Work Scheduling on Heterogeneous Resources

Gabriele Keller Edward Pierzchalski

Supervisor

February 11, 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 GPU Programming is Painful

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 language over multiple types of 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 foundation when trying to attack this problem.

1.2 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.

Memory Transfer 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.

Memory Access Patterns

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?)

1.3 Making Things Better

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 so automatically for a 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. Correctly partitioning and allocating work in the light of the above issues will be a key component of this thesis.

2 Background

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.

2.1 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.

2.2 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 Embeddings

Shallow embeddings are 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 Embeddings

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'`. Instead of a function, 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 evaluation function `eval`:

```
eval : DSL a -> List a
eval dsl = dsl match
  lift list          -> list
  map' f anotherDsl -> map f (eval anotherDsl)
```

Which essentially 'replaces' `map'` with `map` and lifted lists with their underlying ones.

Benefits of Deep Embeddings

This may seem like an unnecessary and over-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, many 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 transferring the data over the PCIe bus.

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 evaluate this new DSL value, it will be converted into a call to `map` that only makes a single list, avoiding memory bottlenecks.

2.3 Accelerate

Accelerate is an EDSL targeting general-purpose GPU programming. It is embedded in Haskell, a high-level general purpose functional programming language. Accelerate models general GPU programming using a deeply-embedded AST, which is then transformed through a variety of intermediate representations and functions over them. These transformations perform a wide variety of changes aimed at converting the AST from the high-level, user-facing description into a low-level description appropriate for compilation to machine code for the relevant platform. The compilation step is intentionally kept separate from the AST transformations, to make changing compilation targets as straightforward as possible. (does this add overhead? could we do cool things if we screwed with the tree earlier, or provided different KernIR primitives specialised for the CPU?)

Recently, a new backend kit has been published for Accelerate, cleanly separating these compiler stages and exposing a simplified kernel-level representation compared to previous versions of Accelerate.

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.

Aside: A Frustrating Operation

There is one family of array operations in the Accelerate DSL that is particularly non-performant for heterogeneous scheduling. The `permute` and `backpermute` operations use a function from one set of array indices to another in order to permute an array. For instance, `backpermute f old` will produce an array `res` such that `res ! i = old ! (f i)`.

Since the range of `f` is unrestricted, the value at any index of the output array may depend on the value at any index of the input array. This complicates the partitioning of the work: although we can split the task of constructing the resulting array, we can't partition the input array between processors.

In many GPU use-cases, the strongest constraint on throughput is memory transfer speed, which makes copying the entire input array to each processor infeasible. The result is that these

permutations are *de facto* synchronisation points: every operation before them must all have their results returned and combined before the permutation. This has a potentially interesting interaction with data fission: normally, a map composed with a backpermute would be fused together in some fashion. We may ask the question: when is it worth fissioning data for the map before recombining for the permutation, and when is the fused operation preffered?

3 Previous Work

3.1 Scheduling

3.2 Accelerate

3.3 Heterogeneous Scheduling in Accelerate

A recent paper by Newton, Holk, and McDonell [2] describes a first foray into heterogeneous scheduling in Accelerate. Their paper forms the foundation of this project.

4 What We're Going To Do

There is much potential work within the scope of what has been discussed so far. For the purposes of this project, the high-level goal is to have Accelerate 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) with a demonstrable performance boost, while also addressing some of the issues expressed in previous work. (Need to clarify what I'm adding that isn't just 'making the scheduler better') In addition, the following are potential points of focus to provide guidance in achieving the desired result.

Dynamic Fissioning

The data fissioning described previously was always static: the SimpleAcc term would be fissioned into some finite and static number of array segments, and each work component would be sent off to the same processing unit (determined by index).

We can improve on this by determining the degree of fissioning and the allocation of fissioned segments dynamically, depending on factors such as the relative processing speeds of the processors.

Small Array Optimisations

One minor issue not addressed by existing approaches to heterogeneous resource scheduling is the cost of operations on small arrays. Loading, running, and using the result of a GPU kernel has significant overhead. Below a certain array size, the processing time of operations on an

array is dominated by the high latencies of both loading the GPU kernel and data transfer between the CPU and GPU.

In these cases, it would be more effective to perform the operation on the CPU. The existing Accelerate framework does not differentiate workloads by array size, and so does not take advantage of this. Once we implement dynamic fissioning, we can look into including this optimisation.

Extensions

- The current LLVM backend for Accelerate does not support some operations, such as stencils. Expanding support for this feature will allow more comprehensive comparisons of fissioning transformations.
- Dynamic fissioning would be improved if we had a notion of 'device affinity', so that data dependencies in the task graph are not unnecessarily copied between processors.
- For some tasks such as sorting and non-monoidal folds, CPUs may be as fast as GPUs. For this reason, after fissioning we may prefer that a given segment of work be performed on a particular kind of processor. Handling this form of 'task affinity' may provide additional improvements to performance. (Find more examples. Is this relevant? Does Accelerate contain enough operations that are performant on CPUs?) (Note: find out why Ryan/Trevor didn't add the benchmark for the CPU+GPU performance on the Black Scholes stuff.)

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'.)

5.1 Timetable

Week 1 - 4: Familiarise myself with Accelerate.

Week 1: User-facing DSL.

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

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) with and without 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. Extensions, if applicable.

5.2 Measures of Success

For those milestones directly related to construction of the scheduler, there are two 'parameters' for success: depth and breadth.[1] 'Depth' in this context refers to successfully processing an AST through all the relevant compiler passes. 'Breadth' refers to having depth completion across Accelerates' many array operations (maps, folds, permutes, and recently iteration constructs). Cleary depth completion is a necessary prerequisite for most of the desired results concerning

[1] This conceptual divison is not due to us: we rephrase it here from Trevor's paper.

scheduling, however breadth is important for assessing the interaction between task fissioning and compilation optimisations like fusion.

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`.

[2] Ryan R. Newton, Eric Holk, and Trevor L. McDonell. Converting data to task-parallelism by rewrites. URL `http://www.cse.unsw.edu.au/~tmcdonell/papers/acc-multidev-icfp2014-sub.pdf`.