Converting Data-Parallelism to Task-Parallelism by Rewrites

Purely Functional Programs across Multiple GPUs

Bo Joel Svensson Michael Vollmer Eric Holk Trevor L. McDonell Ryan R. Newton
Indiana University, USA
{joelsven,vollmerm,eholk,mcdonelt,rrnewton}@indiana.edu

Abstract

High-level domain-specific languages for array processing on the GPU are increasingly common, but they typically only run on a single GPU. As computational power is distributed across more devices, languages must target *multiple* devices simultaneously. To this end, we present a compositional translation that fissions data-parallel programs in the *Accelerate* language, allowing subsequent compiler and runtime stages to map computations onto multiple devices for improved performance—even programs that begin as a single data-parallel kernel.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Languages, Performance

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

1. Introduction

Modern computing platforms contain a mix of processors—often more than one with vectorized arithmetic (SIMD) units. A high-end laptop has three: a vectorized CPU, a low-power integrated GPU, and a high-power discrete GPU. From a language perspective, we would like to be able to target multiple devices in the same platform with a single language (portability), but we'd further like to write individual programs that are *implicitly partitioned* over devices. Indeed, in this paper we propose a method for targeting multiple devices from a single, data-parallel, purely functional program—that is, not just compiling the same program for multiple platforms, but partitioning the work and assigning components to different devices at runtime.

Like other embedded array languages, Accelerate's internal representation is essentially a directed graph of data-parallel kernels, with edges communicating complete arrays. The data-parallelism inside even one kernel can be stretched across multiple devices. But this kernel-splitting approach runs into difficulties for languages that include mutable data structures and side-effects within their data-parallel code (Section 2.2). *Purely functional* array languages such as Accelerate [10, 30], Nikola [29], Copperhead [9] and Intel ArBB [31], have significant advantages for multi-device purposes. Of course, they have their own challenges as well: in compiling these languages to vectorized code it is critical to employ aggressive *fusion* [14, 21, 30] to eliminate intermediate data-structures.

In this paper we add multi-device capabilities to Accelerate, including using a *fissioning* program transformation during the compilation process to expose latent parallelism. This is a source-to-source transform that yields valid programs in the original language, simply converting implicit intra-kernel parallelism, into explicit, inter-kernel parallelism—data to task parallelism. For example, a fold operation on an array would be replaced by two fold operations over halves of the array.

Previous systems have looked at multi-device execution, but run into limitation related to mutable arrays and memory consistency [7, 25, 27, 34]. Further, these systems are also focused at the individual-loop level. Compilers for purely functional languages like Accelerate deal with large numbers of "loops" (e.g. every map and fold) and depend on inter-loop optimizations such as fusion/deforestation for efficiency. Compared to other approaches, our fissioning transformation makes its decisions early, in the compiler rather than runtime, but one upshot is that it formulates fissioning as a discrete compiler phase whose correctness is easier to reason about. Further, by making multiple copies of a program subgraph explicit in the syntax tree, the fissioning approach can allow "divergent evolution" of those fragments that are destined for different devices, *even* at the level of inter-loop optimizations.

We use the fissioning approach to build an Accelerate *multi-device backend*, which can dynamically split work across multiple devices. In this paper, we make the following contributions:

- 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 test the correctness of these transforms through a PLT-Redex model (Section 3).
- We implement these fissioning transforms as a source-to-source optimization pass for Accelerate, which is guaranteed —via the Haskell type system— to be type-preserving in object programs (Section 4).
- We present a multi-device scheduler that makes Accelerate the first purely functional array language implementation to automatically distribute individual operators across multiple devices, without changing the program semantics (Section 5).
- We assess the performance of our optimization pass and multi-device scheduler through a number of benchmarks, showing speedups on two GPUs as high as $2.4\times$ over a single GPU (Section 6).

FHPC'15, September 3, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3807-3/15/09
http://dx.doi.org/10.1145/2808091.2808093

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 $100\times$ speedups, attaining good performance requires highly idiomatic programs that are work intensive and require expert knowledge to produce.

The most popular frameworks for programming GPUs are CUDA [33] from NVIDIA, and its open-standard competitor, OpenCL [24]. Both are extensions of the C or 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 [33].

2.2 Multi-device Partitioning: Language Issues

In CUDA and OpenCL programming, the kernels created by the user are indivisible tasks that run on a single GPU of the user's choosing. In related research, the "single kernel, multiple device" (SKMD) approach has already been attempted with the OpenCL language [27], where it can yield improved performance. Yet this is a challenging prospect because OpenCL allows arbitrary side effects from any 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 [27]; or (3) use a runtime technique to *merge* writes from multiple threads in different memories.[1]

Even with these techniques to handle kernel side effects, matching the semantics of OpenCL operations such as global barriers and atomic instructions has proved infeasible. For example, Lee et al. [27] 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 [9, 10, 29–31]. These languages usually still allow reads at arbitrary, data-dependent array indices, which poses challenges for multi-device distribution. Because of immutability, however, *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-emphasizes the need for arbitrary array indexing in favor of implicit data-access patterns. This leaves us in a good position to begin executing these languages on multiple, distributed memory devices.

2.3 Array Languages, Generally

Array-oriented languages have been around for a long time, including APL [22], Matlab [40], 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* [3] and NESL [4, 5]. Nevertheless, today's hardware environment has inspired a renaissance. 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 [31], GPUs [9, 10, 12, 29], or either one [36]. There has also been plenty of focus on code generation for more narrow domains, such as stream-processing [32, 41] and for specific algorithms [19, 35, 38]. This trend is bolstered by general improvements in DSL *embedding* (meta-programming), such as improvements in sharing observation[2] and AST representation [1, 2, 8]. There have also been advances in array languages specifically, supporting new program transformations and scheduling approaches [9, 31] and techniques for optimized code generation [13, 38].

At its core, a typical array DSL provides a way to compile a pipeline or graph of data-parallel operators—*e.g.* `map`, `filter`, `fold`—into parallel code. Typically, there are no language abstractions *separating* the operators in the pipeline. That means there are no function calls left at the array level, 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 embedded DSLs occupying the other. The Accelerate embedded language—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 transforming this graph and generating target-platform code for each vertex in the final graph. It is this graph that we will expand through vertex splitting and map onto multiple devices. We begin by introducing the Accelerate language in the following section.

2.4 The Accelerate Language

Accelerate [10, 30] is a small language for computations over regular, multidimensional arrays. Embedded in Haskell, it exposes data-parallel combinators on 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`:

[1] The technique has proved useful in many domains [6, 28], but has both significant runtime overhead and, in the OpenCL case, relies on relaxed memory consistency.

