Region-based Memory Management for GPU Programming Languages

Enabling Rich Data Structures on a Spartan Host

Eric Holk Ryan Newton Jeremy Siek Andrew Lumsdaine
Indiana University, School of Informatics and Computing, Bloomington IN 47405
{eholk, rrnewton, jsiek, lums}@cs.indiana.edu

Abstract

Graphics Processing Units (GPUs) can effectively accelerate many applications, but their applicability has been largely limited to problems whose solutions can be expressed neatly in terms of linear algebra. Indeed, most GPU programming languages limit the user to simple data structures–typically only multidimensional rectangular arrays of scalar values. Many algorithms are more naturally expressed using higher level language features, such as algebraic data types (ADTs) and first class procedures, yet building these structures in a manner suitable for a GPU remains a challenge. We present a region-based memory management approach that enables rich data structures in Harlan, a language for data parallel computing. Regions enable rich data structures by providing a uniform representation for pointers on both the CPU and GPU and by providing a means of transferring entire data structures between CPU and GPU memory. We demonstrate Harlan's increased expressiveness on several example programs and show that Harlan performs well on more traditional data-parallel problems.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Run-time environments; Optimization; Compilers; Memory management (garbage collection)

Keywords Harlan; GPU; OpenCL; parallel programming; recursion; algebraic data types; first class procedures; compilers; implementation; performance; optimization

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660244

1. Introduction

GPUs have become popular for high performance computing due to the high throughput afforded by their massively parallel architecture. Programming models such as CUDA or OpenCL have somewhat eased the burden of general purpose GPU computing, but these models are still very low-level. Several higher level domain specific languages such as Copperhead [4] and Accelerate [7] have sought to simplify GPU programming so that programmers can ignore the architectural details and focus on their algorithms.

A large proportion of these languages rely extensively on immutable, multidimensional rectangular arrays. The benefits of immutability are clear, aiding the programmer in reasoning about their program and simplifying debugging with deterministic semantics. The benefits extend to language implementers as well, since compilers have much more freedom to optimize programs when there are fewer observable effects to preserve. Constraining programs to rectangular arrays also facilitates implementation, as these data structures map naturally onto data parallel hardware.

Unfortunately, these languages are also limited. Many problems do not fit nicely into rectangular arrays and would be better be served by more complex structures like trees and graphs. Consider one implementation of control flow analysis on the GPU [22]. While this algorithm showed an impressive speedup, achieving the speedup required the authors to go to great lengths to cast the problem as a linear algebra problem. We intend to simplify GPU implementations of problems such as these by providing even higher level control and data structures in a language that can run on the GPU.

This is the purpose of our language, Harlan [14], which enables richer control flow constructs by providing first class functions (`lambda`). In fact, procedures created in Harlan can move freely between the GPU and CPU. This ability can be used, for example, to allow a computation started on the CPU to move to the GPU and vice-versa. Harlan also supports richer data structures, such as non-rectangular arrays and ADTs. Harlan resembles Scheme in syntax and features lightweight operators for data parallel computation, such as

`kernel` and `reduce`. Below is an example demonstrating how a Harlan programmer might express a matrix-vector product of `M` and `xs`.

```
(kernel ((m M))
  (reduce + (kernel ((y m) (x xs)) (* x y))))
```

The key enabling technology behind Harlan is its **region-based memory system**. Harlan's rich data structures are complicated for GPUs because of their heavy use of pointers. Traditionally working with pointer structures in a GPU language has been either explicitly disallowed or it necessitated serializing the structure on the host side before transferring it to the device memory. Using regions sidesteps these issues by constructing data in a form that is readily transferred between host and device memory. Furthermore, this is done without requiring programmer involvement, as the assignment of data to regions can be determined automatically (region inference [28]). Data can then be moved in units of regions, which enables efficient data transfer between the CPU and GPU memory.

We make the following contributions:

- We present Harlan, a language for GPU computing that supports functional programming and a rich set of data structures (Section 3).
- We show how the use of region-based memory management in Harlan enables rich data structures and supports efficient transfer of data structures between host and device memory (Section 4).
- We describe the implementation of Harlan, including the region system, scheduling of memory transfers, and several optimizations (Section 5).
- We show how supporting rich data structures makes it more natural to express more problems for the GPU by presenting several more extensive examples (Section 6).
- We show that Harlan can be efficiently implemented and yields good performance on a selection of benchmarks (Section 7).

2. Background

2.1 General Purpose GPU Computing

Graphics processing units are specialized processors whose development has primarily been driven by the demand for stunning visuals in video games. As GPUs have become more powerful, they have evolved into general-purpose data-parallel processors and are seeing increased use in scientific computing and other disciplines.

GPUs consist of several processing units, called Streaming Multiprocessors (SMs) by NVIDIA, which are analogous to cores on a traditional CPU. Each of these can track many thread contexts at once, and thus high performance GPU kernels are written in terms of thousands of threads. GPUs make little use of speculation and out of order execution to reduce latency and instead hide latency by switching between threads as their dependencies are satisfied.

The most popular framework for programming NVIDIA GPUs is CUDA [20], which presents the programmer with the illusion of a virtually unlimited set of threads. These threads are grouped into warps that execute in lock step, and the warps are grouped into blocks. OpenCL presents similar concepts using different terminology [19].

Memory management on GPUs is more complicated than on CPUs. Most of the memory falls into the global memory category, which resides in off-chip DRAM. GPUs also provide a small amount of local memory for each SM, which is akin to L2 cache on CPUs. Changes to global memory are visible to all CUDA threads, while local memory changes are only visible to a single thread block. Local memory is very fast, but limited in size. Writing efficient GPU codes requires judicious use of local and global memory. The GPU memory is almost always distinct from the memory of host CPU, meaning applications must carefully schedule transfers over the relatively slow PCI Express bus. We say data that is directly accessible to the CPU as residing in host memory, while data that is stored in the GPU memory is said to reside in device memory.

Programming models for CPUs and GPUs have traditionally been quite different, and yet the actual CPU and GPU architectures are becoming more similar. Newer generations of CPUs typically include more powerful vector processing capabilities, such as with Intel's AVX instructions. Likewise, GPUs are adopting features that were once CPU-only, such as hardware-managed L2 caches. This suggests that a single unified programming model for both CPUs and GPUs is achievable and code that is written to run well on the GPU can 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 tradeoff between simultaneous multithreading and out of order, speculative execution.

Harlan facilitates write-once, run anywhere data parallel programming by abstracting away most of the details of the particular hardware. Programmers specify at a high level what must be done in computational kernels, and the language implementation is responsible for best mapping these kernels to the available hardware. Indeed, because Harlan compiles to OpenCL, it is trivial to switch between executing kernels on the GPU and executing them on the CPU.

2.2 Region-Based Memory Management

Region-based memory management is a technique for managing the lifetime of objects by assigning them to regions. Allocating from a region is typically cheap, and all objects in a region are deallocated at once. The ML Kit compiler used region-based memory management (RBMM) as a general garbage collection strategy [28]. In this system, objects were automatically assigned to regions based on their in-

ferred lifetimes. An important safety property is that a region must outlive all references to any object contained within it.

