Converting Data to Task-Parallelism by Rewrites

Purely Functional Programs across multiple GPUs and CPUs

Ryan R. Newton Eric Holk
Indiana University
{rrnewton,eholk}@indiana.edu

Trevor L. McDonell
University of New South Wales, Australia
tmcdonell@cse.unsw.edu.au

Abstract

High-level domain-specific-languages for array processing on the GPU are increasingly common, but to date they run only on a single GPU. We argue that languages will need to target multiple devices, even simultaneous combinations of GPU/GPU and CPU/GPU. Increased flexibility may be key to making these languages more easily deployable and thus widespread. To this end, we present a compositional translation that fissions data-parallel programs in the *Accelerate* language, allowing subsequent compiler stages to map computations on multiple devices via different code-generation backends. As a result, Accelerate becomes the first EDSL to exercise the CPU and GPU with each data-parallel kernel. Further, because Accelerate code is written at a level of abstraction that does not require per-platform tuning, the same source can run efficiently on and across different devices.

Categories and Subject Descriptors D.3.2 [*Concurrent, Distributed, and Parallel Languages*]

General Terms Languages, Performance

Keywords GPU, Multi-device, Haskell, Data-parallelism, Scheduling

1. Introduction

The opportunity cost of ignoring vectorized arithmetic (SIMD instructions) is rising. Systems from cell phones to servers now have multiple discrete processors, including CPUs, GPUs, and other co-processors, *all* of which include SIMD units on-chip. Recent work in the architecture [37], and PL communities [20], suggests that utilizing all devices on a platform typically yields a significant speedup over using one device type (*e.g.* GPU or CPU). This is consistent with recent findings that the throughput of multicore/SIMD CPUs is often a non-trivial fraction of GPU throughput on real applications [26, 38].

From a language perspective, we would like to be able to target these heterogeneous platforms with a single language. Indeed, this has been our goal with earlier efforts in stream-processing languages [44, 45], and in our recent attempt at a Haskell-based *work-stealing* framework that crosses CPU and GPU [20]. The latter approach chose between a [separate] GPU or CPU implementation of each task, a pattern familiar from other heterogenous scheduling systems [37]. In this paper, we go further and target multiple devices from a *single*, data-parallel, purely functional program—that is, not just compiling the same program for multiple platforms (portability), but *fissioning* the program and assigning components to different devices at runtime.

This partitioning approach runs into difficulties for languages that include mutable data structures and side-effects within their data-parallel code (Section 2.1.2). *Purely functional* array languages such as Accelerate [15, 41], Nikola [40], Copperhead [14], and Intel ArBB [43], appear to have significant advantages for this purpose. Of course, they have their own challenges as well: in compiling these languages to vectorized code it is critical to employ aggressive *fusion* [19, 25, 41] to eliminate intermediate data-structures. As we discuss in this paper, the fusion imperative can be at odds with multi-device partitioning.

This paper takes a first step towards a multi-device implementation of Accelerate, an embedded array language in Haskell that was developed originally to target GPUs. We begin by addressing a prerequisite: adding an Accelerate backend *for CPUs*. Our new CPU backend generates vectorized, threaded code by emitting C code with the Cilk Plus language extensions[1]. The vast majority of new code written for this backend is part of a new *middle-end* optimization pipeline[2] that uses the *micropass compiler* approach (Section 4) and can be used with any backend (Section 4).

Our second step is to build an Accelerate *multi-device backend*[3], which can dynamically split work across across any mix of multiple GPUs and multiple CPUs. This means both partitioning operator DAGs between devices (task parallelism) and fissioning individual data-parallel operators to expose *more* parallelism, syntactically.

In this paper, we make the following contributions:

- We describe the first array EDSL to automatically split (*fission*) individual operators to work across devices.
- We formalize a non-deterministic term-rewriting system that captures the rich optimization/fissioning space for Accelerate programs, while ensuring *any* strategy for navigating that space will produce valid programs for the compiler backend. We explore several scheduling strategies expressed purely as source-to-source syntax transforms (Section 3).

[1] Cilk extensions are supported by the Intel C++ compiler, GCC 4.8 (on a branch), and partially supported in CLang, as of this writing. Our compiler also has nascent support for OpenCL, which has CPU as well as GPU implementations, as described in Section 2.1.1.

[2] `https://github.com/AccelerateHS/accelerate-backend-kit`

[3] `https://github.com/iu-parfunc/accelerate-multidev`

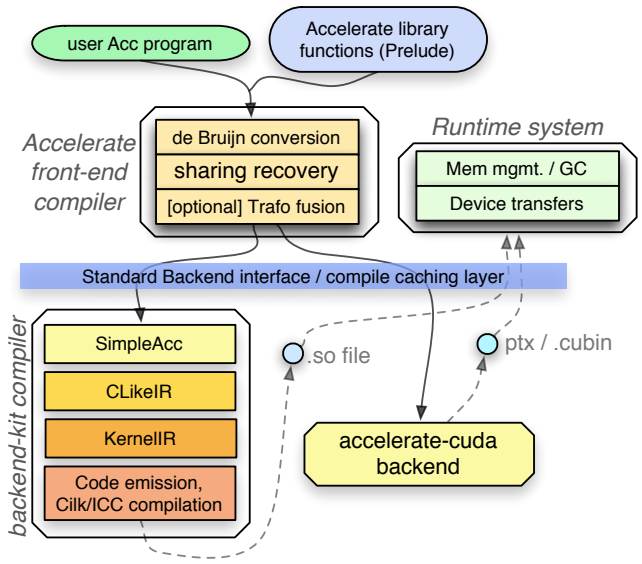


Figure 1. Architecture of the extended Accelerate system. Section 4 describes the backend-kit compiler components; Section 5.1 describes the interface between the front end and multiple backends.

- We present the first CPU-backend for Accelerate, summarize the optimizations that enable it, and compare its generated code to the existing Accelerate CUDA backend (Section 6).
- We describe the design of our Accelerate *backend-kit* (Section 4), which lowers programs to a very simple representation; includes a range of (backend-agnostic) compiler optimizations; and includes infrastructure for runtime services such as compilation caching. The end result is to greatly reduce the marginal cost of adding *future* Accelerate backends, which will pave the way for supporting more devices and heterogeneous configurations.

2. Preliminaries

2.1 General purpose GPU computing

Modern graphics processing units (GPUs) are massively parallel processors optimized for workloads with a large degree of SIMD parallelism. Despite the advertised potential of 100× speedups, attaining good performance requires highly idiomatic programs that are work intensive and require expert knowledge.

The most popular frameworks for programming GPUs are CUDA [47] from NVIDIA, and its open-standard competitor, OpenCL [35]. Both are extensions of the C programming language that include support for defining GPU *kernels*, which contain code executed by many data-parallel threads on the GPU. These threads are arranged in a multidimensional structure of *thread blocks* and *grids*, and executed in SIMD groups called *warps*. Threads must be programmed so that they make efficient use of both the *global memory* region in off-chip DRAM, as well as the on-chip *shared memory* region, a software managed cache that can be used for efficient intra-block communication. All this and more must be managed by the programmer in order to ensure good use of a GPU's hardware resources [47].

Although the programming models for CPUs and GPUs are quite different, as GPUs continue to gain general-purpose capabilities and CPUs continue to acquire more powerful vector instruction sets, we are seeing a convergence between CPU and GPU architectures. This suggests that a single unified programming model for both CPUs and GPUs is achievable, and that it should be possible to write code to run well on the GPU and also run well on the CPU with minimal changes. The primary architectural differences at this point are memory bandwidth, the width of the vector processing units, and the trade-off between increased parallelism or improved single-threaded performance.

2.1.1 Device portability: language issues

Unfortunately, even the limited goal of *portability*—a prerequisite to fissioned multi-device execution—can be hampered by the inclusion of certain language features in CUDA and OpenCL, which are both low-level, imperative programming models (*e.g.* barriers and unrestricted writes). Nevertheless, both OpenCL (and recently CUDA [1, 3]) *do* have options available for targeting x86 processors. However, it is widely recognized that OpenCL code should be changed when re-targeting to CPU [53]. First, the *task-parallel* OpenCL programming model is most appropriate only for CPU code. Second, these low-level programs commit to specific memory access patterns which are difficult to optimize for the very different memory architectures: cache-friendliness on the CPU, versus coalesced operations on the GPU.[4] Third, the prevalence of barriers in the OpenCL and CUDA programming models makes it easy to write code that is inefficient on the CPU. For example, it is common for large numbers of threads (e.g. 1024) to synchronize at a barrier before proceeding. On a GPU such operations are efficient, but on a CPU this is a large amount of thread state and is detrimental to performance.

