Fractal: A Software Toolchain for Mapping Applications to Diverse, Heterogeneous Architecures

Jeremy W. Sheaffer and Kevin Skadron
Department of Computer Science, University of Virginia
{jws9c,skadron}@cs.virginia.edu

December 4, 2011

Abstract

We present Fractal, an API for programming parallel workloads on multithreaded and multicore architectures. Fractal schedules threads on processors with cache configuration in mind in order to minimize cache misses. Fractal achieves speedups of nearly $2\times$ on Sun NIAGARA, while processors with less hyperthreading see less impressive performance improvements.

1 Fractal

Fractal is an API for parallel workloads, which takes into account physical processor configuration and utilizes processor affinity to schedule work with the goal of minimizing L1 cache misses. The API employs a variety of schedulers, designed to minimize cache contention in different ways and for different data access patterns. With hyperthreaded processors executing one thread per virtual core, our schedulers lead to modest speedups of 10%–20%, while our system, executing on a Sun NAIGARA T-1000, an in-order multicore processor with 4 virtual cores per L1, can demonstrate nearly $2\times$ performance improvements.

Fractal provides a write-once, run-anywhere abstraction which, while currently targeting only multicore or hyperthreaded CPUs, is designed with heterogeneous targets in mind. The Fractal framework supports the notion of heterogeneity both in a multinode cluster and, more importantly in the short term, in an environment that supports one or more GPUs, and can be easily extended to work in a networked, heterogeneous cluster.

Fractal's source code is available for download at `http://www.cs.virginia.edu/~jws9c/fractal/`.

1.1 Fractal API

Fractal provides a minimalist API to parallelize independent iterations of loop bodies. In this section we discuss the entry points and their implementations.

1.2 Components

Table 1 contains a complete listing of Fractal API entrypoints. We discuss the functionality-critical members below.

- **`fractal_new()`** and **fractal_delete():**

  `fractal_new()` creates a new fractal context and returns an opaque index in `handle`. A valid index is required for all other fractal API entrypoints (except the error handlers), so `fractal_new()` must be called before any other fractal functions. Multiple simultaneous fractal contexts may exist concurrently; however, to launch simultaneous fractal computations would be unwise, as they would compete for resources. This functionality exists primarily to allow future extension of the API.

Function Prototype	Description
`new(handle_t *handle)`	Create a new Fractal context
`kernel?d(handle_t handle, kernel?d_t kernel)`	Register a ?-D kernel, where '?' is 1, 2, or 3
`scheduler_function(handle_t handle, scheduler_t scheduler)`	Select a scheduler
`loop(handle_t handle, dimension_t dimension, intptr_t initial, intptr_t less, intptr_t stride)`	Describe a loop control
`delete(handle_t handle)`	Free resources related to a context
`launch(handle_t handle)`	Begin a calculation
`get_error(void)`	Check error state
`clear_error(void)`	Reset error state
`barrier(handle_t handle)`	Global barrier
`partition_barrier(handle_t handle)`	Per-L1 barrier
`finish(handle_t handle)`	Wait for calculation to complete
`print_error(FILE *stream)`	Display error information
`override_cpu(handle_t handle, char *cpu_name)`	Select a different CPU from the DB

Table 1: A listing, with brief descriptions, of all of the entrypoints into the Fractal API. All entrypoints and type names are prefixed with `fractal_` (not shown), as are non-standard types. All entrypoints return a `fractal_error_t` value, except for `fractal_print_error()`. `intptr_t` is a signed integer type (from `<stdint.h>` in C99) with sufficient width to store a pointer on the host architecture.

`fractal_delete()` releases the resources associated with `handle`. It is an error to attempt further operations relative to `handle` after calling `fractal_delete()`.

- **`fractal_kernel?d()`:**

  Each of `fractal_kernel1d()`, `fractal_kernel2d()`, and `fractal_kernel3d()` are used to associate callbacks with compute kernels with `handle`. The types `fractal_kernel?d_t` are function pointer types which take one, two, or three `intptr_t` (see caption on Table 1) arguments, respectively.

- **`fractal_scheduler_function()`:**

  The user calls `fractal_scheduler_function()` associates a scheduler `sched` with `handle`. Currently `fractal_scheduler_t` is opaque; however, we envision the possibility of exporting the definition to allow users to implement custom schedulers. There are currently four schedulers available. Details follow in Section 1.3.3.

- **`fractal_loop()`:**

  `fractal_loop()` is used to describe `for` loop-style loop control data and associate it with `handle`. Fractal currently supports up to three loop dimensions (specified with `dimension`). Extension to higher dimensionality is possible, should user response signal it is needed. Work is partitioned by the scheduler according to the loop specifications. The associated kernel is called once per innermost loop iteration, as if the body of that loop.

