold a given point when it is needed, triggering long stalls in threads that wait for data to be loaded. Due to the `__syncthreads()` barrier used for point swapping and register pressure on the loop, these delays can't easily be circumvented by latency hiding.

A more insidious problem arises as a result of using unsynchronized global memory transactions. Since thousands of threads are in flight simultaneously during chaos game iteration, it is possible for one thread to read a particular histogram value into memory, followed by another thread doing the same before the first thread has a chance to write. In this case, the second thread will overwrite the first's changes, meaning that the first thread's point is lost.

Since the chaos game is a Monte Carlo simulation, these inconsistencies may be considered another source of noise, rather than critical bugs. Fortunately, these collisions have a relatively narrow window of time in which to happen, so will have a small chance of occurring. When they do occur, they will tend to happen in the image regions which have the highest density, where the missing samples will have the least impact on the final image. Knowing these factors would likely mitigate the harm caused by this error, we implemented direct accumulation first, and found that in most circumstances it provided an image indistinguishable from reference images. However, for some flames, direct rendering has a distinctly visible negative impact. The flames typically have relatively even coverage of the region of interest, have xforms with strong attractors, and use simpler variations, such that proportionally more time is spent performing memory operations.

To solve this, the deferred accumulation mode was created. The idea behind deferred accumulation is to record information about each point in a log, and then later replay that log in order to build the histogram. While this involves more operations in a theoretical sense, splitting these stages allows for optimizations which cannot be performed in a monolithic system.

For most flames, only three values are required to reconstruct a histogram accumulation: the linear index of the accumulator destination bucket, the color coordinate, and the control point time (to determine which color palettes to blend when obtaining a tristimulus value). Some flames also specify opacity on each xform; for these, that opacity value must be recovered. Cuburn uses runtime code generation to choose the exact format of the log — two examples are presented in Figure 3 — but the procedure is similar across flames. The upper 9 bits are used to store the color index; the color coordinate is scaled to [0,511] and dithered before packing to reduce quantization error. If needed, this value may be reduced to 8 bits with a warning. The next two bits are used to store the opacity index, if xform opacity is used. Rather than storing the opacity, the opacity index identifies which of up to four unique time-varying opacity values to apply. Most flames will only apply opacities other than 1.0 to a few xforms, so this is more efficient than storing xform index diretly, and dramatically more so than storing a scaled integer version of the opacity value itself. The remaining bits in the address are given to the accumulator index. Each control point calculates start and stop points within the log, so time can be calculated implicitly from the current log index. Writes to this index from the iteration kernel are coalesced and need not be synchronized; for Fermi devices, writeback caching makes this access pattern essentially free from the perspective of the iteration kernel.

The accumulation kernel reads log entries, recovers these values, uses them to choose the correct tristimulus value, and adds it to the histogram. In order to do this efficiently, accumulation kernels make use of atomic shared memory transactions. These operations are not as fast as normal shared memory operations, but are considerably faster than global accesses of any kind, and particularly atomic accesses under high load.

This presents yet another problem. Current-generation compute devices are limited to 48KB of shared memory, a tiny fraction of overall framebuffer size. In order to perform these accumulations in shared memory, a kernel must shadow a portion of global memory in shared memory.

```
|                             | Sort key (9 bits)        |  | Shared key (12 bits)              |
| Color (9 bits)        |     | Accumulator address (22 bits)                                   |
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0

|                                      | Sort key (7 bits)     |                                |
|                                                           | Shared key (12 bits)              |
| Color (9 bits)        | XFO |        | Accumulator address (19 bits)                          |
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

Figure 3. Top: the log format used for a 2560x1600 image which does not use xform opacity. Note that each memory cell in the sorted log is scanned twice in this configuration, since bit 12 is not sorted; each pass ignores log entries outside of its boundary. Bottom: The log format used for a 720x480 image. To avoid entropy reduction, bit 11 is included in the sort.