Regions are sometimes also known as arenas, and can be used to amortize the deallocation overhead over many objects. For example, deallocating a linked list takes $O(N)$ time when using `malloc` and `free`, but when all nodes are allocated in an arena, deallocation becomes a constant time operation–one simply needs to deallocate the whole region at once. This approach is used to improve performance in software such as the Apache Web Server.

Programming languages such as Rust [24] and Cyclone [13] use a version of regions to enforce safety properties by ensuring that an object outlives any references to the object. In these cases, it is important to distinguish *regions* from *lifetimes*. A lifetime is the period that an object must remain live to ensure all reference to it remain valid. Lifetimes may be analyzed statically and provided they are proven safe, they do not have any effect at runtime. Regions, on the other hand, do have a runtime impact; they are an actual object that is allocated and deallocated at runtime, and other objects are represented as pointers into their region. Using this terminology, ML Kit and Harlan use regions, while Cyclone and Rust use lifetimes.

3. The Harlan Programming Language

Harlan is a language for data-parallel computing that was designed from the beginning with GPU implementation in mind. Its syntax is reminiscent of Scheme's, and Harlan features a hygienic macro system similar to Scheme's `syntax-rules`. Unlike Scheme, however, Harlan is statically typed. Harlan allows for the rapid development of GPU kernels through its support of nested data parallelism, rich data structures.

3.1 Data-parallel Computing

Parallel operations are indicated using `kernel` and `reduce` forms. For example, a dot product in Harlan is performed as follows:

```
(reduce + (kernel ((x xs) (y ys)) (* x y)))
```

The `kernel` form in this snippet performs the elementwise multiplication of the vectors `xs` and `ys`. Kernels are similar to Haskell's `zipWith` function. The result of this kernel is a new vector with the same length as the input vectors. This result is used by `reduce`, which computes the sum of all of the elements in its vector.

Harlan programs use the same language inside kernels as outside, although there are some restrictions. For example, kernels cannot perform IO. Currently, compilation unceremoniously fails when the programmer violates these restrictions, but Harlan's type system could be extended to detect and report these violations in a programmer-friendly way.

The parallel constructs in Harlan can be arbitrarily nested, allowing us to write a parallel matrix-vector product as follows.

```
(kernel ((row M))
  (reduce + (kernel ((r row) (x xs)) (* r x))))
```

The Harlan compiler is responsible for generating code to implement the high level specification provided by the user. Harlan compiles to C++ and OpenCL. Kernels roughly map to OpenCL kernels, while the reduction operation is implemented as a macro in Harlan's runtime library. Compiling to OpenCL allows Harlan to easily target both GPUs, multicore CPUs and other accelerators. In the future, Harlan may use both types of hardware simultaneously.

3.2 Data Structures

Harlan provides several data types in addition to the usual complement of scalar types. The first of these is the `vector` type, which is an immutable ordered collection of elements of a single type. Harlan's parallel operators all work over vectors. Vectors can contain any data type, including other vectors. Nested vectors need not be the same size, so the following code snippet is perfectly acceptable:

```
(vector (vector 1)
        (vector 1 2)
        (vector 1 2 3))
```

More interesting is the fact that Harlan supports ADTs. These allow Harlan programs to naturally handle tree-structured data and directed acyclic graphs (DAGs). For example, suppose we were writing an interpreter for the λ-Calculus. Below is an example data type we could use to represent expressions.

```
(define-datatype Expr
    (variable int)
    (lambda Expr)
    (app Expr Expr))
```

In this type, we have chosen to represent variables with deBruijn indices. Having defined the type for expressions, we can write a simple non-terminating program as follows:

```
(app (lambda (app (variable 0) (variable 0)))
     (lambda (app (variable 0) (variable 0))))
```

As another example, Figure 1 shows a simple N-Body simulation that uses both ADTs and nested parallelism. Note that the core algorithm is quite short; the bulk of the code is defining auxiliary helper functions for manipulating `point3-t` objects in parallel. Here, the function `nbody` computes the forces on each object. The inner kernel computes the force on body `i` due to each other body, and these are accumulated using a reduction with `point-add`. For simplicity, we assume the gravitational constant is 1.

3.3 First Class Procedures

Harlan leverages its support for ADTs to provide first class procedures in the form of `lambda`. Procedures in Harlan are completely interchangeable between host and device code. Programs can create procedures on the CPU and apply them on the GPU, and vice-versa. Harlan's `lambda` follows Scheme's syntax:

```
(define (nbody bodies)
    (kernel ((i bodies))
      (reduce point-add
        (kernel ((j bodies))
          (let* ((diff (point-diff j i))
                 (d (point-mag diff)))
            (if (< 0 d)
                (point-div diff (* (* d d) d))
                (point3 0 0 0)))))))
```

Figure 1: A simple N-Body simulation written in Harlan.

```
(lambda (x) (+ x 1))
```

The expression above creates a procedure that adds one to its argument. We can then apply this procedure to many items in parallel using a `kernel`:

```
(let ((f (lambda (x) (+ x 1))))
  (kernel ((i (iota 10)))
    (f i)))
```

Notice that in this case the procedure, `f`, is created in the host portion of the code but then applied inside of a kernel which runs on the device.

In the same way, kernels can produce procedures which are applied on host:

```
(let* ((funcs (kernel ((i (iota 10)))
                (lambda (n)
                  (* n i))))
       (f (vector-ref funcs 5)))
  (println (f 24)))
```

Here, we use a kernel to create ten different procedures. We use `vector-ref` to extract one of these procedures and then print the result of applying it to 24. This program prints `120`.

In effect, using `lambda` this way lets the programmer build vectors of lightweight objects. They automatically capture a subset of the available data, without requiring the programmer to define a new data type which may only be used for one return value.

Of course, vectors of functions can be inputs to a kernel as well. We take advantage of that in this "multiple-program, single data" style program:

```
(let ((funcs (make-some-procedures))
      (x (make-a-datum)))
  (kernel ((f funcs))
    (f x)))
```

This style of program does run the risk of increasing branch divergence, a common performance pitfall in GPU programming. Recent advances in JavaScript compilation, such as [12], suggest that even if language allows very irregular programs, the control and data flow are often quite predictable in performance-critical sections. We explore the issue of branch divergence in Section 7.4.

Given that `lambda` is a building block for a variety of control and binding structures, the flexibility from having procedures that can move freely between computational devices will enable a variety of novel parallel program structures.

3.4 Macros

One useful aspect of higher level language features is that they can be used to build new useful abstractions. This is particularly true when the syntax is extensible. To this end, Harlan features a hygienic macro system similar to Scheme's `syntax-rules`. Many of Harlan's built-in language features are defined as macros in the standard library. This style is useful for programmers because they can add new language forms as it is convenient for their application, as we do in the ray tracing example in Section 6.1, and it simplifies the language implementation because the compiler has a smaller set of core forms to handle.

4. Region-based Memory Management in Harlan

Harlan's rich data structures pose several implementation challenges. Often, data will be created in CPU code, such as by reading from a file. The data must then be transferred into the GPU memory. For regular structures, such as dense rectangular arrays, this is a simple memory copy. For tree structures, such as arise with Harlan's ADTs, finding all nodes of the tree in memory involves traversing the whole tree. Furthermore, OpenCL makes no guarantees about the stability of pointer values between kernel invocations, making it impossible to have pointers between OpenCL memory objects.

