

Can androids render Electric Sheep?

AUTHOR: Steven Robertson
CONTACT: steven@strobe.cc
PUBLISHED: 2009-09-21
TAGS: CUDA, Algorithm, Article, MathML
ABSTRACT

I tried to implement a GPU-accelerated version of the algorithm behind Electric Sheep, and failed. Here's what I learned—and what I'm doing to fix it.

The [Electric Sheep](#) screensaver combines a genetic algorithm, an iterated function system, and some post-processing to create one of the most mesmerizing visualizations I've ever seen. Over the summer of 2008, I looked into creating a version of the rendering library of Electric Sheep, a library known as [flam3](#), which would use NVIDIA GPUs to accelerate their transforms.

The algorithm is pretty straightforward. In broad terms, a fractal flame, or simply *sheep*, consists of a set of equations, each having as input a coordinate (xi, yi) and as output a coordinate (xo, yo) . Start with a random point. Plug in this point's coordinates into one of these randomly-chosen pairs of equations, and plot the result. Repeat. When it's done, you get a fractal image. There's more to it than that, of course - here's [Scott Draves' paper](#) on the subject if you want the details.

At first glance, this maps relatively well to the idea of a massively parallel processor like a GPU. Instead of running one point at a time, just run hundreds! Unfortunately, as you might have guessed, it ain't so simple.

Part of the problem in getting the transforms over to the GPU had to do with NVIDIA's CUDA SDK. Normally, CUDA is supposed to be written in a C-like language, which gets compiled to an assembly language equivalent. The assembly - written in NVIDIA's own PTX language - then gets some register-allocation optimizations before winding up as machine code. The PTX emitted by the first implementation of the transform in CUDA's C dialect did not agree with the register allocator, and the resulting code used more than 50 registers after allocation.

50 registers is really bad. In CUDA, each processor unit handles hundreds of threads, and attempts to hide latency by switching between threads rapidly. Context-switching has a low overhead because all registers are allocated at the start of a thread and remain allocated throughout the thread's lifetime. In first-generation GPGPUs, maximizing the occupancy of each processor (and therefore hiding the most latency) required kernels which used a mere 10 registers. Needless to say, this code performed very poorly (for this and other reasons). After fighting with the C code for a while, I gave up on the compiler and decided to code the entire transform kernel in assembly.

It helps that PTX is actually a rather nice assembly language, as far as they go, but it still turned out to be a pretty significant task, taking me the better part of a season to even get to a workable state. Assembly

can be challenging for large applications, but the time and effort had more to do with the unusual memory architecture of NVIDIA GPUs than the programming language in use.

HARDWARE INTERPOLATION

Some essential background information is necessary for the next sections to make sense; most of this is detailed in NVIDIA's [CUDA Programming Guide](#) [pdf]. A lot of this section is well-founded speculation.

Computation on the GPU is handled by *multiprocessors*. Each multiprocessor on a current-gen (GTX 200 series) GPU is capable of tracking 1024 threads in-flight. These threads are grouped into *warps* of 32 threads; each of these warps has a single instruction pointer, meaning that the 32 threads must all execute the same instructions on different data. (Instructions can be predicated, meaning the results of those computations are not stored, so branches where some threads of the warp follow a different path can be simulated by following one path with some threads “turned off,” then the other path with the complement threads disabled. Highly-divergent paths are an absolute disaster, performance-wise.)

Each multiprocessor will step through a warp in two “front-end” clock cycles. (The front-end faces the multiprocessor bus, which interfaces with the memory controller; the back-end contains the ALU and is double-clocked.) During each of these clock cycles, threads being executed can issue a memory access request. If the memory access meets certain criteria, the front-end bundles the requests into a single transaction and ships it off to the memory controller; if not, the requests are transmitted individually.

Main memory on an NVIDIA GPU is accessed by a controller that is shared by all processors. Memory requests are sent down a very wide bus to the controller, which queues them. The controller interleaves storage across all RAM on the board in order to increase bandwidth. When processing requests for contiguous blocks of data, this has a significant effect on performance; the latency cost of bringing 14 separate RAM chips (on my GPU) to the same address is less significant than being able to complete your transaction in a few clock cycles. Unfortunately, it means that the latency cost for accessing a single byte of data is at worst case 14 times higher than for non-interleaved memory architectures. I’m not sure if the controller can process requests for non-contiguous memory simultaneously if the regions are located on separate chips, but given the dire warnings against non-contiguous memory access scattered throughout CUDA documentation, I doubt it.

These latencies are compounded by the lack of any real cache. The decision to omit a cache from each multiprocessor is the right one; for most operations, a small cache would just miss anyway, a large one would be absurdly expensive. Worse still, keeping cache coherent between 30 processors would require jaw-dropping complexity. To sort of make up for it, NVIDIA places 16K of (usually) penalty-free RAM on each multiprocessor, which must be shared by all active threads. While it sounds like it could be used as a manually-managed cache, the size of the shared memory is too small to make that effective; at full capacity, you get a mere 16 bytes per thread. This memory is the only way to communicate with the other threads active on a particular multiprocessor, so it is more often dedicated to coordination instead of cache.

There is also a shared 8K cache per multiprocessor which shadows a region of memory only writable by the host system; this *constant memory* is useful in providing constants (transform coefficients, pointers to buffers, etc) to a kernel, but cannot be used for coordination as it is not guaranteed to be consistent if memory is changed by the host after the kernel is started.

These hardware oddities can be summed up in three simple design rules for CUDA applications:

- Split your tasks into groups of 32 threads.
- Make these groups branch as little as possible.
- Access memory infrequently and contiguously.

PORTING FLAM3

These three goals may seem trivial, but they can be Zen-like in their elusiveness. Some unexpectedly difficult parts of the port are outlined below, along with solutions, in no particular order. I'll continue to update this as I learn more.