Unlike a typical cache, this must be done ahead-of-time, since shared memory is manually managed (and since cache misses would cripple the performance of this kernel as much as that of the iteration kernel). Maximizing the number of accumulation buffers that fit within this region is therefore desirable. To do so, we use a static log-scale compression format to shadow the 16-byte set of four floats which represent a pixel in 8 bytes of shared memory, and efforts are underway to reduce this to 4 bytes. Even at 4 bytes per histogram bin, however, the number of full scans of the log would be prohibitively large. Instead, the log is sorted based on the upper bits of the address — those which identify an index at which to shadow — and this sorted copy is used as the source for iteration scanning.

## VIII. Cuburn sort

GPU algorithms for sorting are an area of active research in the HPC community. Sorting is a common, intuitive operation, useful in many problems and fundamentally related to more, but the unusual nature of massively-multithreaded devices has made the development of a high-performance sort function into a landmark achievement for both a hardware platform and its developers.

To be useful in cuburn, a sort function must be easily callable within the complex asynchronous dispatch code used to schedule operations on the GPU. It must also be considerably faster than the overhead for atomic accumulation, for otherwise development effort would be better spent optimizing that routine directly. Finally, it must support operating on a partial bit range, so that the address coordinates of point log entries can be sorted into distinct groups which target each accumulation shadow region without performing a slow and wasteful sort of the entire log entry.

Of the publicly documented sorts we found, only MGPU sort fits the requirements. However, MGPU sort was not released until some time after we had begun efforts towards constructing one, and given initially promising results, we decided to press on with our own sort. While cuburn sort possesses a number of limitations which make it less well-suited to general-purpose tasks, it is at the time of publishing the world's fastest sort on CUDA architectures at sorting between 7 and 9 bits of randomly-distributed data, as shown in Figure 4.

|  | Cuburn | MGPU | B40C | CUDPP |
|---|---|---|---|---|
| 7 bits | 821 | 740 | 551 | 221 |
| 8 bits | 943 | 813 | 611 | 251 |
| 9 bits | 966 | 877 | 475 | 191 |
| 10 bits | 862 | 910 | 528 | 211 |

Figure 4. Performance of different sort implementations, as measured on a GTX 560 Ti 900MHz. Values are in millions of keys per second, normalized to 32-bit key length.

The sort is accomplished in four major steps. The first pass divides the buffer to be sorted into blocks of 8,192 values, and performs an independent scan operation on each. Unlike the other sorting techniques benchmarked here, cuburn's scan uses shared memory atomics to perform this accumulation, by performing an atomic increment on a shared memory index derived from the point under analysis. This process is much slower than traditional prefix scans, but because it is coordinated across all threads in a thread block, it allows the derivation of a radix-specific offset for each entry in the current key block. This offset is stored into an auxiliary buffer for every key processed, and the final radix counts are stored to a separate buffer.

The second step loads the final radix counts and converts them to local exclusive prefix sums, storing these in a separate buffer. This is performed quickly and with perfect work efficiency by loading the radix counts into shared memory using coalesced access patterns, rotating along the diagonal of the shared memory buffer, and performing independent prefix sums in parallel horizontally across the buffer, updating values in place.

A third step operates on the final radix counts, transforming them in-place to per-key-block, per-radix offsets into the output buffer. This is accomplished by first reducing the buffers via addition in parallel to a very small set in a *downsweep* operation, performing the prefix scan on this extremely limited set in a manner that is not work-efficient but, due to the small number of buffers involved, completes in microseconds, and then broadcasting the alterations back out to the full set of buffers in an *upsweep* operation.

Sorting is accomplished by using the offsets and local prefixes to load each key in the block to a shared memory buffer, then using the local and global prefixes to write each key to the output buffer. Transaction lists are not employed, but large block sizes help to minimize the impact of transaction splitting.