2.1.2 Multi-device partitioning: language issues

Once portability is achieved, then the next step is simultaneous use of multiple devices. The "single kernel, multiple device" (SKMD) approach has already been attempted with the OpenCL language [37], where it can yield improved performance. However, this is challenging as OpenCL allows arbitrary side effects from any data-parallel kernel to any array. As a result, OpenCL-based partitioning systems take one of three approaches: (1) avoid partitioning kernels that use writes; (2) attempt a static analysis to identify the memory access patterns [37]; or (3) use a runtime technique to *merge* writes from multiple threads in different memories.[5]

Even with these techniques to handle kernel side effects, matching the semantics of OpenCL operations like global barriers and atomic operations has proved infeasible. For example, Lee et. al. [37] chose to simply ignore these features, Moreover, it is unclear whether the full OpenCL language can ever be a suitable target for multi-device partitioning.

By contrast, several recent array-based languages targeting GPUs provide only *immutable* data [14, 15, 40, 41, 43]. These languages usually still allow reads at arbitrary (computed) array indices, which poses challenges for multi-device distribution. However, because of immutability, array dereferences in these languages are referentially transparent, so replicating the same array in multiple memories is a viable option. Furthermore, these languages employ a combinator-based style of programming using operations such as `map` and `fold`, which de-emphasises the need for arbitrary array indexing in favour of implicit data-access patterns.

[4] Memory operations to consecutive addresses by consecutive threads are on-the-fly transformed into single, wide loads or stores. Not using coalesced memory accesses can result in an order of magnitude loss of effective memory bandwidth [47].

[5] The technique has proved useful in many domains [12, 39], but has both significant runtime overhead and, in the OpenCL case, relies on relaxed memory consistency.

	Accelerate	Intel ArBB	Copperhead	Data Parallel Haskell	APL
Mutable Arrays	no	no	no	no	yes
Arbitrary Reads	yes	yes	yes	yes	yes
Multidimensional Arrays	yes	yes	yes	no	yes
Sparse/Segmented Arrays	yes	yes	no	yes	no
Nested Parallelism	no	yes	yes	yes	yes
Recursion/Iteration	yes	no	yes	yes	yes
Array-lvl Conditionals	yes	no	yes	yes	yes

Table 1. Comparison of the features of different array-oriented languages, which range from narrowly domain specific to more general purpose. More restricted languages generally enable better auto-parallelization at the cost of expressiveness. Some languages differentiate between the scalar and kernel language; in those cases we report the features for the kernel language.

This leaves us in a good position to begin to explore executing these languages on multiple, distributed memory devices.

2.2 Array DSLs, generally

Array-oriented languages have been around for a long time, including APL [32], Matlab [55], and so on. Even data-parallel languages centered around *high-level combinators* date at least from Blelloch's work in the late 1980s on the *scan vector machine* [8] and NESL [9, 10]. Nevertheless, today's hardware environment has inspired a renaissance. Indeed, it is increasingly practical to generate efficient parallel code from high-level data-parallel descriptions. The last ten years have seen a flurry of activity, with many *array DSLs* (domain-specific languages) targeting CPUs [43], GPUs [14, 15, 40], or either one [50]. There has also been plenty of focus on code generation for more narrow domains, such as stream-processing [45, 56] and for specific algorithms [22, 49, 54]. This trend is bolstered by general improvements in DSL *embedding* (meta-programming), such as improvements in sharing observation[6] and AST representation [4, 5, 13]. There have also been advances in array DSLs specifically, supporting new program transformations and scheduling approaches [14, 43] and techniques for optimized code generation [18, 54].

At its core, a typical array DSL provides a way to compile a pipeline (or DAG) of data-parallel operators—*e.g.* `map`, `filter`, `fold`—into parallel code. Typically, there are no language abstractions separating the operators in the pipeline, enabling the compiler to observe *all* data-flow relationships. This is especially true of *deeply embedded* DSLs, which also add an *extra* code generation phase (either at compile or runtime) in which code for the object language is emitted.

Generally speaking, array languages occupy a spectrum of restrictiveness, with full-featured languages like Matlab and APL occupying one end of the spectrum, and more recent EDSLs occupying the other. The Accelerate EDSL — which we work with in this paper — is towards the restrictive end of the spectrum, disallowing nested parallelism and general recursion. Nevertheless, Accelerate is not the *most* restrictive array language that would be reasonable to implement. For example, it does allow: (1) array values to be returned from conditionals, and (2) arbitrary indexing into arrays. See Table 1 for a feature comparison between array languages.

As with other deeply embedded languages, once the Accelerate abstract syntax tree is extracted during meta-program evaluation, an Accelerate program is effectively a *graph* of data-parallel operators, represented as combinators such as `zipWith` and `permute`. Compiling Accelerate programs amounts to optimising the program and generating code for the target platform, which for Accelerate has been CUDA running on a single GPU. We describe our version of this compilation process in Section 4.

[6] This refers to the recognition of common subexpressions (CSE) in the target language via inspection of meta-language in-heap data structures [23].

2.3 The Accelerate Language

Accelerate [15, 41] is a small language for computations over regular, multidimensional arrays. It is a GPU-oriented EDSL in Haskell that exposes data-parallel combinators for arrays that closely mirror familiar Haskell list-processing idioms. For example, to compute a dot product we write:

```
dotp :: Num n ⇒ Vector n → Vector n → Acc (Scalar n)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in  fold (+) 0 (zipWith (*) xs' ys')
```

The function `dotp` consumes two one dimensional arrays (`Vector`) of values, and produces a single (`Scalar`) result as output. The `Acc` type constructor indicates that the result is an embedded Accelerate computation—it will be evaluated in the *target* language of dynamically generated parallel code, rather than the *meta* language, which is vanilla Haskell.

The arguments to `dotp` are of plain Haskell type `Vector a`. To make these arguments available to the Accelerate computation they must be embedded with the `use` function, which is overloaded so that it can accept tuples of `arrays`:

```
use :: Arrays arrays ⇒ arrays → Acc arrays
```

The functions `zipWith` and `fold` are defined by the Accelerate library, and have *massively parallel* semantics, supporting up to as many threads as data elements. The type of `fold` is:

```
fold  :: (Shape sh, Elt e)
      ⇒ (Exp e → Exp e → Exp e)
      → Exp e
      → Acc (Array (sh:.Int) e)
      → Acc (Array sh e)
```

The type classes `Shape` and `Elt` indicate that a type is admissible as an array shape and array element, respectively. Array shapes are denoted by type-level lists formed from `Z` and `(:.)`—see [15, 34] for details. Array elements can be signed and unsigned integers (8, 16, 32, & 64-bits wide), floating point numbers (single & double precision), `Char`, `Bool`, shapes formed from `Z` and `(:.)`, as well as nested tuples of these. Note that `Array` itself is not in `Elt`.

The type signature for `fold` also shows the stratification into scalar computations using the `Exp` type constructor, and array computations that are wrapped in `Acc`. Collective operations consist of many scalar computations that are executed in data-parallel, but scalar computations *can not* contain collective operations. This stratification statically excludes *nested, irregular* data parallelism.

We represent the grammar of the Accelerate language in Figure 2. Overall, the collective operations in Accelerate are based on the scan-vector model [16, 52], and consist of multidimensional variants of familiar Haskell list operations, as well as array-specific operations such as index permutations. For example, `backpermute` constructs a new array using an index permutation function that specifies, for each index in the output array, which element of the

input array to read, while the function `generate` constructs a new array by applying a function at each index. See [15, 41] for more information.

At a second example, the following n-body code simulates Newtonian gravitational forces on a set of massive bodies in 3D space, using a precise (but expensive) $O(n^2)$ algorithm:

```
type Position   = (Double, Double, Double)
type Accel      = (Double, Double, Double)

calcAccels :: Acc (Vector Position) → Acc (Vector Accel)
calcAccels bodies =
  let n    = size bodies
      cols = replicate (lift $ Z :. n :. All) bodies
      rows = replicate (lift $ Z :. All :. n) bodies
  in
  fold plusV (constant (0,0,0)) (zipWith accel rows cols)
```

The core data-parallel structure of the implementation first computes the forces between every pair of bodies, before reducing the components applied to each body using a multidimensional reduction. Replicating the input vector in opposite dimensions and combining them with `zipWith` is a way to compute over all combinations of values drawn from the input. The function `replicate` is *shape polymorphic*, and extends an array across new dimensions. Here, `All` represents the original vector elements, which are repeated n times as rows or columns. For details on the various forms of shape polymorphism, see [34]. The acceleration between a pair of point masses is then calculated as:

```
accel :: Exp Position    -- The point being accelerated
      → Exp Position     -- Neighbouring point
      → Exp Accel
accel body1 body2
  = x1 ==* x2 &&* y1 ==* y2 &&* z1 ==* z2    -- if
  ? ( constant (0, 0, 0)                     -- then
    , acc )                                  -- else
  where
    acc          = lift (aabs * dx / r,
                         aabs * dy / r,
                         aabs * dz / r)
    (x1, y1, z1) = unlift body1
    (x2, y2, z2) = unlift body2
    dx           = x2 - x1
    dy           = y2 - y1
    dz           = z2 - z1
    rsqr         = (dx * dx) + (dy * dy) + (dz * dz)
    aabs         = 1 / rsqr
    r            = sqrt rsqr
```

The Accelerate code has some small syntactic overhead relative to a pure Haskell version. Since `Bool` can not be overloaded in standard Haskell, logical connectives are written using the operators `(==*)`, `(<*)` and so on, and `(?)` is used rather than `ifThenElse`. As there is no pattern matching, `lift` and `unlift` is used to pack and unpack expressions into and out of constructors such as tuples, respectively.

