Work Scheduling on Heterogeneous Resources

Gabriele Keller Edward Pierzchalski

Supervisor

February 9, 2015

1 Introduction

There are two notable trends in computer hardware. First, CPUs are gaining cores per chip while having reduced gains in clock speed. Second, GPUs are developing into generic computing units. As more computing problems are found to be parallelizable, there is an incentive to try and move work that would have been done on the CPU to the GPU, particularly operations on vectors and streaming data.

1.1 Why Scheduling is Painful

There are several obstacles to effectively partitioning, let alone scheduling, a workload over different work processors. The major issues both revolve around memory, namely copy speed and access patterns.

Speed: Inside of a single GPU, data transfer rates can reach upwards of 100 GB/s, and similarly on a single CPU die up to 20GB/s. However, all of these devices are typically connected over a PCIe bus, which caps out at around 8GB/s. This places strong constraints on how frequently data can be shuttled between components. In the current Accelerate architecture, producer/consumer fusion reduces the number of intermediate structures required to perform work, which helps mitigate this issue. However, this fusion can impede attempts to partition work.

Access: Parallelizable computations may have varying degrees of contiguity in their memory accesses. When computing a simple `map` operation, adjacent output locations depend on adjacent input locations, and so we can partition the work by index almost arbitrarily. Similarly, a tree-traversal monoidal fold can, at any given depth, be split over processors. However, any output index of a backpermute can be affected by the value at any index of the input. Although we can parallelise the work, we cannot partition the memory access. (Expand on examples?)

Correctly partitioning and allocating work in the light of these issues will be a key component of this thesis.

1.2 Describing the Problem

Phrase the problem as follows: you have some source code in some language, describing a computation you want performed. It would be nice if, given this source specification, we could partition and schedule the work on to any processing unit available. We will see that not all source descriptions are up to this task.

The most popular frameworks for GPU programming are CUDA and OpenCL, which are both low-level imperative languages. In theory, the problem of partitioning work over the CPU and GPU can be solved at the level of these languages. In practice, even for GPU-focused tasks

such as a monoidal scan or fold, efficient implementations tend to require intimate knowledge of the memory and process architecture of the GPU. This is an added cognitive strain on users of these frameworks, and makes composing GPU tasks difficult: a map followed by a scan is substantially different to just a map, or just a scan.

These languages are also impractical for use as a single source over multiple processors - for instance, while CUDA and OpenCL can target CPU architectures, the resulting binaries are substantially less performant than code originally written with a CPU architecture in mind.

The full semantics of the CUDA and OpenCL languages allow for arbitrary write-effects from any kernel to any array, and include operations such as global barriers. This makes reasoning about kernels difficult for the same reason that hidden state impedes compositional reasoning in many imperative languages. The effect is compounded when kernels are expected to run on separate devices; the result is that these low-level languages make for a poor source description.

1.3 Combinators

The frameworks described previously are distinguished by being *imperative:* they describe, step by step, the actions performed by the GPU. Alternatively, we can describe our program using *combinators*, which abstract the combination of computations. For example, `map` is a combinator that takes a function `f : a -> b`, along with some list `as : List a`, and produces a new list, `map f as : List b`, which contains the result of applying `f` to every element in `as`. We don't particularly care how `map` performs this: we just care that the result is what we expect.

This separation between *what* we want to happen, and *how* we want it to be done, makes combinators quite useful as a source language for computations to be performed, assuming the combinators have 'sufficiently sane' semantics. (yeah this needs a better definition) Once the user describes their program, we can interpret the description into whatever implementation we need to in order to satisfy the semantics of the given program.

1.4 Describing Semantics

The design of a language framework for a specialised purpose (for instance, automatically parallelised vector programming) results in a Domain Specific Language, or DSL. Many DSLs are implemented *inside* another host language, commonly called 'embedding'. There are essentially two flavours of Embedded DSLs (EDSLs):

Shallow Embedding: This is almost synonymous with constructing a library. Terms in the DSL correspond to library functions that, when evaluated, directly perform whatever computations that they represent.

Deep Embedding: Alternatively, we can use the host languages' data definitions to construct a direct representation of the combinators. Say we define a new version of `map`, call it `map'`, which will not be a *function* that turns lists and functions into other lists. Instead, define it as a data constructor: an object containing a function and a list, i.e `map' : (a -> b) -> List a -> DSL b`, where `DSL` is a type representing our DSL embedding. The representation data type is typically called the Abstract Syntax Tree, or AST.

This representation has issues: we can't nest it, for instance, since `map'` takes a `List` yet produces a `DSL`. However, if we change `map'` to take a `DSL`, we will have completely removed lists from our list-description language! The solution is to introduce a way to 'lift' lists into our DSL, which we can do by including a data constructor `lift : List a -> DSL a`. We can then give `map'` the type `(a -> b) -> DSL a -> DSL b`.

In order to do anything with these data structures, we need to interpret or evaluate them. For instance, we can define a recursive function `eval : DSL a -> List a`:

```
eval dsl = dsl match
  lift list          -> list
  map' f anotherDsl -> map f (eval anotherDsl)
```

Benefits of Deep Embeddings: This may seem like unnecessarily complicated overhead, however it gives us the flexibility to do other things with our description of list computations. One of the most important is that once a user has constructed a term in the DSL, we can manipulate it in the host language. For array DSLs, most of these manipulations are for optimisation.

