Work Scheduling on Heterogeneous Resources

Gabriele Keller Edward Pierzchalski

Supervisor

February 13, 2015

Contents

1 Introduction

In recent years, two trends in computer hardware design have become apparent. Firstly, the number of cores per CPU is increasing, while the rate of increase in clock speed is decreasing. Secondly, GPUs are developing into generic computing units, with APIs that expose more of the power of their architecture. 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. At the same time, research suggests that the gap between CPU and GPU processing rates is not as high as one may expect. [1, 2] This gives an incentive to design languages and frameworks to ease the division of work over both the CPU and GPU.

1.1 Why GPU Programming is Painful

Two of 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 tree-traversal scans, sorts, or folds, efficient implementations tend to require intimate knowledge of the memory and process architecture of the GPU. Harris et al. [3], Satish et al. [4] 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. [1]

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 transfer speed and access patterns.

[1] See Jie Shen et al. [5] for a discussion focusing on OpenCL.

Memory Transfer Speed

Inside of a single GPU, data transfer rates can reach upwards of 300 GB/s. [6] However, CPUs and GPUs are typically connected over a PCIe bus. Modern PCIe busses have a bandwidth cap of around 8GB/s. [7] This places strong constraints on how frequently data can be shuttled between components.

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.

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, [8] 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. [9] 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 we want to perform. 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 operation 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. [10] 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. This example is called 'map fusion', and is a particular instance of a family of similar 'fusion' optimisations designed to reduce the number of intermediate lists (or arrays) involved in a computation.

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.

All levels of the Accelerate framework make heavy use of advanced features of the Haskell compiler. Most prominently, GADTs and type families are used to encode many properties of the AST, such as the shape of arrays, and levels of parallelism (these advanced type annotations are left out of this report for brevity). Other features include the use of De Bruijn indices to capture lambda abstraction, and also to facilitate term reuse. However, despite how interesting all these properties are, they mostly form a backdrop to our main focus, the compilation phases.

The final, machine-code-generating compilation step is intentionally kept separate from the AST transformations, to make changing compilation targets as straightforward as possible. 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.

2.4 Fission

One of these changes introduced a new pair of combinators to the language, `split` and `concat`. They have the intuitive semantics that their names imply, of splitting and concatenating arrays. The combinators are used by the compilation phases to 'fragment' arrays and operations on them. For instance, `map f arr` may be translated[2] into

```
let (x, y) = split arr in
  concat (map f x) (map f y)
```

When later stages in the compiler encounter the `concat` node, they may schedule the two arguments to `concat` to run on different processors before combining them. In the context of work division over processors, this process of splitting arrays is called 'data fission'. [11]

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. As an example, consider the operation `new = backpermute f old`. The function `f` is a mapping from the indices of `new` to the indices of `old`. This will define `new` such that `new ! 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 work: for instance, say we divide the computation of the output array to two processors. Although each processor can work in isolation, both need access to the entire input array in order to perform the operation.

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 preferred?

2.5 Scheduling

Scheduling strategies can be roughly split into two categories: static and dynamic. [12] Like with static and dynamic typing, the categories of scheduling are best described by what we know, and when we know it. In this case, we're interested in where a given chunk of work is going to be processed - on the CPU, or the GPU, or maybe some remote cluster.

[2]This is pseudocode: the details of the Accelerate DSL are omitted for brevity. In particular, in the Accelerate DSL, the `split` and `concat` nodes contain the indices on which they were split, and only have convenient semantics when the two nodes act on the same index. Example drawn from Newton et al. [11].

In the static case, we know at compilation time where a particular step of the workload will be performed. For example, after operator fusion optimisations, the Accelerate backend currently splits arrays into some constant number of fragments. These fragments are then partitioned and scheduled across processors in part by analysing the dependencies of the various workloads. However the primary determinant in partitioning is a single number: what fraction of the fragments are sent to the GPU, with the rest being sent to the CPU.

In the dynamic case, the processor on which a given work fragment is performed can only be determined at runtime. This is usually because the scheduler uses some information that is only available at runtime. This includes performance metrics for the different processors, and the completion time of earlier fragments.