- **`fractal_launch()`** and **`fractal_finish()`:**

  `fractal_launch()` signals the Fractal runtime that the context associated with `handle` is fully specified and the computation is ready to begin. The runtime does final initialization, work scheduling, and thread creation (see Section 1.3.2). The main thread continues asynchronously.

`fractal_finish()` synchronizes on the termination of all associated threads. A program should not call `fractal_delete()` or attempt to terminate without first calling `fractal_finish()`.

- **`fractal_barrier()`** and **`fractal_partition_barrier()`:**

  These functions are special in that they can and must be called from a fractal kernel. `fractal_barrier()` creates a global barrier. No thread may proceed past the barrier until all threads have reached it. `fractal_partition_barrier()` creates a barrier that is local to the set of threads that share a physical CPU (or an L1 cache). Threads local to a different CPU may proceed without consequence.

  Because Fractal is based on POSIX threads, all synchronization and mutual exclusion primitives and operations associated with pthreads are available within Fractal. These include POSIX semaphores and pthread mutexes and condition variables. Indeed, the fractal barriers are based on these primitives. It is safe to mix primitives as needed, however, users desiring more control with pthreads-based, lower-level synchronization primitives will need to take care to avoid deadlock, and these constructs will not necessarily work with all targets.

1.3 Implementation

In this section we discuss some of the details and challenges of our Fractal API implementation.

1.3.1 System Configuration

We faced a number of issues in implementing the Fractal API. Most important among these are unreliable system-level tools for obtaining architecture-level hardware specifications. Linux provides the `/proc` and `/sys` filesystems, which provide information about the currently running system. In this work, we are concerned with the physical configuration of the underlying processor architecture, as well as the mappings of virtual to physical processors, virtual and physical processors to L1 caches, and processors and caches to their system level abstractions. The information provided by the Linux system level interfaces on these topics is usually false. For example, all Intel systems used in our development and testing list the `ht` tag, supposedly indicating that the processor supports hyperthreading. Of these systems, only our Core i7 based machines are actually hyperthreaded. All others falsely report this functionality. Similarly, our Core i7 systems report eight "physical ids" (0–7), one "core id" (0), one "sibling" (ostensibly a count of virtual cores per physical core for hyperthreading), and the existence of one core, while our Core2 Quad systems report one "physical id" (0), four "core ids" (0–3), four siblings, and the existence of four cores; In reality, the Core i7s have four physical cores with 2-way hyperthreading on each core for eight virtual cores and the Core2 Quads have four physical cores with no hyperthreading (thus four virtual cores). There is no way to determine this from the data in `/proc/cpuinfo`, nor from any other data contained under `/proc` or `/sys`. Linux also provides the `sysconf()` interface to query system state, but this is limited in functionality and actually reads `/proc/cpuinfo` for its data.

To get around these issues, and also to allow the use of our API on systems that do not provide the same system information interfaces as Linux (our Solaris port, for example), we have developed a CPU database as part of our implementation. Our database is indexed first by hostname. If that fails the system attempts to read `/proc/cpuinfo` and uses the CPU model name data contained there (this information seems to be correct) and uses it to index the database a second time. If the second lookup also fails, the system will fall back on a default configuration which assumes a single physical core with two virtual cores.

Our CPU database allows uses to specify all of the desired data described above, as well as a number of other details about caches. Having discovered the issues with `/proc/cpuinfo`, we were able to get actual device specifications from marketing materials; however, that is still not sufficient to map a certain virtual processor to a certain L1 cache. Intel provides a tool which was able to simplify this step on the Intel-based Linux systems[1]. Under Solaris, we were forced to microbenchmark.

The need for a CPU database provided an unforeseen benefit: we were able to easily extend our API with an entry into the processor database that allows override of CPU configuration state. This is useful in its ability to allow a "debug" cpu. Our debug configuration defines only a single virtual processor. This is not generally useful for API development, as the difficulty in developing the API is primarily related to the nondeterminism and locking involved

[1] `http://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration/`

in multithreaded and reentrant code. For application developers, however, this allows debugging of their applications in a single-threaded environment without "hacking" anything.

1.3.2 Initialization and Execution

At API initialization, the processor database is queried for a processor specification matching the current system. With this information in hand, the Fractal runtime can allocate its control structures and create threads. In order to enforce the desired scheduling and sharing (Section 1.3.3), the systems explicitly assigns processor affinity when launching threads. Each virtual processor is assigned one thread, except in the case where there is a one-to-one mapping between virtual and physical processors, in which case two threads are assigned per processor.