We solve these problems in Harlan using a region-based memory management system. The type inference process assigns data structures to regions. This guarantees that all elements of a data structure can be easily located. Before invoking a kernel, Harlan copies the regions containing each of the kernel's data structures into the GPU memory, rather than attempting to precisely move each individual element. This approach enables other features as well, such as being able to allocate memory from the GPU.

It is worth noting that several of the limitations imposed by OpenCL, such as forbidding pointers to pointers and recursive functions, are not present in CUDA and thus many of the implementation challenges facing Harlan would not exist if Harlan generated CUDA instead of OpenCL. However, CPU/GPU systems are one particular instance of a distributed system. Many of these challenges will arise in other distributed contexts, such as when it is necessary to move complex data structures or procedures between nodes in a cluster. We invite the reader to view Harlan's region system as a set of techniques that may apply to other distributed systems as well.

4.1 Region Inference

Region inference happens alongside type inference using an approach similar to that of [28]. The Harlan type system separates types into *value types* and *region-allocated types*.

Value types are objects that are passed by value while region-allocated types are represented as pointers into the heap. Region-allocated types carry region parameters, which specify which region they are allocated from.

When the type inferencer encounters a region-allocated type, such as a vector, it creates a new free region variable. In the course of type inference, the algorithm may find new constraints requiring two values to be in the same region. In this case, the two types' region variables are unified. At the end of type inference, the compiler replaces the region unification variables with concrete region variables and binds these variables by inserting `let-region` expressions. The `let-region` expression must enclose all uses of a given region. Harlan does this by inserting `let-region` expressions at the entrance to functions that will bind any regions free in the body that do not escape through the return value.

Consider the following example:

```
(define (foo)
  (let* ((v1 (vector 1 2 3))
         (v2 (vector 4 5 6))
         (v  (vector v1 v2)))
    (vector-ref v 1)))
```

The type inferencer may first determine that `v1` is a vector of integers, and assigns it the type `(vec` ρ_1 `int)`, meaning a vector of integers in region ρ_1. In a similar way, the type inferencer will assign `v2` the type `(vec` ρ_2 `int)`.

Things are slightly more complicated for `v`. The type inferencer knows `v` must be a vector. Furthermore, vectors must contain values of uniform type. Yet, `v1` is a vector in region ρ_1 and `v2` is a vector in region ρ_2. Thus, the type inferencer assigns `v` the type `(vec` ρ_3 `(vec` ρ_1 `int))` and adds the constraint that $\rho_1 = \rho_2$.

The `let*` expression returns `(vector-ref v 1)`, and we can see from the type of `v` that this means the whole `let*` expression has type `(vec` ρ_1 `int)`, which incidentally also becomes the return type of function `foo`.

Having inferred types and region constraints, the compiler now inserts a `let-region` expression enclosing the body of `foo`. There are two distinct variables to consider: ρ_1 and ρ_3. Because ρ_1 escapes the function, it cannot be bound here. Thus, the compiler only binds ρ_3, assigning it a concrete region variable which we will call `r1`.

At this point, the intermediate representation of our function looks something like this:

```
(define (foo)
  (let-region [r1]
    (let* ((v1 (vector [ρ1] 1 2 3))
           (v2 (vector [ρ1] 4 5 6))
           (v  (vector [r1] v1 v2)))
      (vector-ref v 1))))
```

By convention, we use square brackets to denote region variable bindings and region arguments. Thus, `(vector [r1] v1 v2)` explicitly indicates that the vector is allocated from region `r1`.

One might ask why regions are stored as part of the type, rather than the runtime representation of the object. Region references can be viewed as a pair of a region and an offset into that region, so why not represent pointers as two words, one for the region and one for the offset? Instead, pointers in Harlan are simply the offset, and the region portion is determined by the type. The reason is that we would run afoul of OpenCL's injunction against multiple indirection if region references included a pointer to the region. We must be able to find all pointer bases through explicit parameters, rather than discovering them by traversing data structures.

The example above is not finished yet, as we have left the region inference variable ρ_1 unbound! We solve this by allowing functions to be region-polymorphic. The caller will supply a region to `foo`, which specifies where to store the return value. Thus, the final region-inferred version of this function is:

```
(define (foo [r2])
  (let-region [r1]
    (let* ((v1 (vector [r2] 1 2 3))
           (v2 (vector [r2] 4 5 6))
           (v  (vector [r1] v1 v2)))
      (vector-ref v 1))))
```
(1)

Figure 2a shows how these vectors are grouped into regions. Figure 2b shows a diagram of the heap immediately before returning from `foo`, while Figure 2c shows the heap right after returning. Notice that region `r1` is destroyed, but there remains unreachable data in region `r2`. We do not currently perform garbage collection within a region, but doing so would enable Harlan to reclaim this space.

4.2 Region-allocated Types

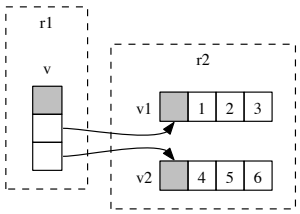
In general, Harlan prefers value types over region-allocated types. This tends to lead to flatter data structures. Although multiple indirection is possible on the GPU, memory references are relatively expensive. Flatter data structures result in less pointer chasing. There are three classes of types in Harlan that interact with the region system: vectors, algebraic data types and closures.

Vectors are region allocated in part because they can be quite large and passing vectors by value could be expensive. In the case of vectors of vectors, `vector-ref` could not be a constant time operation, because the location of each of the child vectors would not be known. Harlan requires the size of value types to be known statically, which is explicitly not the case with vectors. Instead, vectors are represented by constant-size pointers to data of unknown size in regions.

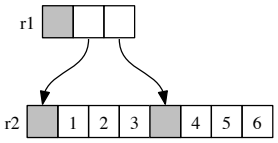
ADTs can be either region-allocated types or value types, depending on the structure of the ADT. For example, in a type such as

```
(define-datatype Number
  (Float float)
  (Int   int))
```

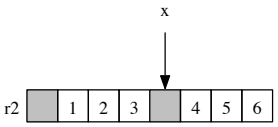
there is no reason to involve the region system at all. Harlan represents this type as a value type, and the size is deter-



(a) This figure shows how the three vectors in (1) are mapped onto regions. The gray cells represent a header containing information such as the vector's length.



(b) The layout of the heap just before returning from function `foo`.



(c) The layout of the heap returning from function `foo`. This assumes the caller has stored the return value in variable `x`. Notice that region `r1` has been destroyed, but `v1` is still in `r2` but unreachable.

Figure 2: Examples showing how data is assigned to and arranged within regions.

mined by the size of the largest variant. On the other hand, the following type is affected by the region system.

```
(define-datatype IntList
  (Cons int IntList)
  (Null))
```

The reason is that the type is recursive. Naively trying to find the size of the largest variant would never terminate, as the list can be of arbitrary length. Instead, the recursive reference to `IntList` is replaced by a reference to a region-allocated `IntList`. After region inference, `IntList` becomes:

```
(define-datatype (IntList [r])
  (Cons int (IntList [r]))
  (Null))
```

An ADT requires a region parameter if any variant is immediately recursive, or if any field in any variant is region-allocated.