3 Previous Work

3.1 Heterogeneous Scheduling

Skynet Compiler

Grewe and O'Boyle [13] use machine learning over a feature space of OpenCL source code to determine how to partition work over processors. The learning-based model is trained on the static code features of OpenCL programs, along with their optimal work partitions. The trained model is used to categorise subsequent OpenCL programs by their static code features. This categorisation subsequently guides the partitioning of work. However, the method by which the compiler separates work is not specified.

In addition to the novel methodoloy, the paper finds very direct empirical evidence of the importance of work partitioning. Figure 2 of the referenced paper displays the performance of several benchmarks against the ratio of work split between the GPU and CPU. The heterogeneity of the results validates the search for effective, kernel-dependant workload partitioning.

Binary Analysis

In work by Lee, Samadi, Park, and Mahlke [8], the authors construct a framework that takes a users' data parallel OpenCL kernel and partitions the work across devices. This is achieved through hooking into the OpenCL API layer to expose a single, large device.

Kernels are split at runtime using a decision tree heuristic, using approximations on the transfer costs of moving data between processors, and also the variance of performance of processors. These parameters are estimated by the partitioning system, however it is not mentioned what the method of estimation is.

To determine whether a workload can be safely separated to several devices, data flow analysis is used on array indexing. If the data flow analysis does not sufficiently guarantee safe array access patterns, the kernel is considered contiguous and thus not parallelisable. This is an example of where a higher-level abstraction may have assisted in splitting the workload. These

are both potentially interesting directions for research on improving Accelerate, however they are beyond the current scope.

Dynamic Profiling

Wang, Zheng, Chen, and Guo [9] review several heterogeneous scheduling algorithms (which they refer to as 'co-scheduling'), and compare against their own novel algorithm. Although static scheduling is discussed, the focus is on methods that use dynamic profiling to achieve workload balance across processors. Two competing dynamic scheduling algorithms are described: *quick scheduling* and *split scheduling*. All of the dynamic scheduling algorithms discussed rely on the workload being split into some collection of iterations or fragments beforehand.

Quick scheduling uses static scheduling on some small, initial part of the workload. By measuring performance in this first stage, the rest of the workload is appropriately partitioned between processors. The authors remark that, since the performance of the GPU may depend on the workload, this single initial profiling step may lead to a partitioning of work that is inefficient for the rest of the program. This is similar to one of the initial scheduling plans for this project.

Split scheduling is a form of 'iterated quick scheduling': the profiling data from one chunk of workload is used to partition the workload for the next chunk. Since data is synchronised between each work division, there is significant overhead due to the increased frequency of profiling relative to quick scheduling.

The novel algorithm proposed by the authors (Co-scheduling based on Asymptotic Profiling, or CAP) lies between quick and split scheduling. In particular, workload chunks are only profiled if performance has changed substantially. Otherwise, chunks are scheduled in progressively larger increments, helping to reduce synchronisation overhead.

In their results, the authors find that their algorithm is competitive with the other dynamic scheduling algorithms in terms of overall speed. In addition, they find a reduction in the spread between GPU and CPU execution times. The implementation of these varied algorithms may be a valuable extension for our thesis, or a future project.

The contributions of the paper are purely towards the scheduling algorithm. The code to run workloads on the GPU and CPU was written by hand, with the scheduler taking the form of 'glue code' in-between. Although all the discussed algorithms rely on the workload being split into a large number of 'iterations', it is not specified in the paper how this workload splitting is performed. The automation of these steps is an important part of the value of frameworks like Accelerate.

3.2 Parallel DSLs

Data Parallel Haskell

Data Parallel Haskell (DPH) is a generic parallelism extension to the Glasgow Haskell Compiler. For DPH, the focus is on *nested data parallelism*, exploring and expanding on the work of the NESL language. [14] While Accelerate uses some features of the Haskell type system to statically *forbid* nested parallelism, DPH instead adds *flattening* to the collection of representation transformations, converting nested parallel operations on a nested array into scalar operations on a flat array.

