REGION-BASED MEMORY MANAGEMENT FOR EXPRESSIVE GPU PROGRAMMING

Eric Holk

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
June 2016

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Arun Chauhan, Ph.D.

Ryan Newton, Ph.D.

Amr Sabry, Ph.D.

November 10, 2015

Copyright 2016
Eric Holk
All rights reserved

To my parents.

This work would not have been possible without their support, encouragement and guidance throughout my life.

Acknowledgements

As anyone who has written a dissertation will attest, it represents a gargantuan amount of work. Like any large project, I did not do this alone. I have been fortunate to have the help of so many people that it will prove impossible to list them all here. I will do my best.

My advisor and the rest of my committee have been invaluable. They are: Andrew Lumsdaine, Arun Chauhan, Ryan Newton and Amr Sabry. At every milestone in this process they offered pointed feedback that has truly made my work better. They have taught my classes and written papers with me. In doing so, they have taught me how to be a researcher and a scientist.

I have been fortunate to go through graduate school with a number of excellent peers and role models. Among these are Roshan James, Michael Adams, Nate Husted, Lindsey Kuper and Alex Rudnick. Several groups at IU deserve special mention. The PL Wonks group was a huge encouragement, giving me the chance to get to know students and faculty, practice talks and generally learn what PL people do. I would also like to acknowledge the Open Systems Lab and its reincarnation as the Center for Research in Extreme Scale Technologies. I want to particularly call out Kelsey Shepherd, Rebecca Lowe, Michael Hansen, Andrew Friedly, Abhishek Kulkarni, Trevor McDonell and Nick Edmonds.

Dan Friedman is an absolutely inspirational educator. He has a knack for raising seemingly simple questions that I have spent a great deal of time trying to answer satisfactorily. Kent Dybvig taught me a better way to think about compilers in particular and software engineering in general. I am grateful as well to Andy Keep for the Nanopass Framework, which regularly reduced the amount of code in various Harlan compiler passes by a factor of ten. I did most of the work adopting Nanopass while working with Andy Keep in Matt

Might's lab at the University of Utah. I'm grateful to Matt for this opportunity and his input into the design of Harlan.

Two students and later post docs were invaluable to me as mentors. These are Will Byrd and Joseph Cottam.

Claire Alvis is an incredibly energetic and productive hacker. Harlan owes much to her early efforts in its implementation.

I am grateful to Jeremy Siek for working with me to develop the semantics of Harlan. His suggestion to describe the semantics in terms of separation logic led to a much nicer solution than I would have achieved on my own.

I was able to explore some of the ideas in this thesis in my blog. I am so grateful for my readers and the comments they gave. Not only did they help me learn to explain these concepts better, they gave me the satisfaction of knowing I was able to help others learn something. Similarly, I want to thank the people who said kind words to me at conferences. It is easy to lose sight of the big picture, and simple words of encouragement from people such as Ron Garcia reminded me that my work is interesting and important.

I wish to acknowledge those who provided funding for this work. Much of the initial support was provided by NSF Grant Nos. 0834722, 1035658, 1111888 and 1248464 and the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.. 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. Support was also provided by a gift from the Mozilla Corporation.

I would like to acknowledge several teachers from before I started graduate school. As an undergrad, Claude Anderson first introduced me to the study of programming languages and inspired an appreciation for the Scheme programming language. In elementary and high school Mike Menefee and David Pevovar both showed so much patience in fostering my love of computing.

I cannot possibly thank my parents, George and Minerva Holk, enough. They have taught me so much, encouraged me in so many ways and have always been there for me.

There are so many other friends and loved ones who have been there for me at every step of the way. Thank you so much for your encouragement. I hope I can build you up just as much as you've helped me.

Lastly and most importantly, I give thanks to my Creator, who not only made me but give me the means and ability to accomplish this Ph.D.