ADTs contain exactly zero or one region parameters. We are not aware of any technical reasons to prevent more than one region parameter, but one region parameter does not seem overly restrictive and simplifies the implementation. In a more general scheme, we could imagine, for example, a binary tree type where all the leftward nodes are in one region and all the rightward nodes are in another:

```
(define-datatype (Tree [r1 r2])
  (Node (Tree [r1]) (Tree [r2]))
  (Leaf))
```

The final class of types that are affected by regions are closures.[1] Closures themselves are value types, but they may close over region-allocated data. In order to ensure the application of the closure can find all of the data in its captured environment, the type of a closure must include a region parameter as well. Consider the following:

```
(let ((v1 (vector 1 2 3))
      (v2 (vector 4 5 6)))
  (lambda (b i)
    (if b (vector-ref v1 i)
          (vector-ref v2 i))))
```

Ignoring regions, this example evaluates to a procedure of type `(bool int) -> int`. The procedure uses its boolean argument to decide which vector to return a value from. Clearly, vectors `v1` and `v2` must live as long as the procedure closing over them does, and we ensure this with the region system. One obvious way to do this would be to add as many region parameters as necessary. The examples we have seen so far allow `v1` and `v2` to be in different regions, so assuming they are in regions `r1` and `r2`, we could change the type of the closure to be `[r1 r2](bool int) -> int`.

Now, consider another function:

```
(lambda (b i)
  (if b i (* i 2)))
```

This evaluates to another procedure of type `(bool int) -> int`. It does not close over any region-allocated data, and therefore does not need a region parameter. There is now a problem, however, which is that these procedures both take arguments of the same type and return a value of the same type but are not interchangeable solely because of differences in their environment. Closures are meant to abstract away the environment, and yet here our programmer is left to wonder why seemingly equivalent functions cannot be used in the same location.

For this reason, Harlan gives every closure exactly one region parameter. Thus, both of these examples gain the type `[r](bool int) -> int`. The second function simply ignores its region argument, but this fact is invisible to callers. In the case of the first example, the fact that both vectors are captured by the same procedure means they are now constrained to be in the same region.

In each of these three classes, notice that region parameters are not as much about where the particular value resides, but rather where that value may point.

[1] We will see in Section 5 that this is in part because closures are compiled into ADTs, which are themselves affected by regions.

4.3 Flexibility in Region Assignment

These rules leave a great deal of freedom for assigning objects to regions. At one extreme, all objects could be assigned to a single region. This has obvious disadvantages. Since regions are used as the unit of data transfer between the host and device memories, assigning all data to a single region means all of the program's data must be transferred at once. Furthermore, we do not do garbage collection within regions, meaning large amounts of stale data would accumulate over the run on the program.

At the other extreme, Harlan could try to assign each object to its own region. This results in much more precise data transfer at the cost of having to start many more memory transfers. In our experience, the cost of launching more transfers is minor compared to the cost of transferring more data than necessary (Section 7.1), but the ideal region assignment may lie somewhere in between these two extremes. Harlan's region assignment algorithm tends more towards assigning each object its own region.

One additional degree of freedom is in the placement of `let-region` expressions. Currently, we insert `let-region` expressions at the entrance of each function. Alternatively, the compiler could be more precise and insert `let-region` expressions deeper in the function to further limit how long data remains allocated.

5. Implementation

5.1 Compilation

The Harlan compiler is written in Scheme as a nanopass-style compiler [26]. Programs are read in as S-Expressions and compiled into C++ programs that use OpenCL. We selected OpenCL over CUDA as a compilation target due to its ability to support a variety of devices, including CPUs and both NVIDIA and AMD GPUs. The OpenCL kernel code is compiled once when the Harlan program begins execution, taking it off the critical path for performance-sensitive portions of the program. Very little of the compiler and runtime is specific to OpenCL, however, so a CUDA backend could be developed without much trouble. The passes are roughly divided into a front end, a middle end and the back end.

The front end checks to make sure the input program is well-formed, loads libraries used by the program, performs hygienic macro expansion and, finally, does type and region inference. From that point, the middle end takes over with a series of passes that progressively lower a Harlan program into a C++ program. This involves steps such as converting `match` expressions into a chain of `if`-statements, inserting array bounds checks, imposing kernel calling conventions, rewriting memory references as region references, and converting Harlan kernels into top-level OpenCL kernels. The middle end is also where some simple optimizations take place (Section 5.5). Finally, the back end generates a C++ program from the generated abstract syntax tree and invokes the C++ compiler to produce an executable program.

We now take a more in-depth look at some of the more interesting compiler passes.

ADT Construction This pass compiles the ADTs away from the language. The compiler generates a C struct containing a tag field and a union of all of the variants for the type. Each variant is also given a constructor function, which takes an argument for each field in the variant and returns an instance of the correct type. The compiler desugars `match` expressions into a chain of `if` statements that check the tag and bind the given variables to each field in the variant.

Kernel Dimension Analysis In order to facilitate other optimizations, Harlan's surface level kernel form is lowered into a version that explicitly gives the number of work items in each dimension. This form more closely matches the way kernels function in OpenCL, and also makes it easier to fuse nested kernels into a two dimensional kernel (Section 5.5).

Kernel Flattening OpenCL does not support spawning kernels from inside of kernels, and thus nested kernels in the Harlan source must be removed. We do this by leaving one of the kernels in a nest as a true kernel and converting the rest into sequential code. The two obvious strategies are to leave either the outermost or innermost kernel as a true kernel and sequentialize the rest. We choose to keep the outermost kernel and sequentialize all of the inner kernels in order to have more code running on the GPU with fewer round-trips to the CPU. This heuristic has worked well so far, but it may not always be the best. Larger kernels provide the GPU with more code to use for hiding memory latency, but they can also reduce performance by increasing register pressure [25]. More analysis could be used to determine the best point in a kernel nest to keep parallel.

Explicit Region Reference Insertion Up until this point in the compiler, all types still maintain their region annotations. This phase replaces references to region-allocated data with explicit reads and writes from a given offset in the appropriate region. After this pass, region-allocated types are replaced with generic region pointers with casts inserted as necessary. Regions that appear in function types are converted to additional parameters so that these regions are available for references to that region within the function.

Kernel Hoisting Kernels in OpenCL must be top-level forms, so this pass extracts kernel expressions from within the bodies of Harlan functions and lifts them to top level. This process includes converting any free variables in the kernel body into parameters to the kernel. At this point in the compiler, regions that a kernel references are also treated as free variables. All the kernels in the program are lifted into a special `gpu-module` form, which is compiled into an OpenCL program. This process is essentially the same process as lambda lifting [18]. Any functions that a kernel might reference are also included in the GPU module, enabling kernels to call other functions. OpenCL C is similar

enough to standard C that we can reuse much of the Harlan compiler's backend to generate the OpenCL program.