One unusual characteristic of this architecture is that performance on data that does not display a good deal of block-local entropy in radix distribution is actually considerably slower than sorting truly random data. Sorting already-sorted data is a worst case, with a penalty of more than an order of magnitude arising from shared-memory atomic collisions during the initial scan. We avoid this in cuburn by ensuring that the radix chosen to sort never includes bits that are zero throughout the log. In some cases, this means that some bits of the accumulation kernel's shadow window are also sorted, since cuburn sort does not scale down to arbitrarily small radix sizes. This oversorting is theoretically less efficient, but is actually faster than including the zero bit in the sort.

A simple but powerful optimization enables discarding of any key equal to the flag value `0xffffffff`. This value is used internally by cuburn to indicate that a particular point log entry corresponded to an out-of-bounds point and should be ignored. Discarding these during the sort stage improves performance by around 40% on average for our test corpus of flames.

## IX. Antialiasing and filtering

Many flames contain sharply-defined curves which leave visible artifacts when their position is quantized to the image grid. Cuburn employs a variant of multi-sample antialiasing to reduce the appearance of these artifacts. During precalculation of the camera transform, which maps IFS coordinates to accumulation buffer indices, the sampling grid for each temporal sample is offset according to a 2D Gaussian distribution with a user-controlled standard deviation that defaults to one-third of a pixel for compatibility with flam3. With thousands of samples per image pixel taken on average, this technique gives very high quality antialiasing with essentially zero overhead.

After iteration and accumulation have finished, a density estimation kernel processes the output. The DE kernel again avails itself of copious amounts of shared memory to accelerate processing of image elements. Because variable-width Gaussian filters are nonseparable, the DE filter is applied in square 2D blocks of size $2R + 32$ pixels in each dimension, where $R = 10$ is a constant governing the maximum permissible filter radius and consequently the gutter size, and 32 is is both the horizontal and vertical size of the thread block. Each thread in the block loads a point from the accumulation buffer (with thread $[0, 0]$ loading the value corresponding to a local offset of $[R, R]$, and thread $[31, 31]$ loading $[R+31, R+31]$), calculates the filter radius, and proceeds to write the offsets in a spiral pattern designed to avoid bank conflicts and enable early termination without warp divergence. After the 1,024 pixels in the current block have been processed, the entire shared memory region is added to the output buffer, including gutters, and the column advances vertically by 32 pixels. This pattern treats every input pixel exactly once, but due to gutter overlap, output pixels may be handled by as many as four separate thread blocks. Nevertheless, this kernel operates efficiently, taking less than one percent of a typical frame's GPU time.

GSAA occurs entirely before DE, which can in some instances create image artifacts. Edges that have been smoothed by GSAA whose tangents lie close to but not exactly on the image grid will have a strong density gradient adjacent to the edge, including one point at every pixel step which will receive a small fraction of the edge's full density. When this occurs on a strong edge adjacent to a very low density area, DE will blur this lowest-intensity point very strongly, leading to "edge bloom" (see Figure 5).

Cuburn adds an edge-detection filter to the DE process. When a weak pixel adjacent to a strong edge in a low-density area is detected, the filter's radius is clamped to prevent edge bloom. Different edge detection methods were tested, including the common Sobel and Roberts cross image kernels, but these methods resulted in incorrect detection, with higher sensitivities resulting in excessive false positives and lower ones allowing some bloom to escape detection. Cuburn therefore uses the $L^2$ norm of a three-pixel-wide variant of the Roberts cross convolution kernels given in (2), which performs very well at properly detecting edges in flames.

$$A = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{2}$$

## X. Acknowledgement

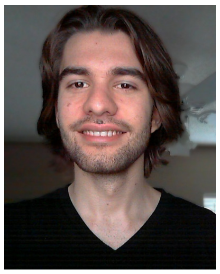
The authors acknowledge Drs. Samuel Richie, Mark Heinrich, Sumanta Pataniak, and Niels Lobo for their guidance and evaluation. We also thank Erik Reckase for his assistance in testing cuburn and in sharing his edge creation tools.