Eric Holk

REGION-BASED MEMORY MANAGEMENT FOR EXPRESSIVE GPU PROGRAMMING

Over the last decade, graphics processing units (GPUs) have seen their use broaden from purely graphical tasks to general purpose computation. The increased programmability required by demanding graphics applications has proven useful for a number of non-graphical problems as well. GPUs high memory bandwidth and floating point performance make them attractive for general computation workloads, yet these benefits come at the cost of added complexity. One particular problem is the fact that GPUs and their associated high performance memory typically lie on discrete cards that are separated from the host CPU by the PCI-Express bus. This requires programmers to carefully manage the transfer of data between the CPU and GPU memory so that the right data is in the right place at the right time. Programmers must design data structures with serialization in mind in order to efficiently move data across the PCI bus. In practice, this leads to programmers working with only simple data structures such as one or two-dimensional arrays and the applications that can be easily expressed in terms of these structures. CPU programmers have long had access to richer data structures, such as trees or first class procedures, which enable new and simpler approaches to solving certain problems.

This thesis explores the use of region-based memory management (RBMM) to overcome these data movement challenges. RBMM is a technique in which data is assigned to regions and these regions can then be operated on as a unit. One of the first uses of regions was to amortize the cost of deallocation. Many small objects would be allocated in a single region and the region could be deallocated as a single operation independent of the number of items in the region. In this thesis, regions are used as the unit of data movement between the CPU and GPU. Data structures are assigned to a region and thus the

runtime system does not have to be aware of the internal layout of a data structure. The runtime system can simply move the entire region from one device to another, keeping the internal layout intact and allowing code running on either device to operate on the data in the same way.

These ideas are explored through a new programming language called Harlan. Harlan is designed to simplify programming GPUs and other data parallel processors. It provides kernel expressions as its fundamental mechanism for parallelism. Kernels function similarly to a parallel map or zipWith operation from other functional programming languages. For example, the expression `(kernel ([x xs] [y ys]) (+ x y))` evaluates to a vector where each element is the sum of the corresponding elements in xs and ys. Kernels can have arbitrary body expressions that can even include kernels, thereby supporting nested data parallelism. Harlan uses a region-based memory system to enable higher level programming features such as trees and algebraic data types (ADTs) and even first class procedures. Like all data in Harlan, first class procedures are device-independent, so a procedure created in GPU code can be applied in CPU code and vice-versa.

Besides providing the design and description of the implementation of Harlan, this thesis includes a type safety proof for a small model of Harlans region system as well as a number of small application case studies. The type safety proof provides formal support that Harlan ensures programs will have the right data in the right place at the right time. The application case studies show that Harlan and the ideas embodied within it are useful both for a number of traditional applications as well as problems that are problematic for previous GPU programming languages. The design and implementation of Harlan, its proof of type safety and the set of application case studies together show that region-based memory management is an effective way of enabling high level features in languages targeting CPU/GPU systems and other machines with disjoint memories.

Andrew Lumsdaine, Ph.D.

Arun Chauhan, Ph.D.

Ryan Newton, Ph.D.

Amr Sabry, Ph.D.

Contents

Abstract i

List of Figures xiii

Chapter 1. Introduction 1

Chapter 2. Background 5

2.1. Parallel Computing Architectures 5

2.2. General Purpose GPU Computing 7

2.3. Region-based Memory Management 11

Chapter 3. Related Work 13

3.1. GPU Applications and Algorithms 13

3.2. Data Parallelism 14

3.3. GPU Programming Languages 15

3.4. Regions 18

3.5. Semantics 20

Chapter 4. Exploring Regions with the Harlan Language 22

4.1. A User's View of Harlan 22

4.2. Region-based Memory Management in Harlan 33

Chapter 5. Harlan Implementation 41

5.1. Compilation 41

5.2. Implementation of the Regions System 51

5.3. Optimizations 52

5.4. In-kernel Error Handling 55

Chapter 6. Region Semantics for Multi-memory Systems 57
6.1. Core Harlan 58
6.2. Operational Semantics 60
6.3. A Separation Logic Primer 69
6.4. Type System 70
6.5. Type Safety 78
6.6. Auxiliary Lemmas 94
6.7. Designing for Proof Mechanization 95