RANDOM NUMBER GENERATION

The iterated function system at the heart of the fractal flame algorithm depends heavily on high-quality random numbers. The current implementation uses the PRNG. However, a reasonable ISAAC implementation requires a minimum state of 144 bytes per random context, and ISAAC produces random numbers using a procedure that must operate in serial over the context. At a minimum, using atomic transactions to block the execution of any other thread on the chip, and keeping one random context per warp, the context alone would consume 4,608 bytes. This blows through the entire allocation of shared memory for a 256-thread warp without storing a shred of data about the IFS itself.

Stronger random number generators, such as the popular Mersenne Twister, could have been selected; these generators would have to be called separately to generate and store a large block of random numbers, which would then be read in as needed by each warp. This solution may be the best in terms of quality of generated numbers, but the coordination required to get this working was deemed too expensive for a first effort.

Ultimately, I decided to go with an [MWC](#) algorithm, using different values of a for each thread selected from a pregenerated table distributed with my build. Given the extensive list of compromises already made in porting `flam3` to the GPU, I doubt that an MWC algorithm would make a significant impact on the results of the computation. This method was implemented using two persistent registers per thread, and required no shared memory. I will reconsider this decision after I see CUDA 3.0's memory hierarchy, which arrives alongside the GTX 300 series sometime soon.

TRANSFORM DATA STRUCTURE

In `flam3`, a single (x, y) pair of functions in the IFS is created from a series of fixed-form functions that operate on the input coordinate pair. The outputs of each of these are summed together with weights applied. The transform description includes the list of functions to run, information about the weights of the functions, and for many functions, coefficients or other parameters. On the CPU, these transforms are stored as arrays; the coefficients for every transform are present and simply ignored if the transform is not in the list. These arrays take around 8K per transform, and there can be many transforms in a particular

sheep. Even using constant memory, this creates an unacceptable amount of memory usage and traffic for the embedded architecture.

Instead of storing the transforms in this format, the CPU reads the entire set of information about the transforms and pushes it onto a stack. The stream is then read in sequence when on the GPU; each transform is executed in order, and all information is popped from the bottom as the transform is processed. This cut most transforms to 200 bytes, allowing the entire set of transforms to fit in the constant memory cache and allowing multiple transforms to be run at one time. It's obvious in retrospect, but it took me a while to see this strategy.

TRIGONOMETRIC FUNCTIONS

Many processors don't have trig primitives implemented as instructions. If you find yourself in an assembly language and need an arctangent, compute a Taylor series for the function of interest. Be careful to measure the divergence of the series from the function being modeled; for things like a tangent, you may have to clamp the input values to a certain range depending on the size and precision of your series.

STORING TRANSFORM OUTPUT

The brightness of fractal flame images is computed as the log of the density of the points in a sample region. Hence, the bright spots on an image correspond to a set of counters in memory which have been written to hundreds or even thousands of times during the course of a render. Combine a high write density to a particular region of memory, a massive thread count, and enormous delays between reading a memory location, adding to it, and writing the result, and you get one of two things: either a result which can be off by an order of magnitude, or a desperate need for atomic transactions.

A previous approach to this problem involved the latter: simply do every operation on the framebuffer using atomic intrinsics. Unfortunately, this slaughtered performance. Not only was the memory interface crippled by millions of separate I/O requests, the majority of them suddenly became atomic - stalling most of the execution units on the chips. I don't have hard numbers for the performance hit yet, but theoretical calculations showed that the code achieved less than 2% of its expected throughput with atomic writes enabled.

I've thought of a different approach - a considerably more complex one, to be sure, but also one that is much more suited to the GPU's memory model, and will almost certainly yield great gains in practical rendering performance. It may even be possible to achieve real-time performance at near-HD resolutions on GTX 300 cards, opening the door to a new class of sheep visualizations taking their input from real-time data. It will take some time to implement it, but I plan to restart my work as a hobby project over the next few months, and get things ready to test when the new GPUs roll out.

The algorithm addresses the key aspects of the memory system: small working set, high latency, no externally-controlled consistency, and efficiency gains with contiguous requests. It is conceptually simple: instead of writing the results of the computation - a coordinate pair and a color value - directly to the frame buffer, pack them into a 32-bit int and write them into a log. When the log is full, hand it off to another thread. This thread will read the log and split it, moving the contents of the log into one of 32 smaller logs, each corresponding to 32 subdivisions of the image. Another thread will take these logs when they get full, further dividing them until each log corresponds to an image area of 256 pixels. 256 four-color pixels, at four bytes per pixel, gives a total memory size of 4,096 bytes. This is small enough to fit three 256-wide thread

blocks onto a multiprocessor at a time, ensuring 75% occupancy of each GPU multiprocessor, giving the device enough threads to work effectively without stalling for memory.

Here's the gotcha: there's no central coordination mechanism on NVIDIA GPUs. While they run, each block of threads can communicate to other threads in the block of memory using shared memory, and with other blocks using main memory, and that's it. So, while this simple strategy takes the memory delay from $O(N)$ to $O(\lg(N))$, it also involves writing a memory allocator and threading library from scratch, in assembly, without a debugger and using nothing but a few atomic intrinsics. It's not an impossible task by any means, but it is a significant challenge. I'll have more details on this process as I go about it.

MOVING FORWARD

I'd like to finish this up sometime. I'll be investing in a new system to coincide with the upcoming release of new graphics cards sometime in the future, and I'll also be performing a lot of heavy lifting on GPUs and DSPs as I work on my thesis. I will investigate OpenCL as well; there's no doubt it would be easier to code this thing in a high-level language¹.

I'll keep you posted.

¹God help me, I just called a C-based language "high-level".