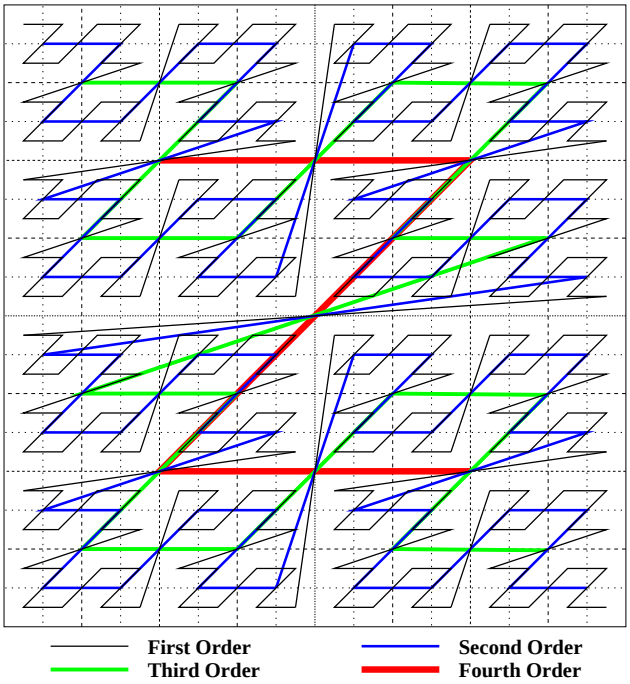

Figure 1: The Morton space-filling curve, with four levels of recursion shown.

1.3.3 Work Scheduling

Before launch, the master thread partitions the work into chunks. The chunks are created in sets of n equally sized work units, where n related to the number of processors (virtual or physical, depending on the exact work scheduler selected). The work partitioner begins by evenly partitioning half the work among the n processors, proceeding recursively until a minimum size work unit is allocated. The last two passes produce units of the same size. The exponential decay in work unit size is designed to reduce the likelihood of experiencing a "long tail" effect, wherein one or a few cores continue to process more compute-intensive workloads long after the majority of cores have become idle. In an ideal situation, every core will process the same slice of work units. There is no mechanism for work stealing built into the API at this time.

Fractal schedules work according to a selection among a number of possible *scheduling patterns*. The Fractal schedulers assume *neighborhood*-based memory access patterns, with tight spatial and temporal locality. Schedulers with more special purpose access patterns, say for a specific implementation of FFT, are feasible. The available scheduling patterns include *naïve*, which divides memory into $2n$ contiguous chunks (where n is the number of available cores and L1 caches; *parallel z*, in which subsequent rows of data are processed simultaneously by threads

sharing core and cache; and *staggered x*, in which threads sharing core and cache stagger accesses to the same matrix row. A *morton* scheduler is under development, which will schedule work based on a Morton space-filling curve shaped access pattern.

The *parallel z* scheduler achieves speedups of $1.79\times$ on NIAGARA while doing a 240×240 matrix multiply and $1.81\times$ on a $3 \times 3 \times 3$ Gaussian blur over $256 \times 256 \times 256$ domain, and more modest speedups on hyperthreaded Intel architectures ($1.12\times$ and $1.01\times$ on the same workloads on a Core i7), when compared with the naïve scheduler. Architecures without hyperthreading are unable to achieve speedups at all.

None of the other schedulers are able to beat *naïve*. We have not fully explored the reasons for the poor performance of these schedulers, though we believe it can be attributed to multiple factors including: out-of-order processing, which seems to explain the better performance of NIAGARA; and the large overhead of OS schedulers, which allows the first of ostensibly cooperative threads to run well ahead before the second thread has a chance to begin. Attempts to enforce closer synchronization the ameliorate this problem increased overhead sufficiently to outweigh their benefit.

2 Conclusions and Future Work

Fractal provides an API that allows users to easily and portably target multicore platforms with many threads of control. Fractal can achieve impressive performance improvements on hyperthreaded architectures. Our results comparing NIAGARA, Core i7, and Core 2 suggest that increased hyperthreading leads to better results for fractal. We would like to explore this more fully.

In addition to hyperthreading, our results suggest that OS schedulers interfere with the fractal schedulers, by allowing threads to run separately that Fractal intended to run together. This results in a competitive situation where it was intended to be cooperative. By modifying the Linux scheduler to be Fractal aware, we believe we could improve Fractal's performance on Linux and show how OS schedulers could be better designed for high performance computing.

Attempting to target GPUs with Fractal proved difficult, as fractal attempts to do fine-grained scheduling of threads, and GPU programming APIs (and indeed GPU architectures) do not allow that. We did not find a satisfactory solution to this problem. Furthermore, the Fractal model of fine-grained work scheduling does not map well to GPU architectures, which automatically schedule threads en masse. Even given satisfactory method of compiling GPU kernels within a Fractal environment, current GPU targets are not likely to see performance improvements from Fractal scheduling.