Chapter 7. Harlan Case Studies 97
7.1. Benchmarking Methodology 97
7.2. Dense Matrix Multiplication 98
7.3. Ray Tracing 100
7.4. Graph Algorithms 110
7.5. Integrating with external applications 117
7.6. GPU Performance Characterization 120

Chapter 8. Conclusion 126

Bibliography 128

Curriculum Vita

List of Figures

1.1 CPU-GPU System Architecture. The size of the arrows indicates the relative data transfer rate of the three interfaces. 3

2.1 CUDA Vector Addition Kernel 8

2.2 CUDA Vector Addition Host Code 9

2.3 The Harlan equivalent to the vector addition program in Figure 2.1 and Figure 2.2. 9

2.4 CUDA Processing Architecture 10

2.5 An example arrangement of regions and values contained within them. 12

4.1 The grammar of Harlan's core forms. Harlan programs consist of a module which should define a function called `main`. The `main` function is not necessary for libraries. Only Harlan's primitive forms are listed here; Harlan forms that are not listed here are implemented as macros or library functions. 24

4.2 A selection of Harlan's non-core forms. These are implemented either as macros or functions in Harlan's standard library. Many of these are described in more detail in Section 4.1.2. 25

4.3 Several simple example kernels and their output. 26

4.4 Several examples of vectors in Harlan. 32

4.5 Examples showing how data is assigned to and arranged within regions. 37

4.6 An example of a possible data structure if ADTs could take multiple region parameters. The leftward nodes are in region `r1` and the rightward nodes are in region `r2`. 38

5.1 Harlan Compiler Passes. 43

5.2 The call graph for a λ-Calculus interpreter written in Harlan. The strongly connected components (SCCs) are indicated by squares, while ovals represent functions and arrows indicate that a function may call another. Note that most of these functions are generated internally by the compiler. 50

5.3 An example of false recursion. 51

6.1 The syntax for Core Harlan. This is a small model of the Harlan language which we will use to study its semantics. The e non-terminal represents Harlan expressions. Variables are indicated by x, while **r** indicates a region variable. l represents a location, either CPU or GPU. 59

6.2 The baseline interpreter for the λ-Calculus. Procedures (`lambda`) have been extended with location and region requirement annotations that are part of Core Harlan but not standard λ-Calculus. 60

6.3 The interpreter from Figure 6.2 modified to thread store and location variables throughout the execution. 62

6.4 An example store. 62

6.5 Region manipulation functions. 65

6.6 The full interpreter for Core Harlan. 66

6.7 Syntax of types for Core Harlan. 71

6.8 Machine typing rules. 71

6.9 Typing rules for expressions. 74

6.10 Location compatibility rules. Intuitively, $l <: l'$ means that code that runs in location l can also run in location l'. 75

6.11 Continuation typing rules. 76

6.12 Auxiliary functions. 78

6.13 Typing rules for values. 78

7.1 The core dense matrix multiplication kernel. This assumes we are multiplying two square matrices, `A` and `B`. 98
7.2 Dense matrix multiplication performance. 99
7.3 OpenCL dense matrix multiplication kernel. 99
7.4 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. 101
7.5 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. 103
7.6 An image rendered by ray tracing using a direct style and KD trees. Importantly, they are identical. 104
7.7 Harlan code to build a KD-tree. 106
7.8 A KD-Tree produced by Harlan. 107
7.9 A 2D projection of the scene and tree from Figure 7.6. 108
7.10 Harlan code to traverse a KD tree. 108
7.11 Possible intersections of a ray and two bounding volumes. 109
7.12 An example graph on which to perform a breadth first search. 111
7.13 Several representations of the graph in Figure 7.12. 112
7.14 Harlan's representation of the graph in Figure 7.12. Assume variables are in scope that match vertex names to identifiers. For example, `(let ((A 0) (B 1) ...) ...)`. 112
7.15 An example of how `kernel-update!` might work. 114
7.16 Harlan SCC coloring code. 116
7.17 Harlan SCC component assignment code. 117