As an example, consider the evaluation of `map f (map g as)`. Semantically, the final result is the application of the composition of `f` and `g` on the elements of `as`. However, since `map` is a simple function, the two calls to `map` will produce two intermediate lists. On a CPU, this is a relatively small (but not ignorable!) overhead; yet if we were to naively send off the work to the GPU as two separate mapping steps (presumably over arrays instead of lists), we would suddenly run into memory bottlenecks.

Compare this with the equivalent deep emedding, `map' f (map' g (lift as))`. Since this is just a nested data structure, we can 'pull out' the functions and compose them, producing a new data structure `map' (f . g) (lift as)`. When we finally interpret this new value, it will be converted into a call to `map` that only makes a single list, avoiding memory bottlenecks.

Accelerate is an EDSL embedded in Haskell, a high-level general purpose functional programming language. We will discuss the fine details of the Accelerate framework in a later section.

2 Background

A large body of work has been produced on generation of parallel code for the GPU and for the CPU [**?**], however less has been done on the problem of doing both for a single general-purpose program. Previous work has investigated dynamic scheduling of a task composed of kernels, however the kernels had to have both CPU and GPU versions provided by the user of the library. (cite) Ideally, we should be able to separate the declaration of our problem from the generation of parallel code to run on whatever hardware happens to be available.

2.1 Accelerate

Accelerate models general GPU programming using a deeply-embedded AST, which is then processed through several optimisation and fusion stages, before being exported to an Accelerate backend. The backend then performs any hardware-specific processing, including compilation (for instance, the CUDA backend embeds the program using skeleton kernel sources).

Take up two pages describing the internals of Accelerate. Note: get a hand on R/T's code so we can see exactly how they go from fissioning the SimpleAcc AST to actually running code on separate processing units.

2.2 Scheduling

Since the AST encodes the semantics of the GPU computation, we can derive the dependency graph of any given part of the program. Given this dependency graph, the problem of scheduling the work onto parallel resources is NP-hard (by reduction to TSP), and so efforts into scheduling tend to focus on heuristics.

A Cheap Trick...

Tiny arrays can go on the CPU!

And an Expensive One

Even purely parallel problems can benefit from also being split onto the CPU. (cite Ryan/Trevor, words about how CPUs can significantly assist a GPU. Note: find out why Ryan/Trevor didn't add the benchmark for the CPU+GPU performance on the Black Scholes stuff.)

3 Previous Work

Obviously going to mention R/T's stuff on implementation, but how much should we talk about scheduling theory? How relevant is it?

4 Requirements

Accelerate should be able to fission a program written in its DSL (already done by Trevor and Ryan - but not automatically?) into 'jobs' that can be scheduled onto a heterogeneous set of workers (CPUs, GPUs) (Need to clarify what I'm adding that isn't just 'making the scheduler better') with a demonstrable performance boost.

5 Work Schedule

As this is partially a development thesis, we must form a development plan. The below schedule details the hopes and expectations for when parts of the development phase will be completed. Included are milestones to be used as checks on whether these expectations have been met. The final report is expected to be written in tandem with the development stages: each week and each milestone has an implicit requirement, 'have something written down about this'. (Need to clarify what parts are research and what parts are development. Also need to clarify what granularity is required other than 'mess with Accelerate to learn about it, then mess with Accelerate to add something to it'.)

Week 1 - 4: Familiarise myself with Accelerate.

Week 1: User-facing DSL.

Milestone: a simple Accelerate program, say a Newtonian gravity simulation.s

Week 2 - 3: Compilation phases.

Milestone: implement a simple phase?

Week 3 - 4: Accelerate backends (mainly LLVM)

Week 5 - 9: Write scheduler.

Week 5 - 6: Have fissioning working on the AST.

Milestone: produce index-partitioned ASTs at kernel generation step.

Milestone: produce ASTs with optimisations for small array sizes.

Week 6 - 7: Emit to both CPU and GPU-facing backends.

Milestone: have above kernels produced and split between CPU and GPU targets, at some static fraction.

Milestone: implement small-array CPU optimisation as part of scheduling.

Week 7 - 9: Implement dynamic scheduling.

Milestone: a runtime utility to determine what backends and processors are available, along with their processing speeds and bus transfer limits (both between and within processors).

Milestone: have task fissioning and task allocation take into account memory dependencies and relative power of processors.

Week 10 - 12: Benchmarks.

Milestone: Compare performance on common problems (Black Scholes, n-body) pre- and post-scheduler.

Milestone: Compare vs. previous scheduler implementation from make formal: Ryan and Trevor, hopefully seeing improvement.

Milestone: Show improvement due to separate stages of scheduler optimisation: small-array CPU scheduling vs. task partitioning.

Milestone: Show improvement due to dynamically informed allocation vs. static strategy.

Week 13 - Due: Buffer time for the inevitable. Clean-up for final report.

References