3. Fissioning and Scheduling Algorithms

Because the Accelerate language design does not commit to any architecture-specific communication patterns, scheduling policies, or granularity decisions, it should be possible to target more than one device using different compiler backends. Section 5.1 describes our common interface to backends and their runtimes.

Our goal in this paper is to compile an Accelerate program into *separately* executable components, to fully utilize the compute capability of a machine. Accelerate programs are, by their nature, already *DAGs* of array operators, exposing *task parallelism* between separate kernels. Thus, some programs will already be suited for using multiple devices, simply by partitioning the array operations already present. However, this can not generally be relied upon, and in fact many useful programs are — after fusion optimizations (§4.2) — a single kernel.

array exps	æ	::=	use $(arrconst)$ | map $(\lambda x.\, e)$ æ | generate σ $(\lambda x_0 ... x_n.\, e)$ | let $p = $ æ in æ | a | (æ, æ) | zipWith $(\lambda x\ y.\, e)$ e æ | fold $(\lambda x\ y.\, e)$ e æ | backpermute σ $(\lambda x.\, e)$ æ | permute $(\lambda x\ y.\, e)$ æ $(\lambda x_0 ... x_n.\, e)$ æ | reshape σ æ | slice σ æ | replicate σ æ
scalar exps	e	::=	let $p = e$ in e | x | c | $prim(e_0, ...)$ | $e_0?(e_1, e_2)$ | $(e_0, ..., e_n)$ | ...
patterns		::=	x | (p, p)
variables	x, a		
dim or hole	s	::=	e | $\square$
full shape	σ	::=	[] | $[s]$ | $[s_0 ... s_n]$
const	c	::=	$1, 2, ...$
arrconst		::=	$[c, c, \ldots, c]$

Figure 2. Grammar of the core Accelerate constructs. Elided here for simplicity are stencils, scans and other flavours of folds, plus segmented versions of these. In the case of replicate, *holes* in shapes are placeholders for existing dimensions that are not being extruded, whereas in slice they signify the dimensions that are *not* being removed / indexed into.

Why fission? Fissioning programs provides a way to *convert* latent parallelism, inside data-parallel operators, into explicit task parallelism. This in turn provides enough tasks to utilize the total horsepower of a machine: multiple GPUs, or CPU and GPU. For some algorithms, such as sorting, CPUs can be as fast as GPUs, or at least a significant fraction thereof. Furthermore, unbalanced machines with multiple CPUs and a single GPU, so there may be a large opportunity cost to ignoring the CPU.

Targeting multiple devices can also be a viable strategy for a programmer not to gain performance, but to gain insight into which device is better suited to their algorithm. In this case dynamic load balancing over the two devices can avoid making an incorrect choice. Our existing *Meta-Par* system for heterogeneous scheduling [20], takes the dynamic load balancing approach and is aware of scheduling Accelerate computations [2]. However, Meta-Par focuses on parallelism *between* Accelerate computations. In this paper, we are instead concerned with executing a *single* Accelerate computation over *multiple* devices. To that end, we develop a framework for correctness-preserving *fissioning* of Accelerate programs.

Arrays and shapes We will write arrays as $[1\ 2\ 3]$ or $\left[\begin{smallmatrix}1&2&3\\4&5&6\end{smallmatrix}\right]$. The *shape* of the former is $[3]$ and of the latter is $[2\ 3]$. This corresponds to the common $(rows, cols)$ ordering. Thus, given a row-major memory representation, the rightmost component of the shape corresponds to the *fastest varying* or *innermost* dimension of the array.

We use zero-based subscripting, a_i, for extracting elements from arrays; thus $[1\ 2\ 3]_0 = 1$ and $[1\ 2\ 3]_2 = 3$. Further, $\langle a \rangle$ denotes the `shape` of an array a. In our formal notation we shall treat shapes as arrays, so if $a = \left[\begin{smallmatrix}1&2&3\\4&5&6\end{smallmatrix}\right]$, then $\langle a \rangle_0 = [3\ 2]_0 = 3$. The *rank* of an array is the number of dimensions (*i.e.* $\langle\langle a \rangle\rangle_0$). Whereas the *size* of an array $|a|$ is the total number of elements in the array. Note that in the Accelerate source language, rank is static and encoded in the type system, and that most of Accelerate's core primitives are *rank-polymorphic*, which is in keeping with tradi-

tions established by many dynamically typed [multidimensional] array languages such as APL and Matlab.

Split and concat Because the focus of this paper is distributing work over multiple devices, splitting and concatenating arrays are central concepts.[7] The first rule of splitting and concatenating, of course, is that they are inverses:

$$\mathsf{split}_i\ a = (b, c) \iff \mathsf{concat}_i\ (b, c) = a$$

The subscript indicates which dimension is being split or concatenated, and can be treated as an index into the shape vector. We will commonly want to split on the innermost (last) dimension in the shape: split_{n-1} , where n is the size of the array in question. We abbreviate this as $\mathsf{split}_{\text{-}1}$, $\mathsf{split}_{\text{-}2}$, and so on. Since arrays may have odd length in the dimension being split, the split pieces will not necessarily be of the same size. In particular, they will be the same size as the original array in all dimensions but the one being split, and in that dimension they will have length `(n 'quot' 2)` or `(n 'quot' 2 +1)`, with the left half always being larger. That is:

$$\begin{array}{rcl} \mathsf{split}_i\ a = (b, c) & \Rightarrow & \langle b \rangle_j = \langle c \rangle_j = \langle a \rangle_j \quad \text{for } j \neq i, \\ & \wedge & (\langle c \rangle - \langle b \rangle)_i \in \{0, 1\} \end{array}$$

The above definition allows zero-sized arrays, *e.g.* $\mathsf{split}_0\ [v] = ([v], [])$ and $\mathsf{split}_0\ [] = ([], [])$. Note that in this notation we allow arithmetic operations such as `(-)` lifted over arrays, which is not standard in the Accelerate library itself but corresponds to `zipWith (-)`. For now we can assume that $\mathsf{concat}_i\ (b, c)$ requires that the pieces be *balanced*, such that the function is only defined when $(\langle c \rangle - \langle b \rangle)_i \in \{0, 1\}$.

3.1 Optimization Search: Term Rewriting System

Figure 3 defines a term-rewriting system that exposes a large *search space* of valid program transformations. An Accelerate optimizer can navigate this space in arbitrary ways, and be assured that the resulting program will run on any combination of Accelerate backends.[8] In the special case of a multi-device fission optimizer, the end goal is to end up with sufficient, balanced task parallelism for the hardware.

In the implementation, it is useful to be able to split at arbitrary points k *within* the i^{th} dimension of an array: *i.e.* $\mathsf{split}_i^k\ a$. However, we must take care when choosing split points, because split/concat elimination depends on having the same sized pieces. Thus, there is an incentive to fission adjacent operations *consistently*. In practice, this is not as big of a problem as it might seen. Adjacent map operations will usually be fused anyways (unless sharing of a result precludes it), and adjacent producers and consumers (map/fold) do not need to have the same work-subdivision. Indeed, the work devision in most programs is in fact consistent; i.e. for a given period of time, a CPU can consistently do $P\%$ of the work compared to the GPU.

3.2 Scheduling theory and algorithms

Task-scheduling of DAGs on parallel resources (minimizing schedule length) is long-studied in operations research and elsewhere, and well-understood to be an NP-hard. It can be solved optimally using branch-and-bound [31], or approximately with various heuristics [36]. Some formulations take into account hetero-

[7] Neither of these operations is primitive in the original Accelerate, but they are straightforward to add as library functions.