7.18 Harlan SCC driver code. 118

7.19 Transfer times between the CPU and GPU memory for buffers of various sizes. 121

7.20 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 8MB or greater. 122

7.21 Roofline model for NVIDIA Tesla K40c GPU. 123

7.22 Vector addition in Harlan compared with vector addition in CUBLAS. 124

7.23 Dot product in Harlan compared with dot product in CUBLAS. 125

CHAPTER 1

Introduction

Thesis Statement

Region-based memory management is an effective way of enabling high level features in languages targeting machines with multiple disjoint memories.

Over the last decade, graphics processing units (GPUs) have become popular for high performance computing due to the high throughput afforded by their massively parallel architecture. Some applications have reported as much as a 10 to 1000× speedup by moving them to the GPU [**53**]. For example, the EigenCFA project reported a 72× speedup over a CPU version of control flow analysis [**67**]. [**29**] reports up to a 40× speedup on a Discrete Fourier Transform (DFT) implementation. [**75**] and [**83**] report impressive speedups as well.[1] However, developing high performance algorithms for the GPU remains challenging. 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 [**13**] and Accelerate [**14**] 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.

[1] All of these claims are somewhat controversial. [**53**] reports that when comparing highly tuned GPU and GPU codes the GPU only outperforms the CPU by about 2.5×.

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 [**67**]. While this algorithm showed an impressive 72$\times$ 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 thesis explores the use of region-based memory management (RBMM) as a means of increasing the expressiveness of programming languages that target GPUs. GPU programming is more difficult than CPU programming for at least two high level reasons: (1) the GPU is a physically separate computation device and (2) current GPUs access memory in a disjoint address space. Existing GPU programming languages, ranging from the low-level CUDA and OpenCL to high-level languages like Copperhead [**13**] and Accelerate [**14**], primarily address the first issue by simplifying the process of writing code for execution on the GPU. Other languages, such as NOVA [**16**], allow richer data structures like algebraic data types (ADTs) and first class procedures, but do not allow these to move between the CPU and GPU. They improve the expressiveness of the language but entirely ignore problem the data movement issues.

This data movement problem stems from the fact that a CPU-GPU system is an example of a distributed system. Figure 1.1 illustrates the architecture of a CPU-GPU system. The two devices run independently of each other, and more importantly, they have entirely different address spaces. This means that transferring intricate pointer structures requires traversing and serializing the whole structure and then recreating it on the target device. Furthermore, pointers in one memory location are meaningless when viewed in the context of another location, and thus to move a pointer structure, any internal pointers must be rewritten with pointers that are meaningful on the new location.