Copperhead

Copperhead is a GPU programming EDSL hosted in the Python programming language. [15] Currently, it targets the CUDA architecture. It has a similar set of combinators to most array DSLs. Since it is hosted in Python, expressions are untyped until executed, which leaves many potential common programmer errors unchecked until runtime.

Similarly to DPH, Copperhead supports nested parallelism. However, while DPH follows in the tradition of handling nested parallelism through flattening, Copperhead attempts to follow the heirarchial parallelism of modern GPU architectures. In particular, the CUDA programming model has a heirarchy of *kernels* running *thread blocks*, which in turn contain *device threads*. When possible, nested parallelism in Copperhead is mapped onto this heirarchy.

For fusion and similar optimisations, the compiler must perform 'shape analysis' at runtime to determine the compatability of array sizes. This contrasts with Accelerate, which uses Haskell's typeclass features and strong typing to ensure shape compatability.

Copperhead has limited support for targeting heterogeneous systems. Devices must be declared explicitly, and any given computation targets a single device, implying that any scheduling must be done by the user.

Dandelion

Dandelion is a .NET extension offering a compiler and runtime for heterogeneous systems, targeting the F# and C# languages. [16] LINQ is essentially a DSL within the .NET framework for describing operations over iterable collections in a monadic fashion. Dandelion supports lifting most LINQ operations to the GPU. It does so by using the Common Compiler Interface to convert .NET bytecode to CUDA source.

This binary analysis step has similar issues to those encountered by Lee et al. [8]. In particular, converting heap allocations to stack allocations requires data flow analysis to construct a static size for the original allocation. If an allocation cannot be statically sized, the conversion fails. The binary translation process also makes assumptions about purity of user-defined functions, which are not checked. Combinator optimisations, like map fusion, are not implemented.

Dandelion uses explicit reasoning on a data-flow, task-dependency graph to implement

heterogeneous scheduling and work division. The graph is constructed through an intermediate engine called PTask, which describes various primitives for encoding iterative processes between data sources and sinks. Many of the concepts are similar to those from stream processing; future work may try and translate some subset to Accelerate.

In addition to targeting CPU and GPU backends, Dandelion also supports distributed cluster computing as part of its model of heterogeneous systems. This is an interesting and natural extension of the concept of 'write once, parallelise everywhere' described by Newton et al. [11].

3.3 Heterogeneous Scheduling in Accelerate

A recent (at the time of writing, unpublished) paper by Newton, Holk, and McDonell [11] describes a first foray into heterogeneous scheduling in Accelerate. This is also the work that led to the previously described improvements to Accelerate - the separation of AST transformation phases, the improved final kernel representation, and the separation and abstraction of the hardware backends. In addition to these contributions, in their paper Newton et al. introduce data fissioning as described previously, and explore two methods of scheduling tasks over the fissioned data.

Bulk-Synchronous Parallel Tasks were fissioned into some constant number of segments, which were in turn round-robin allocated onto processors. Scheduling was performed in batches, and tasks were fissioned in dependency order.

This first method had no notion of device affinity, and so ran into memory bottlenecks since data would be frequently copied from processors to the host before the next batch was allocated. In addition, the batch allocation added significant synchronisation overhead between rounds. These deficiencies resulted in extreme performance degredation, and the authors do not persue the method further (which is a hint that we should not either).

Single Program, Multiple Data As an alternative strategy, the authors use the dependency graph to ensure that at least some of the dependencies for a fragment are all allocated to the same device. Data fissioning occured in the same manner as with the bulk-synchronous method.

The enforcement of the scheduling semantics of fissioning were split between both the fissioning stage of AST processing, and in the final device scheduler. Namely, the fissioning algorithm is what made decisions about work ratios and so expected, for example, the first fraction of the segments to go on one device and the rest to another. This was separately reflected in the scheduler as well. This latter method forms the foundation of this project.

4 Work Plan

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 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. In addition, the following are potential points of focus to provide guidance in achieving the desired result.

4.1 Dynamic Fissioning

The data fissioning described previously was always static: the low-level representation 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.

4.2 Small Array Optimisations

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.