[2] This refers to the recognition of common subexpressions (CSE) in the target language via inspection of meta-language in-heap data structures [20].

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.

	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-level Conditionals	yes	no	yes	yes	yes

```
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 [10, 23] 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.

The type signature for `fold` 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 is enforced because `Array` itself is not in `Elt` and thus there is no way to embed an `Acc` inside an `Exp`. This stratification statically excludes *nested, irregular* data parallelism, enforcing a flat data-parallel model.

We give a representative grammar of the Accelerate language in Figure 1. Overall, the collective operations in Accelerate are based on the scan-vector model [11, 37], including 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 [10, 30] for more information.

As 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 move body =
          sfoldl (λacc next → acc .+. accel body next)
                 (vec 0)
                 (constant Z)
                 bodies
    in
    map move bodies
```

array exps	æ	::=	use $(\mathcal{C})$ | map $(\lambda x.\, e)$ æ | generate σ $(\lambda x_0 \ldots x_n.\, e)$ | let p = æ in æ | a | (æ, æ) | zipWith $(\lambda x\, y.\, e)$ æ æ | fold $(\lambda x\, y.\, e)$ e æ | backpermute σ $(\lambda x.\, e)$ æ | permute $(\lambda x\, y.\, e)$ æ $(\lambda x_0 \ldots x_n.\, e)$ æ | reshape σ æ | slice σ æ | replicate σ æ
scalar exps	e	::=	let $p = e$ in e | x | c | $prim(e_0, \ldots, e_n)$ | $e_0?(e_1, e_2)$ | $(e_0, \ldots, e_n)$ | $\ldots$
patterns	p	::=	x | (p, p)
variables	x, a		
dim or hole	s	::=	e | $\square$
full shape	σ	::=	$[]$ | $[s_0 \ldots s_n]$
const	c	::=	$1, 2, \ldots$
arrconst	$\mathcal{C}$	::=	$[c_0, \ldots, c_n]_\sigma$

Figure 1. Grammar of the core Accelerate constructs.