## XI. Biography

Michael Semeniuk is currently a senior at the University of Central Florida and intends to graduate with his B.S. degree in Computer Engineering in December of 2011. In the Spring of 2012, he will start graduate studies at the University of Central Florida concentrating in Machine Learning and Intelligent Systems. He has accepted a job offer at Lockheed Martin: Global Training and Logistics and is currently working on commercial flight simulator software. His primary interests lie in data mining, nanotechnology, and machine learning.

Matthew Znoj intends to recieve his B.S. degree in Computer Engineering in December of 2011. He will start his M.S. degree in Computer Engineering (Computer Systems and VLSI concentration) in January of 2012. He has accepted a job offer for Design Verification Engineer at AMD. Before coming a regular AMD employee, he held internships at AMD, DRS Technologies, Micromint USA, and Cubic Simulation Systems.

Nicolas Mejia is currently a senior at the University of Central Florida and will receive his Bachelor's of Science in Electrical Engineering in December of 2011. He plans on pursuing a Masters in the field of Digital Signal Processing or Photonics at Stanford University. He currently tutors other students in math and science and he has a wide range of interests from art to science.



Steven Robertson will begin working at Google in February 2012. His website is http://strobe.cc.

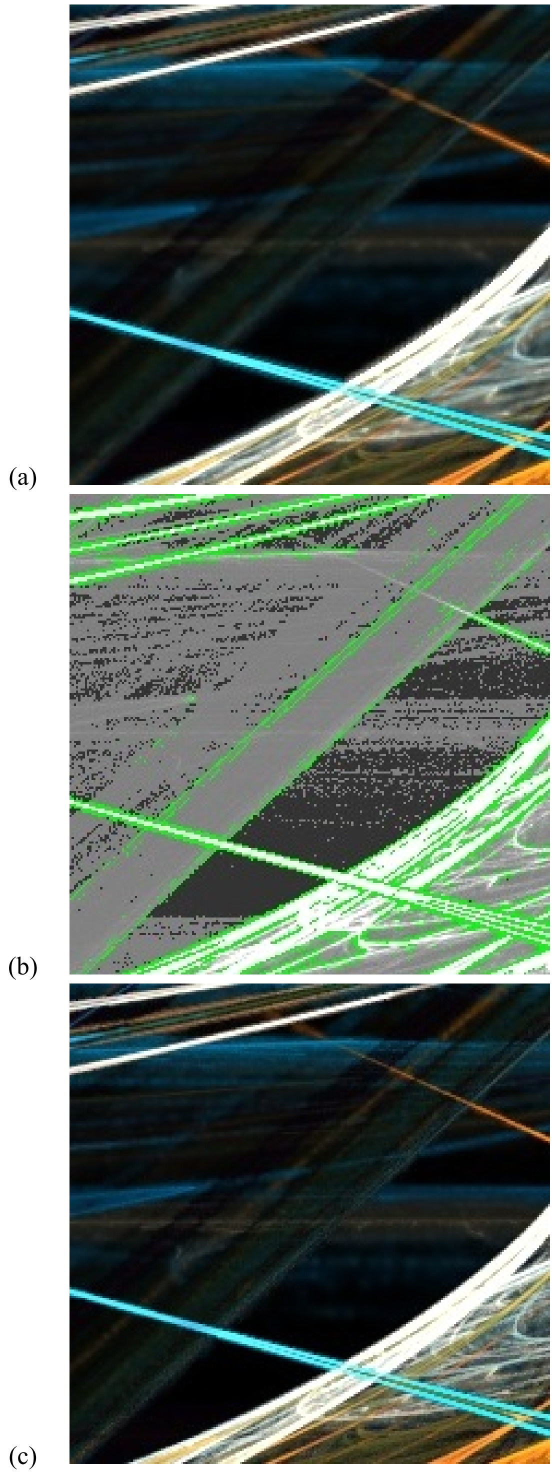


Figure 5. (a) An example of edge bloom caused by improper handling of antialiased edges during density estimation. (b) A view of the same image highlighting detected strong edges. (c) The results of applying edge detection and filter clamping to density estimation.