Work Scheduling on Heterogeneous Resources

Gabi Keller
Supervisor

Edward Pierzchalski

February 8, 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 solvable through parallelism, 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.

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 created equal for this task.

1.1 Low-Level Frameworks

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, these languages make composing solutions impractical - 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 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.

1.2 Fancy DSLs

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`, and some `list : List a` and represents a new list, `map f list : List b`, which contains the result of applying `f` to every element in `list`. We don't particularly care *how* `map` performs this: we just care that the result is what we expect.

This abstraction 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.

When the combinators are expressed, or embedded, as structures inside of an existing language, this style is commonly called an Embedded Domain Specific Language (EDSL). Accelerate is a 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.

1.3 Why Scheduling Things 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 access patterns and copy speed.

Speed: Inside of a single GPU, data transfer rates can reach upwards of 100GB/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. While some array languages allow for arbitrary accesses, combinator-based DSLs like Accelerate restrict computations to those that can be described using 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.

2 Background

A large body of work has been produced on generation of parallel code for the GPU and for the CPU [1], 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.

Accelerate is an EDSL, deeply embedded in Haskell, designed for general purpose GPU programming. However, some parts of computation are better suited for the less concurrent, higher speed processing of the CPU than the highly concurrent processing of a GPU. Examples: generic fold and scan are highly sequential, whereas their monoid versions are not. Map and stencil are iconically ‘embarrasingly parallel’, and so should preferably go on the GPU. (What are the other array DSL constructs? Are there any stateful iterators/transducers? Should they be mentioned here?)

In addition to scheduling tasks based off of computational structure, 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.)

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).

Mention something about figuring out when to do the job fissioning: pre or post fusion? Maybe both? Are we simply fissioning tasks by index and then allocating them accross tasks? Do we do any specialisation based on the processor target (CPU as opposed to GPU)?

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.

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 (dynamically? statically? leave that for later?) 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. (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'.)

References

[1] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 245–256. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=2523756`.