[8] As with the current Accelerate semantics, `fold` has a parallel semantics (without a deterministic topology) and requires an associative operation. This contract is not currently enforced by the type system, but could be by simply allowing only a fixed set of reduction functions, ala ArBB [43], perhaps with combinators to combine them.

$$\begin{array}{rcl}
\mathsf{fold}\ f\ e\ æ & \Rightarrow & \mathsf{let}\ (x, y) = \mathsf{split}_{\text{-}1}\ æ\ \mathsf{in} \\
& & \mathsf{zipWith}\ f\ (\mathsf{fold}\ f\ e\ x)\ (\mathsf{fold}\ f\ e\ y) \\
\mathsf{map}\ f\ æ & \Rightarrow & \mathsf{let}\ (x, y) = \mathsf{split}_i\ æ\ \mathsf{in} \\
& & \mathsf{concat}_i\ (\mathsf{map}\ f\ x), (\mathsf{map}\ f\ y) \\
\mathsf{generate}\ [e_0 ... e_n]\ f & \Rightarrow & \mathsf{concat}_i\ (\mathsf{generate}\ [e_0 ... \lceil e_i/2 \rceil ... e_n]\ f, \\
& & \quad \mathsf{generate}\ [e_0 ... \lfloor e_i/2 \rfloor ... e_n] \\
& & \quad (\lambda x_0 ... x_n.\ f x_0 ... \lceil x_i + e_i/2 \rceil ... x_n)) \\
\mathsf{reshape}\ [e_0 ... e_n]\ æ & \Rightarrow & \mathsf{let}\ (x, y) = \mathsf{split}_i\ æ\ \mathsf{in} \\
& & \mathsf{concat}_i\ (\mathsf{reshape}\ [e_0 ... \lceil e_i/2 \rceil ... e_n]\ x, \\
& & \quad \mathsf{reshape}\ [e_0 ... \lfloor e_i/2 \rfloor ... e_n]\ y) \\
\mathsf{slice}\ [e_0 ... e_n]\ æ & \Rightarrow & \mathsf{let}\ (x, y) = \mathsf{split}_i\ æ\ \mathsf{in} \\
& & \mathsf{concat}_i\ (\mathsf{slice}\ [e_0 ... e_n]\ x, \\
& & \quad \mathsf{slice}\ [e_0 ... e_n]\ y) \\
\mathsf{replicate}\ [e_0 ... e_n]\ æ & \Rightarrow & \mathsf{let}\ (x, y) = \mathsf{split}_i\ æ\ \mathsf{in} \\
& & \mathsf{concat}_i\ (\mathsf{replicate}\ [e_0 ... e_n]\ x, \\
& & \quad \mathsf{replicate}\ [e_0 ... e_n]\ y) \\
\mathsf{permute}\ f\ d\ g\ a & \Rightarrow & \mathsf{let}\ (x, y) = \mathsf{split}_i\ æ\ \mathsf{in} \\
& & \mathsf{zipWith}\ f\ (\mathsf{permute}\ f\ d\ g\ x) \\
& & \quad (\mathsf{permute}\ f\ d \\
& & \quad\ (\lambda x_0 ... x_n.\ g\ x_0 ... \lceil x_i + e_i/2 \rceil ... x_n)\ y) \\
\mathsf{zipWith}\ f\ æ_1\ æ_2 & \Rightarrow & \mathsf{let}\ (x_1, y_1) = \mathsf{split}_i\ æ_1\ \mathsf{in} \\
& & \mathsf{let}\ (x_2, y_2) = \mathsf{split}_i\ æ_2\ \mathsf{in} \\
& & \mathsf{concat}_i\ (\mathsf{zipWith}\ f\ x_1\ x_2, \mathsf{zipWith}\ f\ y_1\ y_2) \\
\mathsf{backpermute} & \Rightarrow & \mathsf{concat}_i \\
\quad [e_0 ... e_n] f æ & & \quad (\mathsf{backpermute}\ [e_0 ... \lceil e_i/2 \rceil ... e_n]\ f\ æ, \\
& & \quad \mathsf{backpermute}\ [e_0 ... \lfloor e_i/2 \rfloor ... e_n] \\
& & \quad (\lambda x_0 ... x_n.\ f x_0 ... \lceil x_i + e_i/2 \rceil ... x_n) æ) \\
\mathsf{use}\ [c_0 \ldots c_n] & \Rightarrow & \mathsf{concat}_{-1}\ (\mathsf{use}\ [c_0 \ldots c_j]) \\
& & \quad (\mathsf{use}\ [c_{j+1} \ldots c_n])
\end{array}$$

Figure 3. Fission rewrite rules. One rule application fissions one data-parallel combinator. We use function application ($f x$) here as a shorthand; all scalar functions are inlined/combined. Also, we use Accelerate surface syntax for clarity, but within our compiler we implement these rules against a *flat* representation of program as a table of array bindings—no array let expressions.

$$\begin{array}{lcl}
\mathsf{concat}_i\ (\mathsf{split}_i\ æ) & \Rightarrow & æ \\
\mathsf{split}_i\ (\mathsf{concat}_i\ (a, b)) & \Rightarrow & (a, b) \\
\mathsf{map}\ f\ (\mathsf{map}\ g\ æ) & \Rightarrow & \mathsf{map}\ (f \circ g)\ æ \\
\mathsf{map}\ f\ (\mathsf{zipWith}\ g\ æ_1\ æ_2) & \Rightarrow & \mathsf{zipWith}\ (\lambda x y.\ f(g\ x\ y))\ æ_1\ æ_2 \\
\mathsf{zipWith}\ g\ (\mathsf{map}\ f\ æ_1)\ æ_2 & \Rightarrow & \mathsf{zipWith}\ (\lambda x y.\ g\ (f\ x)\ y)\ æ_1\ æ_2 \\
\mathsf{zipWith}\ g\ æ_1\ (\mathsf{map}\ f\ æ_2) & \Rightarrow & \mathsf{zipWith}\ (\lambda x y.\ g\ x\ (f\ y))\ æ_1\ æ_2 \\
\mathsf{zipWith}\ g\ æ\ æ & \Rightarrow & \mathsf{map}\ (\lambda x.\ g\ x\ x)\ æ \\
\mathsf{map}\ f\ (\mathsf{generate}\ [e_0 ... e_n]\ g) & \Rightarrow & \mathsf{generate}\ [e_0 ... e_n]\ (f \circ g) \\
\mathsf{map}\ f\ (\mathsf{replicate}\ [e_0 ... e_n]\ æ) & \Rightarrow & \mathsf{replicate}\ [e_0 ... e_n]\ (\mathsf{map}\ f\ æ)
\end{array}$$

Figure 4. Fusion rules corresponding to those performed by the `fuseMaps` compiler pass (Section 4.2). Note that fusing `zipWith` over *identical* arguments is important because the first two zipWith optimizations often lead to this situations. Pushing map through replicate is important both because it saves useless computations and because it enables downstream optimizations: the map can continue fusing to the right, and the replicate is likely to inline into a downstream consumer during the `inlineCheap` pass. Also, above we use function composition as a shorthand; scalar function application is not in the language—all scalar functions are inlined/-combined.

geneous processors [46] and communication delays. In one sense, this technology directly applies because an Accelerate intermediate representation *can* be viewed as a shorthand for a (very large) task graph. That is, each data-parallel operations such as `fold`, could be replaced with a large fork-join task graph. Several issues are immediately apparent: since array size is a runtime property, the resulting graphs would only be knowable dynamically, and it would be completely infeasible to expand these full graphs in order to use explicit scheduling technology.

Instead, it is better to view the Accelerate program as-is: as a coarse-grained task-DAG with *latent parallelism* that can be exploited via fissioning. At least some of the work within the area of *divisible load theory* [7] applies here; Using the terminology of that area, Accelerate workloads are "modularly divisible" loads (DAGs) containing "arbitrarily divisible" loads (fissionable operators), for which some heuristics were proposed [6].

In future, we can apply scheduling policies of this sort as a collaboration between the fissioning pass and the final runtime scheduling step. In our present prototype, we explored two different scheduling policies that fission *all* array operators into a consistent number or size of partitions. While one could *construct* examples with sufficient (balanced) task parallelism such that this much fissioning is not necessary, in practice this simple policy is effective for the lion share of applications.

3.2.1 Algorithm 1: Naive "BSP" style

The first policy we tried was a *bulk-synchronous parallel* approach: each kernel is fissioned P-ways, and the runtime scheduler runs the DAG with a limited look-ahead in the graph: round-robin mapping of operators whose dependencies are met to available devices. The problem with this approach is that not only are synchronization costs (global barriers) paid for at every batch of operators launched, but because there's no notion of device affinity, all output arrays are copied back to the host from all accelerator devices at the end of each round. For example, this resulted in the n-body program running ten times *slower* than on one CPU device, and thus the approach was abandoned.

3.2.2 Algorithm 2: "SPMD" style

Our next approach was to implement "Single Program Multiple Data", where a slice of the entire program on a subset of the data is executed on each device. However, in general such "vertical" slices would need to communicate, and can not run completely independently. Thus the approach we use is to use SPMD "locally" to schedule regions of the program that do not communicate, and then to recursively perform inter-device transfers and repeat the scheduling loop. The algorithm is as follows. The scheduler analyzes the graph produced by earlier compiler passes, and if a final output array from the program is the result of `concat`, then for each fragment:

1. Assign it to a currently-unassigned device
2. Trace its dependencies, greedily adding upstream array operations to the same device.
3. Stop when encountering an operator assigned to another device.