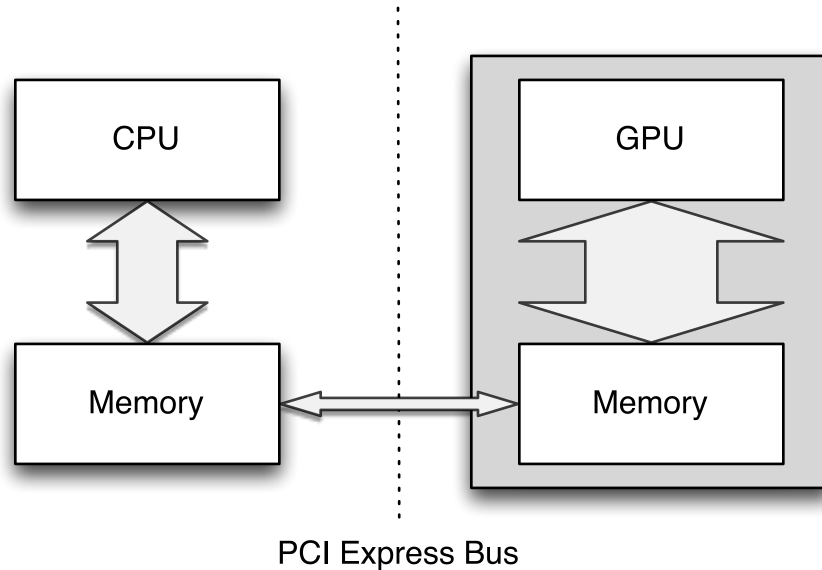


FIGURE 1.1. CPU-GPU System Architecture. The size of the arrows indicates the relative data transfer rate of the three interfaces.

These problems can be solved using RBMM, which is a way of grouping objects that are in some way related into aggregate units called regions. This technique has been used for garbage collection [**80**] and for ensuring memory reference safety [**31**]. In our case, a region inference algorithm can statically determine a set of regions that contains all the data needed by a section of code. These regions can then moved as single units between memories. Within a region, pointers are represented as offsets from the base of a region, which means their relative locations are unchanged when a region migrates to a different physical location. Thus, there is no need to rewrite internal pointers or traverse the entire structure when data is needed by a different computational device.

While this work considers regions in the context of CPU-GPU systems, the basic approach should translate to other distributed systems, including compute clusters.