The core data-parallel structure of the implementation is a `map` over all bodies. The function being mapped over all bodies perform a sequential fold operations, `sfoldl`, of a function computing acceleration between two points over all bodies. Thus, forces are computed between every pair of bodies. The acceleration between a pair of point masses is then calculated as:

```
accel :: Exp Position   -- The point being accelerated
      → Exp Position   -- Neighboring point
      → Exp Accel
accel body1 body2 =
   if x1 == x2 && y1 == y2 && z1 == z2
   then constant (0, 0, 0)
   else acc
  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 compared to what a version in plain Haskell would look like. In this case, as pattern matching cannot be overloaded, `lift` and `unlift` is used to pack and unpack expressions into and out of tuple constructors.

3. Fissioning Rewrite System

Our goal in this paper is to compile an Accelerate program into *separately* executable components, in order to fully utilize the com-

Currently Implemented

$$
\begin{array}{rcl}
\mathsf{fold}\ f\ e\ \text{æ} & \Rightarrow & \mathsf{let}\ (x,y) = \mathsf{split}_{\text{-}1}\ \text{æ}\ \mathsf{in} \\
 & & \mathsf{zipWith}\ f\ (\mathsf{fold}\ f\ e\ x)\ (\mathsf{fold}\ f\ e\ y) \\
\mathsf{map}\ f\ \text{æ} & \Rightarrow & \mathsf{let}\ (x,y) = \mathsf{split}_i\ \text{æ}\ \mathsf{in} \\
 & & \mathsf{concat}_i\ (\mathsf{map}\ f\ x, \mathsf{map}\ f\ y) \\
\mathsf{generate}\ \sigma\ f & \Rightarrow & \mathsf{concat}_i\ (\mathsf{generate}\ \sigma[\sigma_i := \lfloor \sigma_i/2 \rfloor]\ f, \\
 & & \quad \mathsf{generate}\ \sigma[\sigma_i := \lceil \sigma_i/2 \rceil] \\
 & & \quad\quad (\lambda x_0 ... x_n .\ f x_0 ... \lfloor x_i + \sigma_i/2 \rfloor ... x_n)) \\
\mathsf{replicate}\ \sigma\ \text{æ} & \Rightarrow & \mathsf{let}\ (x,y) = \mathsf{split}_i\ \text{æ}\ \mathsf{in} \\
 & & \mathsf{concat}_{\mathsf{newIndex}(\sigma,\ i)}\ (\\
 & & \quad \mathsf{replicate}\ \sigma\ x, \\
 & & \quad \mathsf{replicate}\ \sigma\ y) \\
\mathsf{zipWith}\ f\ \text{æ}_1\ \text{æ}_2 & \Rightarrow & \mathsf{let}\ (x_1,y_1) = \mathsf{split}_i\ \text{æ}_1\ \mathsf{in} \\
 & & \mathsf{let}\ (x_2,y_2) = \mathsf{split}_i\ \text{æ}_2\ \mathsf{in} \\
 & & \mathsf{concat}_i\ (\mathsf{zipWith}\ f\ x_1\ x_2, \mathsf{zipWith}\ f\ y_1\ y_2) \\
\mathsf{backpermute}\ \sigma\ f\ \text{æ} & \Rightarrow & \mathsf{concat}_i \\
 & & \quad (\mathsf{backpermute}\ \sigma[\sigma_i := \lfloor \sigma_i/2 \rfloor]\ f\ \text{æ}, \\
 & & \quad \mathsf{backpermute}\ \sigma[\sigma_i := \lceil \sigma_i/2 \rceil] \\
 & & \quad\quad (\lambda x_0 ... x_n .\ f x_0 ... \lceil x_i + \sigma_i/2 \rceil ... x_n)\ \text{æ}) \\
\mathsf{use}\ [c_0 \ldots c_n]_\sigma & \Rightarrow & \mathsf{concat}_0\ (\mathsf{use}\ [c_0 \ldots c_{\lfloor n/2 \rfloor}]_{\sigma[\sigma_0 = \lceil \sigma_0/2 \rceil]}) \\
 & & \quad\quad (\mathsf{use}\ [c_{\lceil n/2 \rceil} \ldots c_n]_{\sigma[\sigma_0 = \lfloor \sigma_0/2 \rfloor]})
\end{array}
$$

Additional Legal Rules

$$
\begin{array}{rcl}
\mathsf{fold}\ f\ e\ \text{æ} & \Rightarrow & \mathsf{let}\ (x,y) = \mathsf{split}_0\ \text{æ}\ \mathsf{in} \\
 & & \mathsf{concat}_0\ (\mathsf{fold}\ f\ e\ x, \mathsf{fold}\ f\ e\ y)
\end{array}
$$

Figure 2. Fission rewrite rules. One rule application fissions one data-parallel combinator.

pute capability of a machine. Because Accelerate programs are (task-parallel) DAGs of purely functional array operators, some programs will *already* be suited for using multiple devices, simply by distributing the intermediate array operations over multiple devices. However, this cannot generally be relied upon, and in fact many useful programs are—after fusion optimizations—a single kernel.

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 multiple GPUs. We implement fissioning through a non-deterministic rewriting system as described in Section 3.1. We further validate the correctness of these rules by implementing a semantic model of Accelerate and the fissioning system in PLT Redex (Section 3.2).

3.1 The Rewrite System

Figure 2 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-supported devices. 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. Our implementation currently supports fissioning fold,map, and generate, and other operators are supported via transformation to generate-like deferred arrays (Section 4).

The rules in Figure 2 make frequent use of splitting and concatenation operations, as well as manipulating the shapes of arrays. *Split* divides an array into two halves along a given dimension, indicated by a subscript on the split operator. If a has shape $[\sigma_0\ \sigma_1\ \sigma_2\ \sigma_3]$—from "outermost" (left) to "innermost" (right)—then $\mathsf{split}_1\ a$ produces a tuple of arrays (b, c), with $\langle b \rangle = [\sigma_0\ \lfloor \frac{\sigma_1}{2} \rfloor\ \sigma_2\ \sigma_3]$ and $\langle c \rangle = [\sigma_0\ \lceil \frac{\sigma_1}{2} \rceil\ \sigma_2\ \sigma_3]$, where $\langle b \rangle$ and $\langle c \rangle$ denote the shape of b and c. We use split_{-1} as a shorthand for splitting on the innermost dimension of an array.

Similarly, concatenation combines two arrays along a certain dimension. A key observation is that concatenation is the inverse of splitting:

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

These definitions allow zero-sized arrays, *e.g.* $\mathsf{split}_0\ [v] = ([], [v])$ and $\mathsf{split}_0\ [] = ([], [])$. Neither of these operations are primitive in the original Accelerate, but they are straightforward to add as library functions. Our Redex model treats these as primitives for simplicity.

In our formal notation we shall treat shapes as arrays, so if $a = \left[\begin{smallmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{smallmatrix}\right]$, then $\langle a \rangle_0 = [3\ 2]_0 = 3$. The *rank* of an array is the number of dimensions (*i.e.* $\langle\langle a \rangle\rangle_0$). For a shape σ, we use the notation $\sigma[\sigma_i := n]$ to define a new shape with the ith dimension in σ replaced by n. For example, given $\sigma = [1\ 2\ 3]$, $\sigma[\sigma_1 := 4]$ would be $[1\ 4\ 3]$. 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 traditions established by many dynamically typed array languages such as APL and Matlab.

The general strategy for most rules is to split the input arrays in half, apply the operation to both halves and then combine the results into a single array. As an example, consider the expression $\mathsf{fold}\ (+)\ 0\ a$. The second fold rule splits along the outermost dimension, meaning the rule would split a into $x = [10\ 20\ 30]$ and $y = [40\ 50\ 60]$. Folding the two halves yields $[60]$ and $[150]$, and then concatenating these yields the correct result of $\left[\begin{smallmatrix} 60 \\ 150 \end{smallmatrix}\right]$. On the other hand, the first version of the fold rule splits along the innermost dimension, splitting a into $\left[\begin{smallmatrix} 10 \\ 40 \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} 20 & 30 \\ 50 & 60 \end{smallmatrix}\right]$. The two sub-folds would produce $\left[\begin{smallmatrix} 10 \\ 40 \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} 50 \\ 110 \end{smallmatrix}\right]$, so these two arrays must be combined using a zipWith instead of a simple concatenation.

Cases such as replicate require more care because replicate *increases* the rank of its input. Let us consider the expression $\mathsf{replicate}\ [2\ \square]\ b$ with $b = [1\ 2]$, which makes two copies of b along the outermost (in this case vertical) dimension. This expression evaluates to $\left[\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}\right]$ Let us consider a potential naive fission rule for replicate, which simply splits the array along a dimension and then concatenates the replicated result along the same dimension. In this case, splitting b along dimension 0 gives $[1]$ and $[2]$. Replicating these halves yields $\left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} 2 \\ 2 \end{smallmatrix}\right]$. If we then concatenate along dimension 0, we would get:

$$\begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \end{bmatrix}$$

Instead, we need to concatenate along dimension 1, since the replicate command inserted a new outermost dimension. The $\mathsf{newIndex}(\sigma, i)$ clause in the replicate rule in Figure 2 accounts for the shifting of dimension identifiers due to replication. See Figure 5 for a formal definition of newIndex.

3.2 Testing with PLT Redex

As we have just seen, there are some subtleties to the fissioning rules, especially in the presence of changing ranks. To increase our confidence in the fissioning rules, we developed a model in PLT Redex [16]. Developing this model was motivated in part by bugs we found in the first draft of our fissioning rules. Modeling the semantics in PLT Redex allowed us to design and debug fissioning rules with less effort than implementing them in the full Accelerate compiler. The model serves both as a semantics for the Accelerate language as well as a way to explore the fissioning rules. While the PLT Redex model is not a full proof of correctness, it has validated correct behavior in a test suite of 28 tests, exploring all possible fissioning and evaluation choices for these programs. These rules were tested on arrays of up to three dimensions, and is sufficient to exercise all of the rewrite rules. Our model is available at `https:`

$$
\begin{aligned}
F ::= &\ [] \\
&\mid (\textsf{map}\ f\ F) \\
&\mid (\textsf{let}\ (x\ F)\ ae) \mid (\textsf{let}\ (x\ ae)\ F) \\
&\mid (\textsf{zipWith}\ f\ F\ ae) \mid (\textsf{zipWith}\ f\ ae\ F) \\
&\mid (\textsf{fold}\ f\ e\ F) \\
&\mid (\textsf{backpermute}\ \sigma\ f\ F) \\
&\mid (\textsf{permute}\ f\ F\ f\ ae) \mid (\textsf{permute}\ f\ ae\ f\ F) \\
&\mid (\textsf{reshape}\ \sigma\ F) \\
&\mid (\textsf{slice}\ \sigma\ F) \\
&\mid (\textsf{replicate}\ \sigma\ F) \\
&\mid (\textsf{concat}\ i\ F\ ae) \mid (\textsf{concat}\ i\ ae\ F) \\
&\mid (\textsf{tuple}\ F\ ae) \mid (\textsf{tuple}\ ae\ F) \\
&\mid (\textsf{split}\ i\ F)
\end{aligned}
$$

Figure 3. Fission contexts.

`//github.com/iu-parfunc/accelerate-redex`. Here we will discuss the salient aspects of our model.

We start by defining a language and reduction relation for Accelerate programs defined by the grammar in Figure 1 and then we create a reduction relation to define the semantics of the Accelerate language. Our semantic model makes a strict separation between the array-level language and the scalar language. The overall structure of the program is determined by high-level array operations (e.g. map, fold), while the scalar language describes operations on individual elements of an array which are driven by the array operators. The array-level and scalar-level languages have different environments, so scalar functions cannot access array variables directly. To simplify the semantics, we use a traditional substitution based system for array level bindings and use an environment passing interpreter to evaluate scalar expression as a single operation.

Next we extend this base language with extra operators that are useful for fissioning. These operators include split and concat operators, as well as fst and snd operators to project from the tuples produced by split.

Armed with split and concat primitives, we define a set of fission contexts, which describe points where a fission transformation is applicable. which are similar to evaluation contexts except they describe points where a fission transformation is applicable rather than traditional program evaluation [17]. This allows us to describe the fission rules simply by adding more clauses to the reduction relation. The fission contexts are given in Figure 3.

The fission rules implemented in the PLT Redex model are shown in Figure 4 and a set of helper functions are defined in Figure 5.[3] The rules implemented here match the full set of legal rules described in Figure 2. Many of these rules are nondeterminisic—they can fission across any dimension of the inputs. PLT Redex ensures that any combination of fission reductions preserves the behavior of the program.

It is critical that programs transformed by our rewrite system evaluate to the same result as without the fissioning rules.

The full reduction relation may apply at any point during program evaluation. This is different from the implementation in Accelerate, where all fissioning occurs before program execution begins. Freely mixing evaluation and fissioning *subsumes* the ahead-of-time method that we implemented, so any evidence our model finds of correctness also extends to our chosen application of the rewrite rules.

Finally, we validate our model by running it on a number of test cases. Our test cases include programs to ensure the semantic model matches our notion of what Accelerate programs should mean, as well as programs designed to test each of the fissioning rules in isolation. We inspected a number of execution traces manually to ensure the fissioning rules behaved as expected. In each case, we ensure that all execution traces of a program using the fissioning rules result in the same final value being computed as the unfissioned program.

[3] For readers unfamiliar with Racket's ... pattern syntax, the ellipsis indicates that the preceding pattern should be matched zero or more times. As an example, a rule such as $(x \ldots y) \rightarrow (x \ldots)$ would rewrite $(1\ 2\ 3\ 4)$ as $(1\ 2\ 3)$. This is similar to the * operator in regular expressions.

4. Implementing Fission in Accelerate

The fissioning rules shown in Figure 2 were implemented in the Accelerate front-end via a type-preserving source-to-source translation of the AST. This transformation takes a valid Accelerate program and produces a new program by splitting it into additional operations. Crucially, this translation pass is applied after the program has undergone kernel fusion optimization, so that fission does not interfere with critical fusion optimizations.

Consider the following program `map f xs`. The fissioning transformation will split the `map` operation into two `map` operations, each computing over half of the input array `xs`, together with a final step to combine the partial results. When this code is executed, these two new `map` operations can, potentially, be executed in parallel on different devices. While the formalism of Section 3 emphasizes defining a space of legal fissioning moves, our prototype implementation simply traverses the entire program and fissions each operator once, and thus guaranteeing that there is no bottleneck in the program with *insufficient task parallelism*, provided that the array sizes are large enough to saturate the parallelism of the individual compute devices.

The primary concerns when implementing fissioning were the following:

- The fissioning transformation should be *safe*: the fissioning transformation should produce valid, type-safe Accelerate programs without changing their original meaning.
- The fissioned code should be *efficient*: whenever possible, the fissioning pass should optimize the code it generates to avoid extra run-time overhead in the form of copying or allocation.

Our implementation achieves these two goals in the following way. For the first goal, the type-preservation guarantee of the Accelerate compiler is verified statically by the Haskell compiler, and the fissioning pass retains this same methodology. For the second goal, the fissioning pass manipulates *delayed* array representations in the abstract syntax, and attempts, when possible, to merge the code to *split* an array with the code that *generates* that array.

4.1 Safe Fissioning

In Accelerate, the type of an array encodes both its element type as well as its rank (number of dimensions). For example, `Array (Z:.Int) Float` denotes a one-dimensional array of single-precision floating-point numbers. To safely implement our fissioning rules, we must ensure that the array is greater than zero dimensions, for our fissioning rules to apply. Thus we need to produce a type witness for the shape of the array. We do this by matching on the shape of the array, which—being of non-zero dimension—must be of the form `(sh :. Int)`. Once we know that, we are able to apply the logic of each fissioning rule, as formulated.

All of the necessary program logic for splitting and concatenating arrays in different situations is inlined directly into the typed AST by the fissioning pass. For example, the code for a concat is filled in as *generate* node of its own. In doing this, the fissioning pass must convince the Haskell type-checker that the modified program remains type safe. In particular, because the nameless ASTs in Accelerate carry scalar and array environments *in their types*, the fissioning pass must prove that the resulting program is well-typed