4.3 Extensions

Below is a list of potentially valuable additions to the goals of this project. Since they are not critical to the goal, they will be persued only if time permits.

Improved Concatenation Currently, `concat` nodes are converted into array generators using conditional indexing on the segments. For instance if we have `b = concat b1 b2`, and if `b1` and `b2` are split at some index `s`, this will be translated into

```
b = generate (\i -> if (i < s) then (b1 ! i) else (b2 ! i))
```

Although there are situations where the intervening `generate` may be removed by inlining or through indexing with a constant value, performance may be improved otherwise by adding memory-copying primitives to the lower-level array representations.

LLVM Support 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.

Device Affinity 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.

Processor Affinity For some tasks such as sorting and general 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.

4.4 Initial Technical Plan

The existing framework relies on the scheduling backend reacting to `split` and `concat` nodes in the final representation, and splitting work accordingly. Currently, the process of introducing these fragmenting nodes occurs in a single phase, after high-level map fusion. Investigating the interaction of data fissioning with other operations and optimisations will involve separating out the fusion steps, so as to insert fissioning between them. Adding information about available processors will require providing some data to most of the compilation steps: we can achieve this with something like the reader monad, if needed. Adding affinities may involve adding metadata to the fissioning nodes themselves.

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'.

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 metadata-generating phase, as a test of understanding.

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: (Extension) have task fissioning and task allocation take into account memory dependencies and relative processing speed 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 Newton et al. [11], hopefully seeing improvement.

Milestone: Show improvement due to separate stages of scheduler optimisation: small-array CPU scheduling, task partitioning, preventing permute fusion.

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

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

5.2 Measures of Success

For those milestones directly related to construction of the scheduler, there are two 'parameters' for success: depth and breadth.[3] '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 scheduling, however breadth is important for assessing the interaction between task fissioning and compilation optimisations like fusion.

[3]This conceptual divison is not due to us: we rephrase it here, but it is originally from Newton et al. [11].

Bibliography

[1] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, and others. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, . URL http://dl.acm.org/citation.cfm?id=1816021.

[2] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5762730.

[3] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. 3(39):851–876. URL http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf.

[4] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. URL http://www.nvidia.com/docs/io/67073/nvr-2008-001.pdf.

[5] Jie Shen, Jianbin Fang, H. Sips, and A. L. Varbanescu. Performance traps in OpenCL for CPUs. pages 38–45. IEEE. ISBN 978-1-4673-5321-2, 978-1-4673-5321-2, 978-0-7695-4939-2. doi: 10.1109/PDP.2013.16. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6498531.

[6] NVIDIA. GeForce GTX 780-TI specifications, 2015. URL http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780-ti/specifications.

[7] PCI-SIG. PCI Express 3.0 frequently asked questions, 2015. URL https://www.pcisig.com/news_room/faqs/pcie3.0_faq.

[8] 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.

[9] Zhenning Wang, Long Zheng, Quan Chen, and Minyi Guo. CPU+GPU scheduling with asymptotic profiling. 40(2):107–115. ISSN 01678191. doi: 10.1016/j.parco.2013.11.003. URL http://linkinghub.elsevier.com/retrieve/pii/S0167819113001415.

[10] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *Acm Sigplan Notices*, volume 48, pages 497–510. ACM. URL http://dl.acm.org/citation.cfm?id=2429128.

[11] 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.

[12] M. D. Jones. Practical issues in OpenMP. URL http://www.buffalo.edu/content/www/ccr/support/training-resources/tutorials/advanced-topics--e-g--mpi--gpgpu--openmp--etc--/2011-09---practical-issues-in-openmp--hpc-1-/_jcr_content/par/download/file.res/omp-II-handout-2x2.pdf.

[13] Dominik Grewe and Michael F.P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction*, pages 286–305.

[14] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM. URL http://dl.acm.org/citation.cfm?id=1248652.

[15] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. 46(8):47–56. URL http://dl.acm.org/citation.cfm?id=1941562.

[16] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. pages 49–68. ACM Press. ISBN 9781450323888. doi: 10.1145/2517349.2522715. URL http://dl.acm.org/citation.cfm?doid=2517349.2522715.