The scheduling loop continues until the entire Accelerate program is executed. If fissioning were not run, then the program would execute by means of the scheduling loop grabbing DAG nodes in a demand driven fashion and arbitrarily scheduling them on free devices. Thus, coordination is required between the fissioning process and the scheduler: as the fissioning operator makes specific decisions about work ratios, it thus assumes that work fragment n (of P) will be assigned to the corresponding device n.

Note that this approach to SPMD execution is substantially different than more traditional approaches [48], wherein it is common for SPMD scheduling to be an "extra-syntactic" property that is not explicit in the program representation, but is implicit in how the program is used. The advantage of our approach is that scheduling decisions — fissioning — become source-to-source transformations. The liability here is that it increases program sizes, since fissioning is represented directly in the DAG (de-duplicating lambdas would alleviate this somewhat). We evaluate the effect of this program bloat on compile time in Section 6.

4. Implementation: Micropass approach

We now describe our new compiler infrastructure that provides the context for the fissioning transformations described in Section 3. This infrastructure is provided in a package called *accelerate-backend-kit*, and, as the name implies, a major goal of this package is to lower the barrier to creation of new Accelerate backends, maximizing reuse of the compiler passes shown in in Figure 5. This goal is accomplished making the final IR (KernelIR) as utterly simple and portable as possible, minimizing the amount of work required for a new code generation backend, and also consolidating runtime services (caching and the generic backend interface, Section 5.1).

The compiler is constructed using a *micropass*[9] methodology, where the goal is to separate all logically unrelated compilation tasks into separate passes — for example, an analysis pass followed by an optimization pass that consumes the metadata produced by the analysis. Our Accelerate compiler uses 26 passes divided into three phases, corresponding to three different intermediate representations (IRs) described below.

4.1 Phase 1: From Acc to SimpleAcc

The first phase of the compiler pipeline converts from the Accelerate front-end representation to a simplified IR, named `SimpleAcc`.[10] The main difference between the two is that the original Accelerate language includes:

- Polymorphic head and tail operations on tuples.
- Tuples of arrays.
- Arbitrary nesting of array expressions, including inside scalar expressions.

The first phase of the compiler uses standard transformations—found in passes `staticTuples`, `liftComplexRands`, `gatherLets`, and `removeArrayTuple`—to yield a *flat* representation of the program as a table (graph) of array and scalar operations.

The reason array tuples can be removed entirely, is that, like other JIT compilers, the Accelerate compiler is effectively a *whole program* compiler that also lacks obfuscating non-local control constructs at the array level. Conditionals are present, but because the language lacks side effects, they are straightforward to push inside tuples. For example:

```
if e then (arr1,arr2) else (arr3,arr4)
```

[9] The *micropass* technique employed here refers to a specific style of compiler construction developed at Indiana University by Kent Dybvig and Daniel P. Friedman. It was developed originally in a compiler class [51], that has since been adopted by other Universities (*e.g.* Northwestern and the University of British Columbia). Since then it has been employed by a number compilers including Chez Scheme [33], Ikarus/Vicare Scheme, and most recently the Harlan DSL [30].

[10] The Accelerate `backend-kit` infrastructure also had its origin in a class on Array DSL compiler construction, and for this reason an additional goal of the design was to make it easier to use, specifically by using basic Haskell98 data structures for AST representation, rather than the GADT-heavy representations in the Accelerate front-end.

Becomes:

```
let tmp = e in
(if tmp then arr1 else arr3,
 if tmp then arr2 else arr4)
```

And by the end of this first phase of the compiler, we have a flat structure of top-level `ProgBind`s:

```
data ProgBind decor =
     ProgBind Var Type decor (Either Exp AExp)
```

Where each `AExp` contains only a single array operation, referencing any other values by name. That is, the program is in effectively in *A-normal form* at the array level (though not at the `Exp` level).

4.2 Phase 2: Optimization and lowering

The second phase contains the bulk of our compiler, including analysis and optimization passes, and lowering passes that significantly decrease the "width" of the language (eliminating many array operations), making the job simpler for downstream code generators—so that we might have more of them!

- Desugar to backpermute – later eliminate multidimensional slice and replicate operations, rewriting as backpermute.
- Desugar to generate – eliminate the operators `map`, `zipWith`, `backpermute`, replacing them with `generate`.

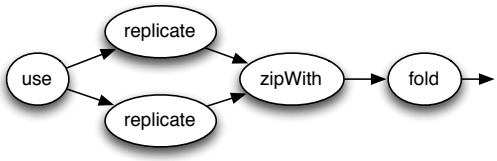
The phase ordering is delicate, and it is critical that *fusion*—the single most important optimization for an array DSL [41]—happens before the desugaring steps above. Essentially, fusion (and fission) are what take advantage of the high-level semantics, and restricted communication patterns of specific operators like zipWith, and after that point, any remaining instances of these operations can be desugared without losing important information. Actually, fusion happens in multiple phases within the compiler, and the `fuseMaps` pass performs all the fusion rewrites that are valid in the current IR (Figure 4).

There is, however, a **fold/map fusion conundrum**. It is very important to combine *producers* of large data structures (*e.g.* `generate`) with *consumers* (*e.g.* `fold`). This is especially so in Accelerate, as in the N-body example program from Section 2.3, where large arrays are often simply "iteration spaces" (loop indices), for performing a computation. Unfortunately, the result of fusing a `generate` and a `fold` is not expressible in the language of Figure 2.

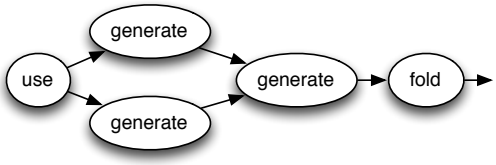
The `backend-kit` does this somewhat differently. As we will see below, once we reach the `CLikeIR` representation, it is possible to directly represent fused producers and consumers. Indeed, it is a hallmark of micropass compilers, that they must always be able to represent their decisions in the IR explicitly.

Besides fusion, the other optimizations in this phase are dead code elimination and inlining. Inlining in this case refers to inlining *arrays*, that are created by computing a function of other arrays (*e.g.* using `generate`). This pass depends on a static work estimation (`estimateCost`). Together, these passes can in some cases enable fusion and deforestation in spite of *sharing*, whereas previous optimization approaches for Accelerate would not duplicate *any* work (*i.e.* never fuse array operations whose results are used more than once) [41]. Finally, phase 2 performs dead code elimination, for example, removing arrays that have been inlined at all of their use sites.

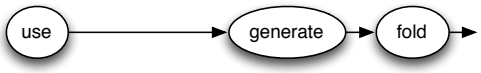
Fusion & inlining example One clear example of the need for fusion can be found in the n-body example from Section 2.3. In that code, replicate was used to generate two n^2 size matrices for processing n bodies. It is essential that these matrix be deforested! The array dataflow graph for this application starts out as:



The first fusion pass (`fuseMaps`) does nothing on this example, but replicate is desugared into backpermute, and finally generate:



Finally, `inlineCheap` has its turn, and inlines the two replicates into what was formerly a zipWith. Finally, at this point we have gone as far as we can in the `SimpleAcc` representation:



Other lowerings In addition to optimizations, phase 2 is responsible for lowering the language by removing features: multidimensional arrays, implicit shapes, arrays of tuples, scalar-level tuples.

- `sizeAnalysis` and `trackUses` – Analysis passes that support the lowering and optimization passes. The former is a simple static analysis to track the size of intermediate arrays based on the size of input arrays (often, but not always, statically knowable), the latter simply builds data structures to track which top level array results depend on which others.
- `explicitShapes` – Rather than a *shape* being an implicit part of each array (length of each dimension), this pass introduces explicit scalar bindings to track shapes, *e.g.* `arr_shape` holding the shape of `arr`. When shapes are known statically, these variables are bound to constants that can be inlined. This pass eliminates all scalar-level primitives for retrieving shapes, computing the size of arrays of a certain shape, and so on.
- `oneDimensionalize` – This pass makes all arithmetic related to multidimensional indexing explicit in the program,[11] committing to a row-major representation and introducing simple *strided* array operations in lieu of multi-dimensionality.
- `normalizeExps` – This pass performs a standard flattening of nested expressions, introducing temporary variables and leaving the language in a state more readily converted to, *e.g.*, C-like languages that distinguish between statements and expressions.
- `unzipArrays` – Replace arrays of tuples by multiple array-of-scalar values.
- `unzipETups` and `convertToCLike` – `Exp`-level scalar tuples are eliminated over a two pass sequence, which requires finally changing over to the `CLikeIR` to encode lambdas with untupled arguments and multiple results.