```
F[(fold (λ (x y) e) e_0 ae)]  ⟶ F[(let (t (split -1 ae))                                [F-fold-inner]
                                     (zipWith (λ (x y) e)
                                              (fold (λ (x y) e) e_0 (fst t))
                                              (fold (λ (x y) e) e_0 (snd t))))]
                                 where (s ... s_n) = shape-of[[ae]], s_n > 1

F[(fold (λ (x y) e) e_0 ae)]  ⟶ F[(let (t (split 0 ae))                                 [F-fold-outer]
                                     (concat 0
                                             (fold (λ (x y) e) e_0 (fst t))
                                             (fold (λ (x y) e) e_0 (snd t))))]
                                 where (s_0 s ...) = shape-of[[ae]], s_0 > 1, rank[[ae]] > 1

F[(map (λ (x) e) ae)]  ⟶ F[(let (t (split ⟨(s_1 ...)⟩ ae))                             [F-map]
                                     (concat ⟨(s_1 ...)⟩
                                             (map (λ (x) e) (fst t))
                                             (map (λ (x) e) (snd t))))]
                                 where (s_1 ... s s_2 ...) = shape-of[[ae]], s > 1

F[(generate (s_1 ... s s_2 ...) (λ (x_1 ... x x_2 ...) e))]  ⟶ F[(concat i            [F-generate]
                                     (generate (s_1 ... ⌊s ÷ 2⌋ s_2 ...)
                                               (λ (x_1 ... x x_2 ...) e))
                                     (generate (s_1 ... ⌈s ÷ 2⌉ s_2 ...)
                                               (λ (x_1 ... x x_2 ...)
                                                  (let (x x + ⌊s ÷ 2⌋)
                                                    e))))]
                                 where i = ⟨(s_1 ...)⟩, ⟨(s_1 ...)⟩ = ⟨(x_1 ...)⟩, s > 1