5.2 First Class Procedures

We implement first class procedures by defunctionalization [23]. Lambda expressions of the same type are compiled into a single ADT combined with a dispatch function. Each variant of the ADT represents each possible class of closure, while the dispatch function contains the code associated with each variant. As an example, consider the following program.

```
(module
  (define (main)
    (let ((c 2))
      (let ((f (lambda (x) (+ 1 x)))
            (g (lambda (y) (* c y))))
        (println* (f 5) (g 5))))))
```

This program would be compiled into something like the following.

```
(module
  (define-datatype lambda-int->int
    (lambda-f)
    (lambda-g int))

  (define (dispatch-int->int closure a)
    (match closure
      ((lambda-f)
       (let ((x a))
         (+ 1 x)))
      ((lambda-g c)
       (let ((y a))
         (* c y)))))

  (define (main)
    (let ((c 2))
      (let ((f (lambda-f))
            (g (lambda-g c)))
        (println* (dispatch-int->int f 5)
                  (dispatch-int->int g 5))))))
```

Notice that we have constructed one new data type and one new function that together are responsible for both `lambda` expressions. The `lambda` expressions themselves are replaced by calls to the appropriate constructors, and applications are replaced by calls to the generated dispatch function with the closure as an explicit argument.

The type system ensures that only closures of the correct type can be applied in each location. We take advantage of this fact to group procedures according to their type.

One downside is that each generated data type is the size of the largest closure of a given type. This has not yet caused problems for us, but if it did, we could mitigate the impact by running a control flow analysis to more precisely limit which closures can flow to each call site.

5.3 Recursive Functions in Kernels

OpenCL explicitly forbids the use of recursive functions in kernels, and some OpenCL compilers fail to terminate in the presence of such functions. Harlan works around this limitation by converting recursive procedure calls to `goto`s with an explicitly managed stack.

We do this by generating a call graph and then using Tarjan's algorithm [27] to find the strongly connected components. Each component represents a set of mutually recursive functions. Harlan combines these into a single function in OpenCL, where each Harlan function corresponds to a label within the large OpenCL function. Parameters to each label are represented as local variables.

Care is needed with the return pointer, since OpenCL also forbids code pointers. Instead, we generate a label at each return point and give each of these labels a unique identifier. We then generate a special return label, which jumps to the correct return point based on the return identifier on the stack. This is analogous to the way we worked around OpenCL's restriction on pointers to code in our implementation of first class procedures.

We also considered implementing recursion by translating sets of mutually recursive functions into continuation passing style. This approach would likely have led to too many continuations being allocated from the same region, so instead we opted for an explicitly managed stack.

Due to the way Harlan implements first class procedures, the generated code may include recursive functions even if the source level program is not recursive. The reason is because the code for all closures of the same type become a single function. If any of these closures call another closure of the same type, then the generated dispatch function will appear recursive. This makes having compiler support for recursion all the more important, as it may not be obvious or even under that programmer's control that a program would appear recursive to OpenCL.

5.4 Regions

Harlan's pointer structures are all written in terms of regions. Conceptually, a heap-allocated object, like a vector or linked list, is represented as a pair of a region and an offset into that region. The region portion is known statically, and so all Harlan heap objects are represented at runtime simply as offsets into a region.

Regions are represented as a block of memory that consists of a header followed by program data. The header stores the current size of the region and an allocation pointer, which points to the end of the allocated data. When data is allocated from a region, the allocation pointer is simply incremented by the size of the object being allocated, and the original value is returned. If the allocation pointer is greater than the size of the region, the Harlan runtime resizes the region to hold the new allocation.

Since all heap-allocated data is accessed relative to a region, these regions must be provided to functions that either accept arguments in regions or return a region-allocated object. In this case, the Harlan compiler inserts an extra parameter for each region a function uses. This is similar to implementing type polymorphism by passing type descriptors [10].

When the region is resident in CPU memory, it may be resized if the allocation pointer moves beyond the end of the region. Harlan currently uses a doubling policy to resize regions when there is not enough room for the requested allocation.

Region resizing is not possible when allocating from within kernels, and in this case the kernel would simply fail with an error indicating that the region did not have enough space available. Future versions of Harlan may choose to resize the region on the CPU side and then retry the kernel to automatically recover from the error.

The region's backing OpenCL buffer may be left unallocated during most of the program, and instead be allocated immediately before the region is needed on the GPU. With this approach, the backing buffer can be freed as soon as the region is no longer live on the GPU as well. Our experience is that allocating and deallocating OpenCL buffers is cheap relative to the cost of transferring the contents of the region between host and device memory, so this technique enables Harlan programs to work with larger working sets, provided not all regions are needed at once by a kernel.

5.5 Optimizations

The Harlan compiler currently performs several simple optimizations. These optimizations let us keep Harlan fairly simple as a language yet still get good performance. We focus on optimizations that affect the structure of kernels and rely on the underlying C++ compiler to perform its standard set of optimizations.

5.5.1 Kernel Fusion

There are two types of fusion optimizations which Harlan performs. The first is applicable when one kernel receives its inputs directly from another, as in the code snippet below.

```
(kernel ((x (kernel ((i is)) (+ i 1))))
  (* 2 x))
```

Here, the two kernels can be combined into a single kernel:

```
(kernel ((i is))
  (let ((x (+ i 1)))
    (* 2 x)))
```

Eliminating the intermediate kernel improves performance by providing more operations for the GPU to use to hide latency and also avoids allocating memory to store the result of the first kernel. A similar optimization is applied in the case of reductions over kernels, such as in:

```
(reduce + (kernel ((x xs) (y yx)) (* x y)))
```

In this case, the compiler would eliminate the temporary result that stores the product of `xs` and `ys`, and instead compute this product while performing the reduction.

A second form of fusion combines two kernels where one is immediately nested inside another into a two-dimensional kernel. This transformation is applicable to cases such as the following fragment from a simple matrix multiplication program.

```
(kernel ((row A))
  (kernel ((col (transpose B)))
    (dot-product row col)))
```

5.5.2 Let Lifting

Let lifting takes advantage of the fact that data in Harlan is immutable and thus tries to lift computations as high as possible to prevent the computations from being needlessly repeated. This is effectively a form of loop-invariant code motion [9]. Lifting computations can transform the code so that more kernels are adjacent to each other, thus increasing the number of kernels that can be fused together. Even in cases where more kernels cannot be fused, let lifting can improve memory transfers by allowing data that is reused by several successive kernels to remain resident on the GPU. This is especially true when combined with the lazy data transfer optimizations.

5.5.3 Lazy Data Transfer

Regions are used as the unit of data transfer between host and device memories. At a basic level, Harlan migrates the entire contents of each region used by a kernel to the device memory before executing the kernel, and upon exiting the kernel Harlan migrates all regions back to the host memory. Several obvious optimizations are possible. First, region transfers can be initiated lazily. Regions needed by a kernel are migrated to the device memory as before, but they remain there until the host code references a portion of that region. Second, regions consist of some amount of live data followed by unallocated space. Because the portion of the region beyond the allocation pointer is not meaningful, Harlan only needs to transfer the portion of the region up to the allocation pointer. Returning a region to the host memory is done in two transfers. The first reads the region header to determine the most up to data value of the allocation pointer, since kernels may have allocated from this region. Then, the remaining portion of the region up to the new allocation pointer is transfered. This approach does introduce incur a slight overhead in initiating two transfers, but in our experience this overhead is negligible in light of the savings from not transferring useless bits. These optimizations together yield a significant improvement in performance.