The end result of this pipeline is to move from a complex data model (tuple of multidimensional array of tuple) to a very simple data model (arrays of scalars only, strided operations).

[11] In the OpenCL backend there is an opportunity cost to eliminating multidimensional arrays, in that it makes it harder to directly utilize 2D and 3D kernels supported by the hardware. As we go further with our OpenCL alternative to our Cilk backend, we will make this pass optional.

4.3 Phase 2 Intermission: perform fissioning

In backends that use it, fissioning happens as a compiler pass in between `fuseMaps` and `desugToBackperm` (Figure 5). The pass applies the rewrite rules in Figure 3, with the policy of fissioning *all* array operators P-ways, with fixed ratios for work sizes.[12]

The code generation strategy closely follows the rewrite rules. For example, if we start with the program:

```
-- Suppose a is a vector of length 10:
let b = map f a
```

The fission pass then introduces bindings for variables `b_i` representing the fragments of the pre-fissioning array. Our current implementation *always* splits on the *outermost* dimension (*i.e.* splitting the physical representation in half in a row-major representation):

```
let (a_1,a_2) = split a
    b_1 = map f a_1
    b_2 = map f a_2
    b = concat b_1 b_2
```

And if `b` is not removed by dead-code elimination, then it currently leaves behind this code:

```
b = generate (Z :. 10)
     (λ i → (i <* 5) ? (b_1 ! i, b_2 ! i-5))
```

If executed to generate a manifest array in memory, this conditional is not the most efficient method, and we may improve it by adding primitives to the language for block-copy memory operations. However, the above code has the advantage that `inlineCheap` will score it as inexpensive, and any kernels explicitly dereferencing `b!i` will inline the body of the generate (the conditional) in the place of the array reference. In fact, if the index is statically known, *e.g.* `b!3`, then the conditional can be eliminated at compile time, leaving `b_1!3` only.

4.4 Phase 3: from CLikeIR to KernelIR

At the end of phase 2, we convert to `CLikeIR`, which is still a purely functional language, using a single assignment (SSA) form at the expression level. The combinator set of Figure 2 is at this point reduced to a few core forms including generating manifest kernels in memory and generating kernels that have reductions associated with them (which may be various scan or fold variants). Looking in the `accelerate-backend-kit` source reveals the following definition:

```
data TopLvlForm =
    ScalarCode ScalarBlock -- Bind one or more vars.
  | Cond Exp Var Var       -- Conditionals
  | Use  AccArray          -- Array literals/inputs
  | GenManifest (Generator (Fun ScalarBlock))
  | GenReduce {
      reducer    :: Fun ScalarBlock,
      generator  :: Generator (Fun ScalarBlock),
      variant    :: ReduceVariant Fun ScalarBlock,
      stride     :: Stride Exp }
```

At this phase, it *is* possible to fuse a `GenManifest` into a `GenReduce`. This is the point where the n-body program, for example, reduces to a single `use` and a single kernel.

KernelIR The final intermediate representation makes arrays mutable, and array allocation explicit, replacing a `GenManifest` into a separate `NewArray` and `Kernel` operation, with the latter being nothing but a low-level parallel loop construct, similar to the [multi] loop representation at the heart of SAC [27] and Delite [50]. In the process of desugaring `Generate`, the compiler also performs a form of closure conversion, converting the λ that parameterizes each `Kernel` to take all [formerly] free-variables as explicit arguments. Finally, it is worth noting that attempting fissioning in this low-level IR would have all the same problems as fissioning OpenCL kernels themselves! Thus, fissioning happens much earlier, even if we must be careful to leave the post-fissioning, and post-SPMD-partitioning programs in a state that is still maximally optimizable through kernel fusion.

[12] In the future, we will build an adaptive profiling system that automatically feeds profiling data back into this phase of the compiler. In the current prototype, this is manual.

```
-- Phase3: The final step: Lower to a kernels only.
-- Perform optimizations that have been waiting on the
-- alternate representation.
phase3 :: CLikeIR () → KernIR (FreeVars)
phase3 prog =
  desugarGenerate   $     -- (freevars)
  fuseGenReduce  id $     -- (freevars)
  convertToGPUIR    $     -- (freevars)
  kernFreeVars      $     -- (freevars)
  prog

-- Phase2: The bulk of the compilation process:
-- eliminate unnecessary forms and lower the language.
phase2 :: SimpleAcc () → CLikeIR ()
phase2 prog =
  convertToCLike      $    -- ()
  unzipArrays         $    -- (inptbl,(subbinds,(foldstride,sz)))
  unzipETups          $    -- (subbinds,(foldstride,size))
  normalizeExps       $    -- (foldstride,size)
  oneDimensionalize   $    -- (foldstride,size)
  deadCode (fmap fst) $    -- (size)
  trackUses           $    -- (size,uses)
  inlineCheap (fmap fst) $ -- (size)
  estimateCost        $    -- (size,cost)
  desugToGenerate     $    -- (size)
  desugToBackperm     $    -- (size,uses)
   -- (*) Fissioning, when activated, happens here!
  fuseMaps  id        $    -- (size,uses)
  trackUses           $    -- (size,uses)
  explicitShapes      $    -- (size)
  sizeAnalysis        $    -- (size)
  desugarUnit         $    -- ()
  prog

-- Phase1: Convert the sophisticate Accelerate-internal AST
-- representation into something very simple.
phase1 :: (Arrays a) ⇒ Acc a → SimpleAcc ()
phase1 prog =
  typecheck         $ -- Initial Typecheck
  removeArrayTuple  $ -- Changes to SimpleAcc
  gatherLets        $
  liftComplexRands  $
  staticTuples      $
  accToAccClone     $
  prog
```

Figure 5. Full code for the compiler pipeline. The comments to the right track what metadata is stored on each top-level `ProgBind` after running the pass on that line.

5. CPU Backend with CilkPlus

The compiler pipeline described in Section 4 is *platform independent*. In this section, we descibe the specific code generator we have made to target x86 processors[13] using CilkPlus and the Intel C compiler. This in itself is not new: many array languages target CPUs already (*e.g.* SAC[27], ArBB[43]), though most systems target one or the other and few have mature backends for both CPU and GPU (*e.g.* Delite[50]).

Cilk Plus Cilk itself dates from 1996 [11], and is well-known for popularizing the *work-stealing* approach to dynamic scheduling.

[13] This includes both traditional CPUs and Xeon Phi accelerators, which have similarities to GPUs, but support both x86 and the Cilk runtime.

Cilk is a simple language extension that adds parallel subroutine calls to C/C++. Only two core constructs make up the core of Cilk: `cilk_spawn` for launching parallel subroutine calls, and `cilk_sync` for waiting on [all] outstanding calls. Modern Cilk implementations provide a `cilk_for` drop-in replacement for `for` (implemented in terms of `cilk_spawn` and `cilk_sync`).

When the company CilkArts was acquired by Intel and reimplemented inside the Intel C compiler (ICC), it was combined with existing ICC vectorization capabilities to form *Cilk Plus* (*i.e.* Cilk Plus Array Notation). This is the key reason we use Cilk as our x86 code generation choice. We are able to generate a parallel for loop for each `Kernel`, simply with:

```
#pragma vector always
 #pragma ivdep
 _Cilk_for (int i3 = 0; (i3 < inSize); i3 += inStride)
 {
```

ICC is able to handle both SSE/AVX SIMD vectorization, and multithreading with load balancing (traditional Cilk). Indeed, ICC also provides us with one other feature we use for parallelization: *Cilk hyperobjects* [21], which we use when generating code for fold operations, by incorporating reduction variables with parallel loops such as the one above.

Implementation status We began implementing our prototype compiler in Fall 2012. The compiler is complete in the depth dimension (all necessary passes), but is not complete in its breadth: does not implement scan, stencil, or Accelerate's recently added iteration constructs. Once the compiler is at feature parity with the main Accelerate release, we will publish our packages to Hackage (currently available on github.com).

5.1 Accelerate Runtime Interface: `Backend` Class

When evaluating fissioned Accelerate computations over multiple CPUs and GPUs, we require that (a) each backend is able to efficiently execute its portion of the computation; and (b) backends can communicate efficiently with each other. The `Backend` class encapsulates a low-level interface over code generator and expression evaluation in an Accelerate backend:

```
class Backend b where
  type Remote b r
  type Executable b r

  copyToHost   :: b → Remote b a → IO a
  copyToRemote :: b → a → IO (Remote b a)
  copyToPeer   :: b → Remote b a → IO (Remote b a)

  compileFun1 :: b → AST.Afun (x → y)
                   → IO (Executable b (x → y))
  runFun1 :: b → Executable b (x → y)
               → Remote b x
               → IO (Remote b y)
```

The associated type `Remote` specifies how a backend stores array data. For example, GPUs typically have their own high-performance memory, which is separate from the host's main memory. Data transfer is by DMA (direct memory access) and needs to be explicitly managed by the runtime. Thus, the functions `copyToHost` and `copyToRemote` transfer data between the host and device memory spaces. The function `copyToPeer` allows for efficient copying of an array to a backend instance of the same type, so that two GPUs can share data directly via DMA without intervention by the CPU or via an intermediate copy to host memory over the relatively narrow PCIe bus.