F[(replicate σ ae)]  ⟶ F[(let (t (split i ae))                                         [F-replicate]
                                     (concat newIndex[[σ, i]]
                                             (replicate σ (fst t))
                                             (replicate σ (snd t))))]
                                 where (s_0 ... s_i s_1 ...) = shape-of[[ae]], i = ⟨(s_0 ...)⟩, s_i > 1

F[(zipWith (λ (x y) e) ae_1 ae_2)]  ⟶ F[(let (a (split i ae_1))                         [F-zipWith]
                                     (let (b (split i ae_2))
                                       (concat i (zipWith (λ (x y) e) (fst a) (fst b))
                                                 (zipWith (λ (x y) e) (snd a) (snd b)))))]
                                 where (s_0 ... s_i s_1 ...) = shape-of[[ae_1]], i = ⟨(s_0 ...)⟩, s_i > 1, a, b fresh

F[(backpermute (s_0 ... s s_1 ...) (λ (x_0 ... x x_1 ...) e) ae)]  ⟶ F[(concat i (backpermute (s_0 ... s_2 s_1 ...) (λ (x_0 ... x x_1 ...) e) ae)  [F-backpermute]
                                             (backpermute (s_0 ... s_3 s_1 ...)
                                                          (λ (x_0 ... x x_1 ...)
                                                             (let (x x + s_2) e))
                                                          ae))]
                                 where s_2 = ⌊s ÷ 2⌋, s_3 = ⌈s ÷ 2⌉, i = ⟨(s_0 ...)⟩, ⟨(s_0 ...)⟩ = ⟨(x_0 ...)⟩, s > 1

F[(use (array (s_0 s ...) c_0 ... c_1 ...))]  ⟶ F[(concat 0 (array (⌊s_0 ÷ 2⌋ s ...) c_0 ...)   [F-use]
                                             (array (⌈s_0 ÷ 2⌉ s ...) c_1 ...))]
                                 where ⟨(c_0 ...)⟩ = ⌊s_0 ÷ 2⌋ × shapeSize[[(s ...)]], valid-array[[(array (s_0 s ...) c_0 ... c_1 ...)]], s_0 > 1
```

Figure 4. Fissioning rules as implemented in PLT Redex.

```
shape-of[[(array shape c ...)]]            = shape
shape-of[[(use ae)]]                       = shape-of[[ae]]
shape-of[[(generate σ (λ (x ...) e))]]     = σ
shape-of[[(let (x ae_0) ae_1)]]            = shape-of[[ae_1]]
shape-of[[(zipWith f ae_1 ae_2)]]          = shape-of[[ae_1]]
shape-of[[(fold (λ (x y) e) e_0 ae)]]      = (s_0 ...)     where (s_0 ... s) = shape-of[[ae]]
shape-of[[(backpermute σ f ae)]]           = σ
shape-of[[(permute f_1 ae_0 f_2 ae_1)]]    = shape-of[[ae_0]]
shape-of[[(reshape σ ae)]]                 = σ
shape-of[[(slice σ ae)]]                   = filter-shape[[σ, shape-of[[ae]]]]
shape-of[[(replicate σ ae)]]               = expand-shape[[σ, shape-of[[ae]]]]
shape-of[[(fst (tuple ae_1 ae_2))]]        = shape-of[[ae_1]]
shape-of[[(snd (tuple ae_1 ae_2))]]        = shape-of[[ae_2]]
shape-of[[(concat i ae_1 ae_2)]]           = (s_0 ... s_1 + s_2  s_3 ...)     where (s_0 ... s_1 s_3 ...) = shape-of[[ae_1]], (s_0 ... s_2 s_3 ...) = shape-of[[ae_2]], i = ⟨(s_0 ...)⟩
shape-of[[_]]                              = #f
```

Figure 5. The shape-of metafunction, which is used by the fissioning rules in Figure 4.

and well-scoped even with the introduction of new bindings and the modification of some existing bindings.

Additionally, the fissioning pass must maintain some general invariants present in the internals of the Accelerate compiler and enforced by its types. As an example, the scalar functions that parameterize array operations are "half closed": they are closed in the scalar environment, but open in the array environment. This is one invariant that is and must be maintained through the fissioning program transformation. While the compiler comes short of full formal verification, having the ability to statically verify many invariants of a newly-added compiler pass gave us more confidence in the safety and correctness of the fissioning pass than we would have had if it did not preserve types in this way. Further, when combined with the testing approach used in our PLT Redex model, this gives us high confidence that we don't insert bugs into programs while optimizing for multi-device execution.

4.2 Efficient Fissioning

Our fissioning rules described in Section 3 assume functions for splitting and concatenating arrays. This level of abstraction is convenient for explaining the meaning of the rewrite rules, but it hides some complexity that is present in the implementation.

In the Accelerate AST, arrays can either be *manifest* or *delayed* [30]. A delayed array is represented by a shape, and a function that maps indices to expressions. As the name indicates, the actual creation of the array is delayed, so that it can be inlined into a later computation.

Rather than implementing a split function that takes an array in memory then allocates two new arrays to fill, the fissioning pass directly manipulates the representation of delayed arrays.

At an example, consider the `dotp` program from Section 2.4. Following the standard Accelerate compiler pipeline, this program will be fused into a single operation, with the `zipWith` computation embedded into the `fold` operation as a delayed array. Without fissioning, the program looks like:

```
dotp xs ys =
  fold (+) 0
    (Delayed (intersect (shape xs) (shape ys))
             (λix → (xs!ix) ∗ (ys!ix)))
```

With fissioning, this same program splits the single `fold` into two `fold`s, and introduces a new `zipWith` to combine the partial results. The non-fissioned code had the optimization of doing the work of the `zipWith` inside the `fold` kernel, and fissioning piggybacks on this optimization by pushing its work inside the delayed array structures as well. The end result of this transformation will have the following form:

```
dotp xs ys =
  let s1 = fold (+) 0
             (Delayed
               (let sh = intersect (shape xs) (shape ys)
                    n  = {- first partition size -}
                in indexTail shape :. n)
               (λix → (xs!ix) ∗ (ys!ix)))
      s2 = fold (+) 0
             (Delayed
               (let sh = intersect (shape xs) (shape ys)
                    n  = {- second partition size -}
                in indexTail shape :. n)
               (λix → let ix' = {- index with offset -}
                      in (xs!ix') ∗ (ys!ix')))
  in
  zipWith (+) s1 s2
```

Because of the delayed array representation, array splits introduced by fissioning carry relatively little overhead. At run-time, the splits are themselves delayed arrays.

That is not to say that fissioning cannot increase the runtime overhead costs in programs that it is applied to. For example, while fissioning introduces new kernels that can be evaluated independently, these partial results must be combined, which implies a single synchronization point in the program. More generally, executing fissioned code may impose additional data transfer cost from the host to the device, or between devices themselves. We explore these issues in Section 6.

5. Implementing a Multi-device Runtime

The fissioning pass described in the previous sections is designed to expose task parallelism in Accelerate programs. Of course, some programs expose sufficient task parallelism even *without* fissioning, and the runtime we describe in this section can handle those programs too—with or without fissioning applied. Indeed, an advantage of the rewrite-based approach is that fissioning becomes an orthogonal concern from scheduling.

Our runtime system thus takes on the traditional role of a *scheduler*, as found in operations research [43]. Namely, it must:

- Identify tasks to execute, consisting of DAG of one or more collective array operations;
- Select a device to execute the task on;
- Copy any dependencies to the device, if not already present; and finally
- Compile and execute the operation(s).

We implement the multi-device runtime system on top of the existing Accelerate CUDA backend, reusing as much of that (well-tested) backend as possible. The major difference between the traditional and new runtime system has to do with where and how inter-task dependencies are managed.

The CUDA backend relies on the CUDA driver to track inter-dependencies between tasks and transfer events, starting kernels on the hardware only after their dependencies are met. In this way it can essentially offload entire graphs of actions onto the CUDA driver. This is possible, because it has no need to *hold back* any work: as all tasks would eventually run on the same device.

The multi-device runtime system cannot take this approach. If it overcommits too many tasks to one device via the CUDA driver, then it commits too early and loses the ability to load balance those tasks onto other devices. Thus our new runtime must track task dependencies and completion explicitly on the Haskell side, and be judicious about how much to commit to any one device.

5.1 The Runtime System

The decision of which task to place on which device is performed by a scheduler component of the runtime system. We use a standard approach, as in previous work [18], where one worker thread serves as a representative of each CUDA device. Each of these proxy threads waits to be assigned work items. This worker thread is in control of both launching kernels on, and copying data to, the associated device. On the other hand, recovering results *from* the device(s) is a different matter, and will be addressed in a moment. The heart of the scheduling algorithm on each thread is thus the following simple loop:

```
deviceLoop dev done work = do
  -- Is any work available ?
  workItem ← takeMVar work
  case workItem of
    ShutDown → putMVar done Done >> return ()
    Work w → do
      w                  -- Perform work, then indicate
      registerAsFree dev -- that the device is now free
      deviceLoop dev done work
```

5.2 Task Extraction Heuristic

Because every (non-fused) array-level operation becomes a CUDA kernel, these are the *atoms* of our task graph. While each such kernel could be scheduled independently, our current prototype uses a heuristic that follows the nesting structure of the AST itself. That is, the heuristic is based on the structure of let bindings in the program. In short, the *outer spine* of let bindings is preferred as the axis along which to subdivide tasks among devices. Our prototype fissioning pass is tuned to work well with this heuristic.

As an example, in the following nesting of let bindings, the runtime system will partition out as tasks: `a1`, `a2`, `a3` and `result`. Ideally these tasks are both independent and expensive.

```
let a1 = ... in
let a2 = ... in
let a3 = ...
in result
```

On the other hand, given the structure below the runtime system will partition out only three tasks: `a0`, `a3` and `result`, scheduling `a1` and `a2` onto the same device as part of a single task.