Furthermore, because data in Harlan is immutable, it is possible for the runtime system to establish fairly tight bounds on the portions of a region that might have changed. A more advanced version of lazy data transfer could take ad-

```
(define (render-image scene origin
                      width height)
  (interpolate-range
   (y 1.0 -1.0 height)
   (interpolate-range
    (x -1.0 1.0 width)
    (let ((dir (unit-length (point3f x y 1))))
      (match (reduce select-closest
                (kernel ((object scene))
                  (object origin dir)))
        ((miss) (point3f 0 0 0))
        ((hit dist color) color))))))
```

Figure 3: A portion of the ray tracing program. This program represents a scene as a vector of procedures that computer the intersection of an object with a ray. The program also makes use of custom syntax in `interpolate-range`, which uniformly samples a range of floating pointer numbers.

vantage of this to more precisely transfer only the changed portions of a region.

6. Extended Examples

We now explore several larger Harlan programs to see how Harlan's higher level features aid in their implementation.

6.1 Ray Tracing

Ray tracing is a way of rendering images that works by simulating the movement of light rays through a scene. In contrast to rasterization techniques, which typically only handle triangles, ray tracers can work with mathematical surfaces directly. For example, rather than approximate a sphere as a triangle mesh, ray tracers can instead use the equation defining a sphere to directly compute the intersection of a ray and that sphere.

We take advantage of this fact and represent objects in a scene as functions. The ray tracer provides an object with a source and a direction for a ray, and the objects reports whether this ray intersects the object. If the ray does intersect, the object also returns the color that should be used for that portion of the object.

A portion of the ray tracer code is given in Figure 3. This code snippet makes use of several of Harlan's features. The `interpolate-range` construct is used to sample a range of floating pointer numbers and a certain number of evenly spaced points, and illustrates the use of macros. In this example, interpolate range maps a pixel in the output image into a point in space in the scene, which is later used to compute the direction of the ray that intersects the given pixel. This construct is implemented as a macro that expands into a kernel, enabling our ray tracer to compute many pixels in parallel.

For each pixel, the ray tracer computes a ray and then tests for intersection with each object in the scene. The reduction with the `select-closest` function finds the nearest intersection, and uses this as the final pixel value. The scene is represented as a vector of objects, which are constructed by functions that return other functions. One example object constructor is given below.

```
(define (make-sphere center radius)
  (lambda (source direction)
    ...compute intersection of the ray and sphere...))
```

Having creating many objects in this fashion, the main rendering kernel applies each of these to a source and direction vector. This method of defining objects allows for easy composition. For example, one might write a function that takes an object as an input and produces an object that scales the input object by some factor.

The results of testing an object for intersection are reported through a simple ADT, given below.

```
(define-datatype ray-result
  (miss)
  (hit float point3f-t))
```

Functions return `(miss)` when the ray does not intersect, and when the ray does intersect they return a `hit` with a distance value (used by `select-closest`) and a color represented as a `point3f-t` value.

This way of structuring a ray tracer has some performance implications, which we evaluate in Section 7.4.

6.2 Breadth First Search

Graph algorithms such as breadth first search are naturally irregular problems, and thus a good candidate for Harlan's native support for irregular data structures. One challenge in constructing graph algorithms is that there are data structures that are easy for programmers to work with, but these do not always give the best performance. The code in Figure 4 is written using an adjacency list representation. The graph is represented as a vector of nodes, each of which contains a vector of all of the nodes they have an edge to. This representation gives a concise breadth first search implementation, but for efficiency we would rather store the graph as a matrix in compressed sparse row (CSR) format. A nice side effect of Harlan's region system is that the adjacency list looks very similar to CSR format in memory.

The algorithm used in Figure 4 is a straightforward level-synchronized breadth first search (BFS). Each node's status is tracked in a vector of colors. White nodes have not been explored, gray nodes are currently under consideration, and black nodes are completed. The kernel processes each node in parallel. If a node is white, then the kernel checks if any of its adjacent nodes are gray, and if so the node itself turns gray. Gray nodes become black, and black nodes stay black. The kernel would then run in a loop until the color vector reaches a fixed point.

```
(kernel ((i (iota (length graph)))
         (c colors))
  (match c
    ((black) (black))
    ((gray)  (black))
    ((white)
     (if (reduce or
           (kernel ((j (vector-ref graph i)))
             (match (vector-ref colors j)
               ((white) #f)
               ((gray)  #t)
               ((black) #f))))
         (gray)
         (white)))))
```

Figure 4: The core of the breadth first search code. The outer kernel applies to each node identifier (given by `(iota (length graph))`) and the current color vector. For nodes that have not yet been visited, another reduction and kernel is used to check if any incoming edges have been visited.

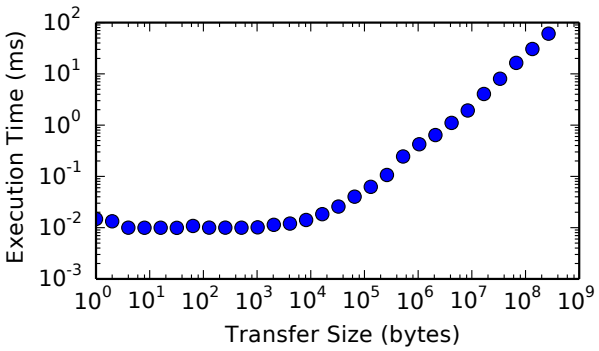


Figure 5: Transfer times between the CPU and GPU memory for buffers of various sizes.

7. Evaluation

In this section, we present performance measurements for several programs in Harlan and compare their performance against a high performance implementation in other systems. We conducted our experiments on Delta and Big Red II. Delta is a GPU cluster within FutureGrid that uses NVIDIA Tesla C2075 GPUs. Big Red II is a new hybrid CPU/GPU cluster featuring NIVIDA Tesla K20 GPUs.

7.1 GPU Memory Performance

We argued in Section 4.3 that the per-transfer overhead is small relative to the time spent doing the actual transfer. Figure 5 shows the total time to transfer data of varying sizes from the host memory to the device memory. The amount of time is relatively flat until about 8KB and afterwards it increases linearly.

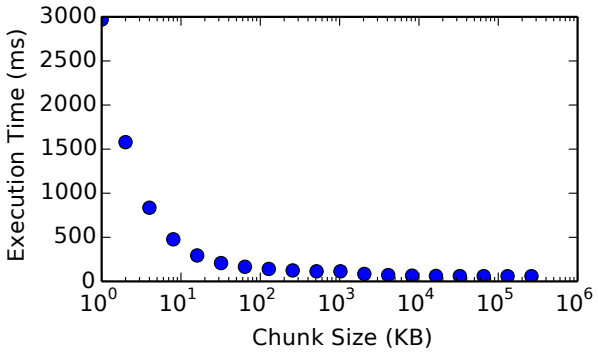


Figure 6: Time to transfer 256MB of data from the CPU to the GPU, dividing the total data into chunks. The per-transfer overhead is minimal when the chunk size is 64KB or greater.