The `Backend` class makes an explicit separation between the process of compiling and executing an Accelerate expression. This is done so that the associated type `Executable` can be used to represent something directly executable by the backend instance, for example compiled object code. Thus, compilation overhead can be amortized if the function is evaluated multiple times. The type `AST.Afun` is an array valued function that has been lowered by the Accelerate front-end from the surface language representation in HOAS (higher-order abstract syntax) of type `Acc`, into the internal de Bruijn representation.

6. Evaluation

In this section we use our prototype implementation to test the hypotheses we have put forward in this paper:

- That higher arithmetic intensity kernels are much faster on the GPU, but that lower intensity ones are better on the CPU due to saved data movement (a principle known from prior work, confirmed as a sanity check here).
- That the fissioning rewrite rules of Figure 3, do not introduce a substantial loss in performance, even if the program is never actually executed on multiple devices in the compiler backend; and, further, that the compile time overhead of the larger ASTs is manageable.[14]
- That fissioned programs run on multiple (single-threaded) backends, do as well as multithreaded backends such as Cilk, due to the regular nature of these workloads. (Still, Cilk is a wiser choice in general if there is any chance of other load on the machine.)
- And, finally, that purely functional array programs run faster on multiple devices than on one, even over heterogeneous devices.

We conducted our experiments on Delta, a GPU cluster within FutureGrid. Each node in Delta contains two NVIDIA Tesla C2075 GPUs as well as two six-core Intel Xeon X5660 CPUs with hyper-threading for a total of 24 logical threads. While interpreting the benchmarks in this section, refer to the following glossary of backend configurations used:

- Sequential C – a version of the Cilk backend without any parallel annotations.
- Cilk – As described in Section 5.
- CUDA – Origianl Accelerate backend
- Fission/Seq – Here we perform fissioning inside the AST but execute the entire thing on one sequential C backend.
- SPMD/Seq – Here we pretend that each core of the CPUs is its own "device" and launch a separate (sequential) process for each.
- CPU/GPU – Uses SPMD, but treats the CPU as one device and the GPU as one device.

We evaluate these in the context of only a couple benchmarks (n-body and blackscholes) and synthetically generated microbenchmarks. However, even these two benchmarks are representative of a large class of similar problems: (1) kernels with linear work in the data copied, (2) kernels that perform superlinear work because they fold over an iteration space.

Microbenchmarks The first experiment, shown in Figures 6 and 7, is simply to scale the arithmetic intensity of an kernel and observe its effect on the relative performance of the CPU and GPU. Both tests perform a map over a 2M element array, while

[14] In future work it would make sense to introduce a more efficient representation for multiple kernels that share the same, or nearly the same, *bodies*, but unfortunately this adds substantial complexity to all the passes that use this representation.

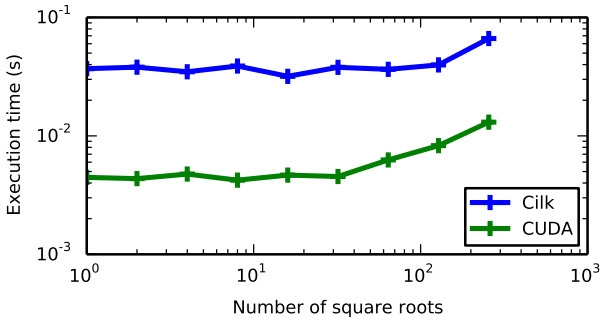


Figure 6. GPU vs. CPU backend performance as a function of arithmetic intensity. In this first version of the benchmark, we build a large piece of code that applies many square-root operations (without a loop).

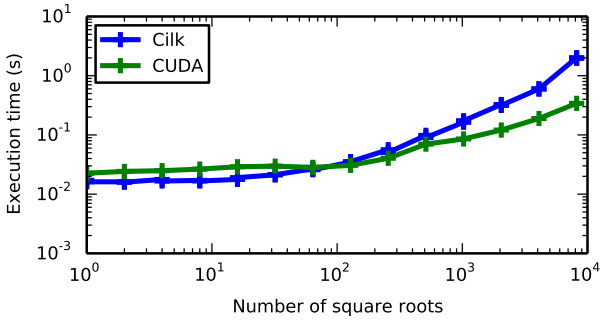


Figure 7. An alternate version of the benchmark in Figure 6, which performs the same FLOPS/byte transferred, but which uses a loop for executing the `sqrt` functions.

performing n sqrt operations for each element. In the first test, one large arithmetic expression is generated, in the second a loop is generated. In spite of doing the same amount of "work", we see the GPU does much better vis-a-vis the CPU when scalar (per-thread) loops are not involved.

Fissioning cost One might naturally be concerned that fissioning more than is absolutely necessary *pollutes* the program, bloating its size, adding to binary size at runtime. Indeed, this is one reason why in our initial prototype, we choose not to ever fission more ways than the number of devices. However, Figure 8 shows a surprising result in this respect. For n-body, performance steadily improves when introducing more fissioning! After verifying that this was not due to accidental parallelism, or to disparities in vectorization, it became clear that fissioning had, incidentally, *served as a loop-tiling optimization*, improving sequential execution speed by increasing cache-friendliness.

Static vs. dynamic work division As another experiment in fissioning, we used a separately compiled sequential binaries within the CPU, on each core (SPMD/Seq). Figure 9, shows the result of splitting the n-body application 1-12 ways, running each partition in a *sequential* C backend (*i.e.* not Cilk).

6.1 Heterogeneous CPU-GPU Parallelism

The final thing we test is whether the *two* CPUs on this machine have sufficient throughput to meaningfully help a single GPU

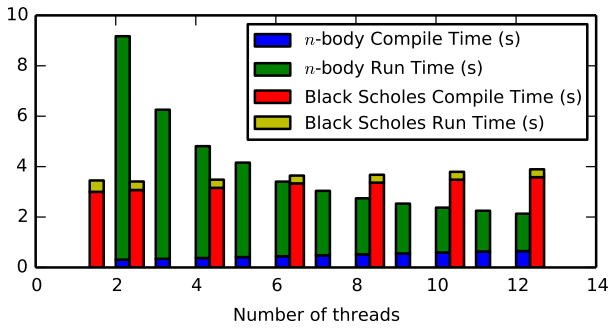


Figure 8. Effect of Fissioning pass on compile and run times. The x-axis represents how many partitions each parallel array operator was split into. Note that P-way fissioning increases AST size by a $P + k$ factor, with an additional constant, k added to the coefficient because of the need to generate *reconcattenated* arrays (even though they are usually later eliminated as dead code). The n-body benchmark ran with 25,000 bodies and the Black Scholes benchmark simulated 2,000,000 options.

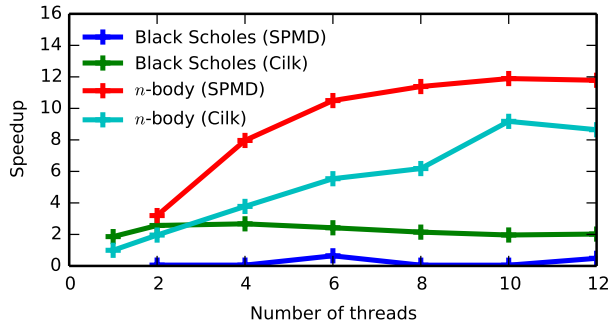


Figure 9. Parallel speedup for SPMD vs. Cilk from 1 to 12 threads on a two-CPU system. Cilk has the advantage of dynamic load balancing and is multithreaded rather than multi-process. Still, for these balanced workloads SPMD can sometimes do well. Here, n-nody used 15,000 bodies and Black Scholes used 100,000 options. (Note, the fissioning phase currently has trouble with the large, $100k$ element array—it uses an inefficient method to fission the Haskell `Data.Array` itself, converting through a list!—and Blackscholes/SPMD does very poorly.)

along. The short answer is that for the n-body workload, each CPU is roughly 3.5× times slower (only 1.7× slower together). Nevertheless, that means a 48% reduction in execution time (not counting data transfers), when both CPUs are incorporated (Figure 10). As mentioned in Section 4.3, in our prototype we currently have to set split ratios manually if we want anything other than the default of splitting into P even pieces. In the n-body case we split at a 1:2 ratio (CPU/GPU).