```
let a0 =
    (let a1 = ... in
     let a2 = ...
     in
     result1)
in let a3 = ...
in  result
```

The Accelerate compiler backend transformations provides an AST for which this heuristic is suitable. Although not a good in general, this heuristic works well for our specific use case in Accelerate. We intend to explore other strategies in the future.

5.3 Execution of Tasks

When a task is identified to run on a device, the algorithm proceeds as follows:

- Create the asynchronous result array structure, which are returned immediately so that other tasks can refer to, and wait on, arrays computed by the current task.
- Fork a *task thread* (distinct from per-device worker threads):
  1. Find dependencies of the task.
  2. Wait for dependencies to be computed.
  3. Compute a per-device affinity score based on how many of the dependency arrays are present on each device.
  4. Ask a scheduler for the best device available, given the computed affinity scores.
  5. Create a work item and execute it on the assigned device.

The work item created in step 5 above is sent to the worker thread associated with the device selected by the scheduler. The work item sent over performs the following sequence of operations:

- Transfer dependency arrays to associated device.
- Compile task (if not already cached)
- Execute task.
- Write results into the asynchronous arrays created earlier and mark those arrays as completed.

All the operations performed at this stage make use of the already existing Accelerate CUDA back-end. We add a information about which memory contains which arrays for the purpose of the scheduler and runtime system, but the copying is performed by existing Accelerate CUDA functionality. When it comes to compiling and executing tasks, we even gained the benefits of the Accelerate CUDA back-end's caching of already compiled code.

5.4 Scheduling of Tasks

This section outlines the scheduling algorithm used in the benchmarks in Section 6. The scheduler we implemented is multi-threaded, spawning task-threads on demand.

Each task thread starts out by requesting the use of a device from the runtime system. To do this it blocks on a queue of *device tokens*, which represent an entitlement to use a currently free device. Upon receiving a token, the task thread atomically performs the following to choose a specific device:

- If there is only one device available, pick it.
- If more than one device is free, sort the devices based on affinity score and pick the highest scoring device.
- The device is marked as busy and is sent the task work item.

6. Evaluation

In this section we evaluate the performance of our implementation on a selection of benchmarks. Our benchmarks aim to test both weak and strong scaling behavior of our system. The selected benchmarks both illuminate the strengths of the approach, as well as identify several areas for future improvement.

Benchmarks were conducted using two Tesla C2075 GPUs (compute capability 2.0, 14 multiprocessors = 448 cores at 1.15 GHz, 5GB RAM) backed by two 6-core Xeon X5660 CPUs (64-bit, 1.6 GHz, 200GB RAM) running GNU/Linux (Red Hat 4.4.7-9). We used GHC-7.8.3 and NVCC-6.0. Results are generated using criterion[4] via linear regression.

6.1 Weak Scaling Benchmarks

Weak scaling benchmarks consist of a synthetic benchmark, *megapar*, of completely independent tasks and exposing many opportunities for task parallelism. We also test the N-body simulation where the data is duplicated onto each device. These tests constitute a sanity test for our runtime system.

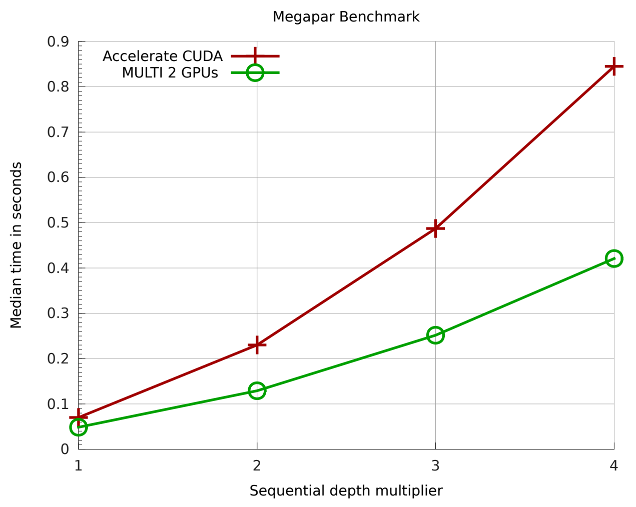
The Megapar benchmark consists of a parallel map over a wide (2M element) array, where the scalar function consists of a long running loop. The figure compares overall execution time as we increase the iteration count of the scalar loop. The duplicated N-body benchmark represents scaling when running the N-body calculation on two *independent* sets of bodies.

Both of these benchmarks expose completely independent tasks, and the results from each device do not need to be combined in a final step. These benchmarks demonstrate speedups of close to $2\times$, validating that our system makes effective use of both GPUs in these sanity checking benchmarks (Figure 6).

6.2 Strong Scaling Benchmarks

With our runtime system validated, our strong scaling benchmarks exercise our approach to fissioning via our rewrite rule system,

[4] `http://hackage.haskell.org/package/criterion`



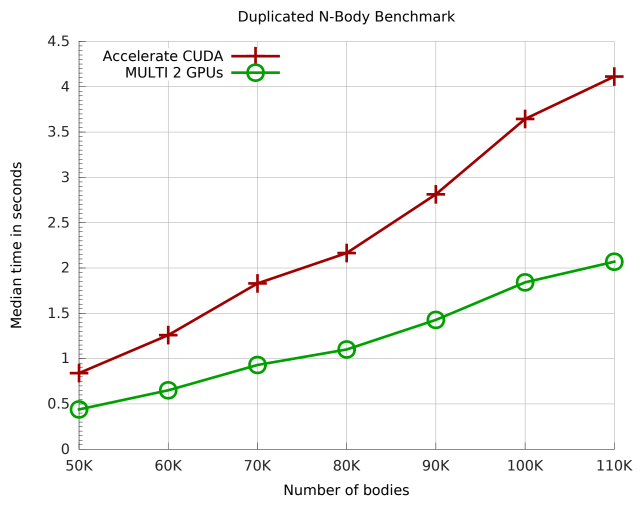
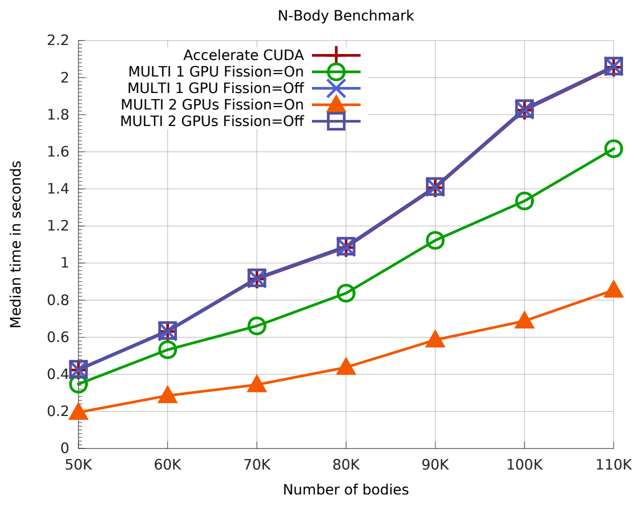


Figure 6. Weak scaling benchmarks.



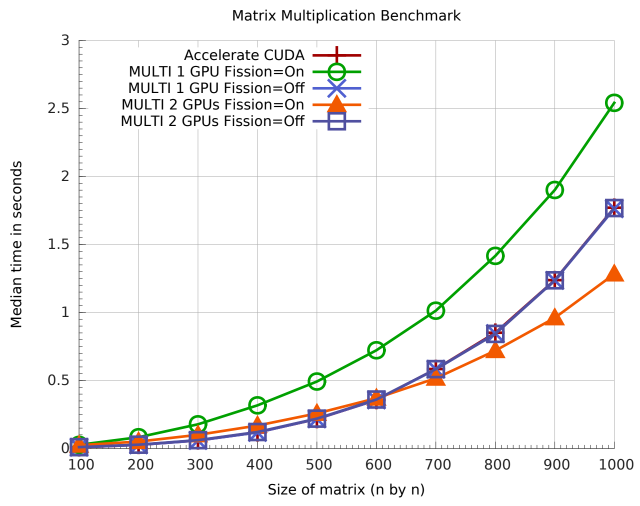


Figure 7. Strong scaling benchmarks.

where both our new multi-GPU backend as well as the existing single-GPU backend execute the same problem size. As further sanity checks of our runtime and fissioning systems, we execute both the fissioned and unfissioned program through our new backend on both one and two GPUs. Executing the original program through our multi-device runtime on one GPU provides a measure of any additional overhead introduced by our scheduler compared to the existing Accelerate CUDA backend.

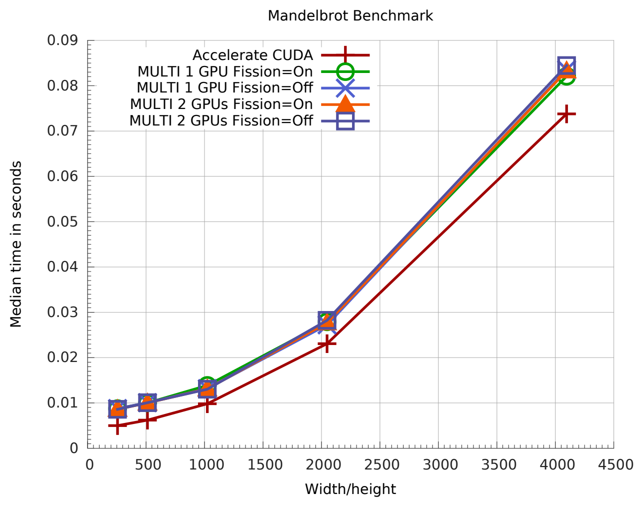
It is important to remember that, unlike the benchmarks from the previous subsection, here the devices need to cooperate to compute a program over a single data set. That is, if we split an array to execute its pieces over multiple devices, we must then combine those partial results to reach the final value.

N-Body The N-body example simulates Newtonian gravitational forces on a set of massive bodies in 3D space, using the basic $O(n^2)$ algorithm shown in Section 2.4.

The N-body benchmark has a high compute to data-transfer ratio, and benefits greatly from fissioning and multi-device execution. Executing the original or fissioned program through our multi-device scheduler on a single GPU yields performance similar to the existing Accelerate CUDA backend. Executing the fissioned program over 2 GPUs yields a maximum speed of $2.4\times$ (Figure 7). At these sizes, splitting the working set in half on each GPU allows the bodies to fit entirely into the 2MB cache, resulting in a benefit even on a single GPU. At larger sizes, once the cache is exhausted in both the single and multi-GPU cases, the performance delta stabilizes at $2.0\times$.

Matrix Multiplication The matrix multiplication benchmark achieves a modest speedup at large matrix sizes. This benchmark requires a large partial result to be communicated between devices and combined, which we observe is only worthwhile at larger input sizes (Figure 7). Additionally, this program exercises the fold fissioning rules, so the combination step is more expensive relative to the other benchmarks we consider, such as N-body, which only need to concatenate their partial result vectors.

Mandelbrot The Mandelbrot set is generated by sampling values c in the complex plane, and determining whether under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ that $|z_n|$