Figure 6 looks at memory bandwidth in a different way, by transferring a total amount of 256MB but breaking it into a number of chunks of various sizes. As expected, small chunk sizes take significantly longer, but after around 64KB the overhead for many transfers is relatively small.

Regions in Harlan start at some minimum size and grow as necessary to accommodate their contents. The data present here suggest that there is no reason to make the minimum region size smaller than 8KB. The small threshold for where the bandwidth costs overtake the latency costs validate our strategy of assigning data into as many regions as possible. Data structures will likely already be larger than this threshold for the increased throughput of the GPU to overcome the cost of transferring the data over the relatively slow PCI-Express bus.

7.2 Microbenchmarks

We start with two micro benchmarks, vector addition and dot product, and compare the Harlan implementation against the CUBLAS equivalent. We use CUBLAS from CUDA 5.5. Figures 7 and 8 show the results from these two tests. The timings include the time to transfer memory as well as the actual kernel execution time. Harlan significantly underperforms CUBLAS for vector addition. We suspect this is due to Harlan introducing more region transfers than are necessary, and that future optimizations can alleviate this problem.

Harlan performs much better on the dot product benchmark, as shown in Figure 8. In this case we perform almost as well as CUBLAS. The performance of a good dot product implementation should be limited by the hardware memory bandwidth, which shows that both of these implementations are performing as well as they should.

Both benchmarks show several discontinuities in the graphs of Harlan's performance. These result from Harlan needing to increase the region size. The remaining graphs show similar discontinuities for the same reason.

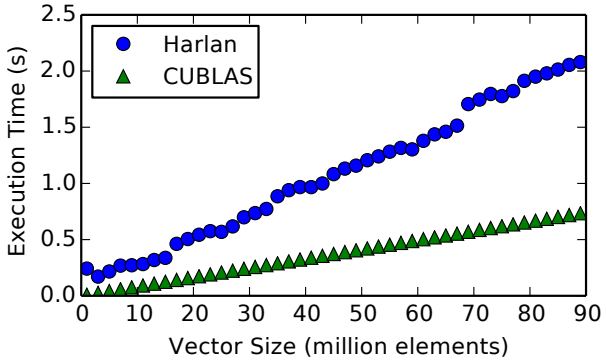


Figure 7: Vector addition in Harlan compared with vector addition in CUBLAS.

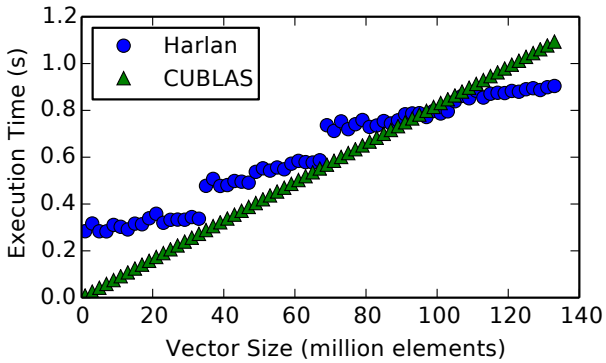


Figure 8: Dot product in Harlan compared with dot product in CUBLAS.

7.3 N-Body Simulation

As an example of a more realistic application, we present results from a naive N-Body simulation in Figure 9. The Harlan version is adapted from the code presented in Figure 1, and the Accelerate version is adapted from the naive N-Body solver in the Accelerate repository. Both programs are idiomatic in their own language. We used version 0.13 of Accelerate. The results show the execution time increasing as the square of the number of bodies, as is expected.

The Accelerate version takes significantly more time than the Harlan version. The reason is that Accelerate and Harlan use different reduction strategies, and Harlan's happens to work better for this particular benchmark. Because the reductions happen inside a kernel, Harlan implements these as sequential loops that all run in parallel, while Accelerate uses parallelism within each reduction.

The benchmarks presented here show that Harlan is competitive with existing GPU programming approaches. It is important to note that Harlan's region system does not impose a performance penalty while increasing the expressiveness of the language.

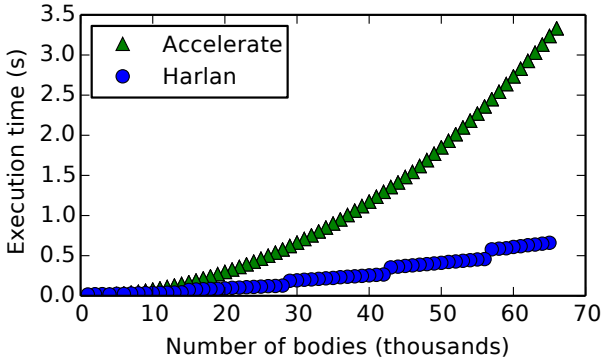


Figure 9: N-Body simulation in Harlan compared with Accelerate.

	Delta	Big Red II
Sorted	7.106s	4.885s
Unsorted	7.092s	4.897s

Table 1: The effect of thread divergence on ray tracing performance. In this particular case, thread divergence does not make a significant impact on the overall execution time.

7.4 Ray Tracing

One potential risk with Harlan programs is that they can lead to code with many more branches, and these branches could lead to poor performance on the GPU. To measure this penalty, we ran two versions of the ray tracing program from Section 6.1. In both versions, we render a randomly generated scene consisting of 100 scaled and translated spheres. The scaling and translation is accomplished by creating wrapper object functions that alter the incoming ray before intersecting the ray with the base surface. For this benchmark, half of the objects were scaled and then translated, while the other half were translated and then scaled. In one variant, objects with the same sequence or transformations are stored together in the scene vector, while in the other variant these are interleaved together. In the case where the scene is sorted, all threads in a block should go the same direction at a branch, while the unsorted case should have more thread divergence.

Table 1 shows the results of running this benchmark on both Delta and Big Red II. We did not see a significant difference in the performance for either variant. This is likely because the parallelism is per pixel, rather than per object, and thus all threads consider the same objects in the same order. This suggests that though Harlan programs have the potential to have poor branch behavior, they will often be structured to minimize these effects.

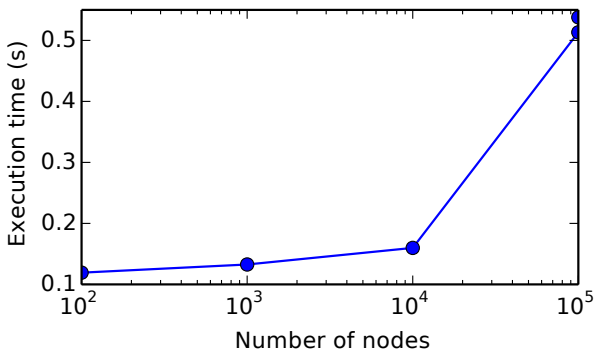


Figure 10: Execution time for BFS traversal of a graph of varying numbers of nodes. Graphs are randomly generated, with each node having 16 edges.

7.5 Breadth First Search

Figure 10 shows the results of running the program from Section 6.2 on randomly generated graphs of various sizes. These results are from running on Delta. The graphs were generated by choosing 16 random edges with a uniform distributed for each node.

8. Related Work

Some of the initial inspiration for this project came from EigenCFA [22]. This project showed that GPUs are applicable to more than just traditional scientific computing problems. EigenCFA also highlighted the limitations on the expressiveness in current GPU languages.