7. Related Work

There have been several efforts to cooperatively use both the CPU and GPU in a single application. Many projects are targeted at large scale scientific computations, in order to take advantage of the recent trend towards GPU computing in supercomputers. However, much of this work is focused on particular applications, such as

	Time (s)	Speedup
Sequential	15.7	1.00
Cilk	1.41	11.13
GPU Only	0.8	19.6
CPU + GPU	0.54	29.1

Figure 10. Performance of heterogeneous CPU plus 1-GPU solution vs. either device type alone for the n-body benchmark with 25,000 bodies. In the heterogeneous, case the CPUs were given one third of the work.

MapReduce [17] or graph traversals [24, 42]. Our work is different, in that it focuses on a general purpose language.

QiLin [37] is able to automatically schedule tasks on both the CPU and the GPU. The QiLin system requires a training phase to determine the optimal balance between the CPU and GPU, which can vary significantly depending on the application. QiLin includes a library of linear algebra primitives, but otherwise programmers must provide both a CPU and GPU implementation in order to take advantage of QiLin's load balancing, whereas we require only a single source program.

Delite/LMS [50] is a library-based parallelisation framework for DSLs in Scala that allows specifying complex optimisations in a modular manner. The Delite code generator is able to target multicore CPUs and GPUs, demonstrating impressive performance on both.

Dominik et. al. [28] use a machine learning based approach that analyses program characteristics to decide how to statically partition an OpenCL kernel across the CPU and GPU. Their later work extends the model to take GPU contention into account [29]. This method requires the program to fall into certain classes understood by the machine learning model. In contrast, Wang et. al. [57] use a dynamic scheduling approach that uses runtime profiling to predict the best way to split work between the CPU and GPU.

8. Discussion and Future Work

Much work remains to fully explore the potential of single kernel, multiple device EDSLs, particularly in the area of scheduling algorithms. In this paper we have presented a proof of concept: a prototype that makes Accelerate the first purely functional SKMD EDSL. Our prototype demonstrates the possibility of transparently using whatever hardware is available on the users machine, without changing the high-level source code.

Nevertheless, for the field of high-level array DSLs, generally, deployability and user friendliness remain problems. Many EDSL packages are difficult to install or have inflexible requirements. For example, in the Haskell context, utilizing Accelerate on GPUs from within a published cabal package requires an explicit dependency on the `accelerate-cuda` package, which restricts the potential audience and erects barriers (installing CUDA) to anyone wishing to use the package. It is no surprise then that in spite of DSL prototypes being available for years, virtually no "normal" libraries and programs depend on them. What then would it take for, say, the `sort` functions from the library `vector-algorithms`[15] to transparently use vectorized code generation and accelerators on the machines which it is installed? We believe it will require two essential ingredients:

1. A failure-resilient method to probe a host machine, determine its vector and accelerator capabilities (e.g. AVX[16] plus a GPU), and then automatically install a software stack enabling programs to utilize those resources.
2. A parallel code-generation and scheduling framework that is capable of efficiently utilizing whatever devices are available, such as GPUs and CPUs, alone or in combination, whether or not there is already load on the machine.

This paper has taken a step towards the latter, and in future work we hope to get closer to this vision as a whole, and see vectorized code become a *normal* part of the library ecosystem.

[15] `http://hackage.haskell.org/package/vector-algorithms`

[16] AVX/SSE refer to extensions to x86 supported by regular Intel and AMD processors. This is the form of "vectorized" execution available on CPUs.

Acknowledgments

Edward Amsden developed an early version of the SPMD scheduler. This work was supported by NSF grant XPS-1337242.

References

[1] gpuocelot: A dynamic compilation framework for ptx. `https://code.google.com/p/gpuocelot/`.

[2] How to write hybrid CPU/GPU programs with Haskell. `http://parfunk.blogspot.com/2012/05/how-to-write-hybrid-cpugpu-programs.html`.

[3] PGI CUDA-x86. `http://www.pgroup.com/resources/cuda-x86.htm`.

[4] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, 2009.

[5] E. Axelsson. A generic abstract syntax model for embedded languages. In *ICFP: International Conference on Functional Programming*. ACM, 2012.

[6] S. Bataineh and B. Al-Asir. Efficient scheduling algorithm for divisible and indivisible tasks in loosely coupled multiprocessor systems. *Software Engineering Journal*, 9(1):13–18, 1994.

[7] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[8] G. E. Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[9] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-95-170, 1995.

[10] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP: International Conference on Functional programming*. ACM.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30:207–216, August 1995.

[12] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation. In *OOPSLA: International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2011.

[13] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5), 2009.

[14] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Principles and Practice of Parallel Programming*, pages 47–56, 2011.

[15] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Declarative Aspects of Multicore Programming*. ACM, 2011.

[16] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing*. IEEE Computer Society Press, 1990.

[17] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a coupled CPU-GPU architecture. In *High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[18] R. Clifton-Everest, T. L. McDonell, M. M. Chakravarty, and G. Keller. Embedding foreign code. In *Practical Aspects of Declarative Languages*. Springer International Publishing, 2014.

[19] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*. ACM, 2007.

[20] A. Foltzer, A. Kulkarni, R. Swords, S. Sasidharan, E. Jiang, and R. R. Newton. A meta-scheduler for the par-monad: Composable scheduling for the heterogeneous cloud. In *ICFP: International Conference on Functional Programming*. ACM, 2012.

[21] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Parallelism in Algorithms and Architectures*. ACM, 2009.

[22] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[23] A. Gill. Type-safe observable sharing in Haskell. In *Haskell Symposium*. ACM, 2009.

[24] M. Goldfarb, Y. Jo, and M. Kulkarni. General transformations for GPU execution of tree traversals. In *High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[25] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems*. ACM, 2006.

[26] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Performance Analysis of Systems and Software*. IEEE Computer Society, 2011.

[27] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multi-threaded execution. *Journal of Parallel Programming*, 34(4):383–427, Aug. 2006.

[28] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *International Conference on Compiler Construction*. Springer-Verlag, 2011.

[29] D. Grewe, Z. Wang, and M. O'Boyle. OpenCL task partitioning in the presence of GPU contention. In *Languages and Compilers for Parallel Computing*, 2013.

[30] E. Holk, W. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for GPUs. In *International Conference on Parallel Computing*, Sept. 2011. Accepted for presentation.

[31] U. Hönig and W. Schiffmann. A parallel branch–and–bound algorithm for computing optimal task graph schedules. In *Grid and Cooperative Computing*, pages 18–25. Springer, 2004.

[32] K. E. Iverson. A programming language. In *AIEE-IRE: Joint Computer Conference*. ACM, 1962.

[33] A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. In *ICFP: International Conference on Functional Programming*. ACM, 2013.

[34] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *ICFP: International Conference on Functional Programming*. ACM, 2010.

[35] Khronos OpenCL Working Group. *The OpenCL Specification*, 2010.

[36] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[37] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Parallel architectures and compilation techniques*. IEEE Press, 2013.

[38] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *International Symposium on Computer Architecture*. ACM, 2010.

[39] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Symposium on Operating Systems Principles*. ACM, 2011.

[40] G. Mainland and G. Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Haskell Symposium*. ACM, 2010.

[41] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *ICFP: International Conference on Functional Programming*. ACM, 2013.

[42] D. Merrill, M. Garland, A. Grimshaw, D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Principles and Practice of Parallel Programming*. ACM, 2012.

[43] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: a retargetable, dynamic compiler and embedded language. In *International Symposium on Code Generation and Optimization*, 2011.

[44] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *Symposium on Networked Systems Design and Implementation*. USENIX Association, 2009.

[45] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, 2008.

[46] T. N'takpé and F. Suter. Critical path and area based scheduling of parallel task graphs on heterogeneous platforms. In *Parallel and Distributed Systems*. IEEE, 2006.

[47] NVIDIA. *CUDA C Programming Guide*, Oct. 2012.

[48] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*. IEEE, 2012.

[49] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: a generator for platform-adapted libraries of signal processing algorithms. *International Journal on High Performance Computing Applications*, 18:21–45, 2004.

[50] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *Principles of programming languages*. ACM Request Permissions, 2013.

[51] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *ACM SIGPLAN Notices*, volume 39. ACM, 2004.

[52] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Symposium on Graphics Hardware*. Eurographics Association, 2007.

[53] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance traps in OpenCL for CPUs. In *Parallel, Distributed and Network-Based Processing*. IEEE, 2013.

[54] J. Siek, I. Karlin, and E. Jessup. Build to order linear algebra kernels. In *Parallel and Distributed Processing*, 2008.

[55] The MathWorks, Inc. Matlab.

[56] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*. Springer-Verlag, 2002.

[57] Z. Wang, L. Zheng, Q. Chen, and M. Guo. CAP: Co-scheduling based on asymptotic profiling in CPU+GPU hybrid systems. In *Programming Models and Applications for Multicores and Manycores*. ACM, 2013.