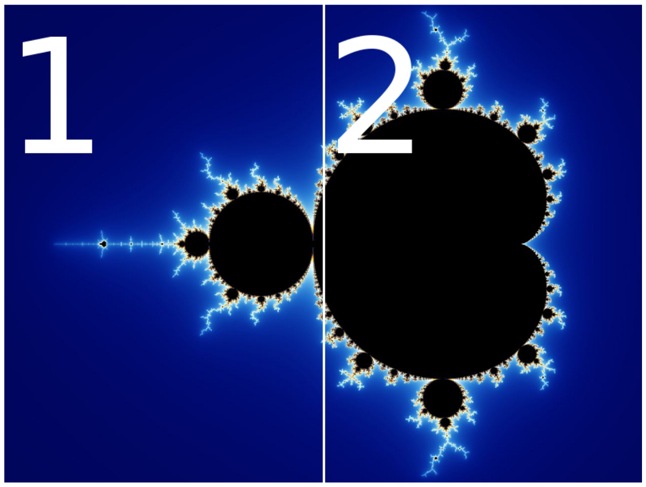


Figure 8. The Mandelbrot program is an example of an unbalanced problem. The area marked as 1 is considerably cheaper to compute than the area marked 2.

remains bounded however large n gets. This is an example of an *unbalanced workload*, as the time to compute each point c on the complex plane varies. Figure 8 shows a visual representation of the result, and indicates how our fission rules distributed the iteration space between the two devices. Areas that are colored black take the longest time to compute. Thus, we can see that the second piece (right half) is considerably more expensive to compute compared to the first. We discuss possible approaches to resolving this problem in Section 8.

7. Related Work

There have been several efforts to build languages or tools for more effectively leveraging multiple GPUs. Many projects either rely on a programmer to specify some or all of the details of how their program is distributed across devices, or only distribute sections of their input vector or array to identical programs on multiple GPUs.

SkelCL [39] is an extension to OpenCL that allows programmers to specify at a high level the general strategy to use when distributing a computation across multiple devices. It abstracts away lower level concerns like the details of copying data.

Delite/LMS [36] is a library-based parallelization framework for DSLs in Scala that allows specifying complex optimizations in a modular manner. The Delite code generator is able to target multicore CPUs and GPUs, and demonstrates impressive performance on both.

Wu et al. [42] describe kernel fusion and fission operations, to be used in optimization of data warehousing applications. Their intent is to schedule smaller data-parallel kernels to hide PCIe transfer time.

SkePu [15] is a C++ template library for single and multi-GPU systems based on code skeletons, for operations such as map and reduce. SkePu is capable of launching array computations on multiple devices.

AMGE [7] is a CUDA source to source compiler that augments executables with information about array access patterns. The AGME runtime system uses this access pattern information and run CUDA applications across multiple GPUs. The same kernel is launched on all GPUs and unified addressing is required to ensure that all data is reachable from all devices at once.

PLT Redex has seen adoption in several projects as a way to explore programming language semantics. Kuper et al. [26] also use PLT Redex in the context of a language designed for parallelism.

8. Discussion and Future Work

Much work remains to fully explore the potential of single kernel, multiple device embedded languages, 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 embedded language. Our prototype demonstrates the possibility of transparently using two GPUs without changing the user's high-level source code.

We have demonstrated that our fissioning transformation and multi-device runtime perform well on a set of benchmarks — namely N-body and matrix multiplication— while the Mandelbrot program was used to highlight the problems with our current, fixed domain decomposition. We plan to integrate our approach to fissioning with an auto-tuning mechanism that would explore different fissioning and decomposition strategies, and thus automatically find the best split point for unbalanced workloads such as Mandelbrot.

While we have used the system on unbalanced workloads, we have not tested it on a system with GPUs of different performance characteristics. Doing this, as well as heterogeneous GPU/CPU decompositions, is left as future work.

Additionally, we believe fissioning may be exploited to automatically handle datasets that do not fit in the memory of a single GPU. We leave this exploration to future work.

We have modelled our system of fission rewrite rules in PLT Redex and tested their correctness. We found this to be extremely useful, allowing us to test and debug the rewrite rules, thereby increasing our confidence in the system as a whole and helping us in weeding out incorrect rewrite rules. However, important future work is to formally prove the correctness of our rewrite rules.

Acknowledgments

This work was supported by NSF grant XPS-1337242.

References

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

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

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

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

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

[6] 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.

[7] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. Hwu. Automatic execution of single-GPU computations across multiple GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 467–468. ACM, 2014.

[8] 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.

[9] 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.

[10] 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.

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

[12] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30. ACM, 2012.

[13] 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.

[14] 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.

[15] J. Enmyren and C. W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10. ACM, 2010.

[16] M. Felleisen, R. B. Finder, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[17] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. .

[18] 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.

[19] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

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

[21] 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.

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

[23] 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.

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

[25] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama. Integrating multi-gpu execution in an openacc compiler. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13. IEEE Computer Society, 2013. .

[26] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270. ACM, 2014.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

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

[34] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. .

[35] 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.

[36] 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.

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

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

[39] M. Steuwer, P. Kegel, and S. Gorlatch. Towards high-level programming of multi-GPU systems using the skelcl library. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1858–1865, May 2012.

[40] The MathWorks, Inc. Matlab.

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

[42] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 2433–2442. IEEE, 2012.

[43] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.