Many languages such as Accelerate [6] and Copperhead [4] provide a higher level of abstraction for GPU programming. These languages enable parallelism through a collection of data parallel operators, such as map, fold and scan. Accelerate makes use of Haskell's type system to enforce properties such as constraints on the shapes of arrays, while Copperhead relies on program analysis to determine these properties. The Delite project is exploring similar with a focus of exploiting heterogeneous parallelism [5].

Regions in Harlan bear some similarity to places in X10, which can also be compiled for the GPU [11]. In both systems, data is assigned to places or regions, but X10 fixes places to a specific device and transfers between places are written explicitly.

Legion uses a logical region system to ensure safety in parallel programs while providing control of data movement through the memory hierarchy to the programmer [1]. Regions are used to identify independent portions of computation that can then be dynamically scheduled on the available resources. Harlan's region system differs significantly in that it is primarily concerned with representing pointer structures. Deterministic Parallel Java also uses regions with an effect system to ensure determinism in parallel programs [3]. While determinism is not an explicit goal of Harlan, the lack of mutable data eliminates many sources of nondeterminism.

Region-based Software Virtual Memory (RSVM), like Harlan, uses regions as its unit of data transfer [17]. RSVM allows arbitrary cross-region pointers by mapping region identifiers to memory locations at runtime using a lookup table. This coupled with transparent swapping allows GPU kernels to work on data sets that do not fit in device memory.

The CPU-GPU Communication Manager (CGCM) manages the transfer of data between the host and device memories in terms of allocation units [16]. These allocation units are similar to Harlan's regions. Many of the communication optimizations in this work would apply to Harlan.

OptiX is a domain specific language for ray tracing [21]. OptiX shaders can use recursion to follow reflected rays. Recursion is implemented through a continuation passing style and trampolining technique. [29] describes a technique for supporting recursion on GPUs by explicitly managing stacks, which is the approach used by Harlan.

Work on compiling Mozilla's Rust for the GPU is similar to this in its focus on enabling higher-level abstractions in GPU kernels [15]. Rust for the GPU supports some `enum` types, but cannot handle pointer structures like those possible in Harlan.

NOVA [8] is a programming language that is very similar to Harlan. NOVA is also a LISP-like language that expresses data parallelism using primitives like `map`, `reduce` and `scan`. Like Harlan, NOVA supports ADTs, first class procedures and recursive functions in kernels. In addition, NOVA supports type-polymorphic functions. One key difference is that NOVA is only used to define kernel code, producing functions that may be called by a host program written in a language like C++. Harlan, on the other hand, defines a language for both the kernels and the host code.

9. Conclusions and Future Work

We have presented an expressive GPU programming language called Harlan. Harlan uses a region-based memory management system to support rich data structures such as trees, non-rectangular arrays and first class procedures. These structures allow many new programs to be easily written to run on the GPU. Harlan is able to maintain reasonable performance on problems that are in the more traditional domain of GPUs.

Harlan's semantics are designed to enable optimization, and future work should explore these optimizations. In this work, we have presented one region inference strategy, but many others are legal. Investigating the tradeoffs in different strategies would be valuable.

Though Harlan supports trees, its parallelism constructs work in terms of vectors. New parallel constructs are likely necessary in order to most efficiently work with tree structures. Nested data parallel languages like NESL [2] could provide inspiration.

Finally, modern computers typically have both a CPU and GPU available for computation. Harlan is able to trivially use either because it uses OpenCL, meaning Harlan could potentially use all available hardware automatically, perhaps preferentially scheduling kernels on different hardware depending on the characteristics of the kernel and the hardware.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 0834722, 1035658 and 1248464, as well as a gift from the Mozilla Corporation. This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

References

[1] Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2012)

[2] Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. Journal of Parallel and Distributed Computing 21(1), 4–14 (Apr 1994)

[3] Bocchino, Jr., R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel programming must be deterministic by default. In: Proceedings of the First USENIX conference on Hot topics in parallelism. USENIX Association (2009)

[4] Catanzaro, B.C., Garland, M., Keutzer, K.: Copperhead: compiling an embedded data parallel language. In: PPOPP. pp. 47–56 (2011)

[5] Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. ACM (2011)

[6] Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming. pp. 3–14. DAMP '11, ACM, New York, NY, USA (2011)

[7] Chakravarty, M., Keller, G., Lee, S., McDonell, T., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming. pp. 3–14. ACM (2011)

[8] Collins, A., Grewe, D., Grover, V., Lee, S., Susnea, A.: NOVA: A functional language for data parallelism. Tech. Rep. NVR-2013-001, NVIDIA (July 2013)

[9] Cooper, K.D., Torczon, L.: Engineering a Compiler. Elsevier Science (October 2003)

[10] Crary, K., Weirich, S., Morrisett, G.: Intensional polymorphism in type-erasure semantics. In: Proceedings of the third ACM SIGPLAN international conference on Functional programming. ACM (1998)

[11] Cunningham, D., Bordawekar, R., Saraswat, V.: Gpu programming in a high level language: Compiling x10 to cuda. In: Proceedings of the 2011 ACM SIGPLAN X10 Workshop. pp. 8:1–8:10. X10 '11, ACM, New York, NY, USA (2011)

[12] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2009)

[13] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in Cyclone. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. ACM (2002)

[14] Holk, E., Byrd, W., Mahajan, N., Willcock, J., Chauhan, A., Lumsdaine, A.: Declarative parallel programming for GPUs. In: Proceedings of the International Conference on Parallel Computing (ParCo) (Sep 2011)

[15] Holk, E., Pathirage, M., Chauhan, A., Lumsdaine, A., Matsakis, N.D.: GPU programming in Rust: Implementing high-level abstractions in a systems-level language. In: Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (May 2013)

[16] Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic cpu-gpu communication management and optimization. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. ACM (2011)

[17] Ji, F., Lin, H., Ma, X.: Rsvm: A region-based software virtual memory for gpu. In: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques. pp. 269–278. PACT '13, IEEE Press, Piscataway, NJ, USA (2013)

[18] Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Functional programming languages and computer architecture. pp. 190–203. Springer (1985)

[19] Khronos OpenCL Working Group: The OpenCL Specification (Nov 2012)

[20] NVIDIA: CUDA C Programming Guide (Oct 2012)

[21] Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., Stich, M.: OptiX: a general purpose ray tracing engine. In: ACM SIGGRAPH 2010 papers. ACM (2010)

[22] Prabhu, T., Ramalingam, S., Might, M., Hall, M.: EigenCFA: accelerating flow analysis with GPUs. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM (2011)

[23] Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference - Volume 2. ACM (1972)

[24] The Rust programming language. `http://www.rust-lang.org/`

[25] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application

performance evaluation of a multithreaded gpu using cuda. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM (2008)

[26] Sarkar, D., Waddell, O., Dybvig, R.K.: A nanopass infrastructure for compiler education. In: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming. ACM (2004)

[27] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)

[28] Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation 132(2) (1997)

[29] Yang, K., He, B., Luo, Q., Sander, P.V., Shi, J.: Stack-based parallel recursion on graphics processors. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM (2009)