The vehicle for exploring RBMM applied to GPU programming is a new programming language called Harlan. Harlan [**38**,**39**] enables richer control flow constructs by providing first class functions (`lambda`). Procedures created in Harlan can move freely between the GPU and central processing unit (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))))
```

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 [**80**]). Data can then be moved in units of regions, which enables efficient data transfer between the CPU and GPU memory.

The rest of this thesis proceeds as follows. Chapter 2 provides some background information on general purpose GPU (GPGPU) programming. Chapter 3 surveys other work in this area and sets Harlan in its proper context. Chapter 4 presents plans for developing and evaluating Harlan as a GPGPU language with region-based memory. Chapter 5 goes into more detail about the implementation of the Harlan language and it's region-based memory system. Chapter 6 gives a formal semantics for a subset of Harlan and presents a proof of type safety. Chapter 7 provides a more in depth look at several Harlan applications to show its usefulness on a variety of problems that are challenging for previous GPU programming approaches. Finally, Chapter 8 summarizes this work and makes several concluding remarks.

CHAPTER 2

Background

Many problems in computing are simply too big for a single machine. While we have often been able to rely on clock speed increases to improve performance, in recent years there has been a marked shift towards increasing parallelism instead. Unfortunately, realizing the benefits of parallelism nearly always requires changes to the program and requires the programmer to be aware of an increasing number of architectural details. In this chapter, we will survey several parallel programming architectures and paradigms in order to better understand the challenges and solutions in general purpose GPU computing.

2.1. Parallel Computing Architectures

Parallelism can be found at every level of computer architecture. Within a single CPU core, we find instruction-level parallelism (ILP) in which the processor will attempt to simultaneously issue multiple instructions provided the data dependencies allow it. Many CPUs include a number of single instruction, multiple data (SIMD) instructions that provide small scale vector processing. Vector processors extend this concept even further. Machines may have more than one CPU and CPUs now nearly always have multiple processor cores. Both of these configurations are considered symmetric multiprocessors (SMPs). To scale up even further, multiple machines can be connected into a cluster that behaves like a single machine. Message passing models such as MPI simplify programming clusters by providing a single program, multiple data (SPMD) programming model.

Lately there has been a trend towards *hybrid* computers, which combine a traditional CPU with one or more *accelerators*. These accelerators are a separate processor that are optimized for special purpose computation. GPUs fall under the category of accelerators, as do field programmable gate arrays (FPGAs) and Intel's Xeon PHI.

2.1.1. Symmetric Multiprocessors. While once purely in the realm of high end workstations, SMPs are now commonplace. Nearly all recent laptop and desktop computers including a multicore processor. Many smartphones now include four or more processor cores. Symmetric multiprocessors are machines with multiple identical processors, typically even running at the same clock speed.[1] Generally, SMPs are shared memory machines, meaning each core can access any of the memory in the system. Sometimes memory is configured such that certain portions of memory can be accessed more efficiently by certain processors. These are called non-uniform memory access (NUMA) machines.

On aspect of SMPs is their cache coherency protocol. Because each processor may access any of the memory, care is needed to ensure consistency and correctness of modifications to shared memory locations. Processors enable synchronization through atomic operations, such as compare and swap (CAS). These operations perform a memory read, modification and write as a single unit, such that no thread running on another processor can interfere. Atomic operations like CAS are sufficient to implement a variety of synchronization primitives, such as mutexes.

2.1.2. Accelerators. Recent years have seen a resurgence in interest in compute accelerators. These are special purpose processors that are installed inside a general purpose computer. One of the most common accelerators is the graphics processing unit, which was originally designed to accelerate 3D rendering tasks. A number of other accelerators are available, such as FPGAs, the Intel Xeon PHI, or Google's Tensor Processing Unit. Each of these is optimized for certain types of computation, such as vector processing or machine learning. Due to their specificity, programs must be specially written to take advantage of an accelerator. Often, these programs must target a specific accelerator from a particular vendor, though application programming interfaces (APIs) such as OpenGL and OpenCL aim to alleviate this somewhat.

[1] Some mobile processors are able to independently adjust the clock speed of each core in order to maximize power efficiency.

2.1.3. Clusters. The next step up in the computer hierarchy is to build a cluster of machines. One example is the Beowulf cluster [**77**] and most high performance computers today use a cluster architecture. Programs are typically written in a SPMD style, where each node runs the same program and they divide the computation by passing messages.

Although Harlan as currently implemented targets only single node systems, its region system could probably be extended to enable safely passing rich data structures between nodes in a cluster. Indeed, similar techniques have already been used in the context of Haskell [**86**].

2.2. 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 now see increased use in scientific computing and other disciplines.

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

The most popular framework for programming NVIDIA GPUs is CUDA [**59**], 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 [**50**]. High performance GPU kernels are aware of the divisions between blocks and warps.

A simple CUDA example program is shown in Figures 2.1 and 2.2. The kernel code in Figure 2.1 adds two vectors, `a` and `b`, and stores the result in `c`. The kernel executes with `N` threads, so each thread is responsible for adding one element of the input vectors. CUDA kernels run as a grid made up of blocks of threads. This grid is often bigger than the input

```
__global__
void add_vectors(int N,
                 const double *a,
                 const double *b,
                 double *c) {
    int i = blockIdx.x;
    if(i < N)
        c[i] = a[i] + b[i];
}
```

FIGURE 2.1. CUDA Vector Addition Kernel

data, so there is a check to be sure the thread index is within bounds. One reason the grid can be larger than the data size is if the block size does not evenly divide the data size.

Executing the kernel in Figure 2.1 requires a fair amount of addition code, which is shown in fig:cuda-vector-addition-host-code. This involves allocating GPU memory and copying the data from the host memory to the device memory. The kernel is actually called in the `add_vectors<<<N, 1>>>(...)` line, which specifies the kernel should launch `N` blocks with a block size of `1`. Finally, when the kernel is completed, another call to `cudaMemcpy` copies the results from the GPU to the host memory.

By contrast, an analogous Harlan program is shown in Figure 2.3. This is significantly shorter, in part because the Harlan language takes responsibility for all data movement.

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 but must be managed explicitly by the programmer. 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.

Figure 2.4 summarizes the logical architecture of a CUDA-capable device. The processor consists of some number of SMs, each containing some number of threads (represented as wavy lines) grouped into warps (represented as boxes around threads) all sharing a

```
int main() {
    double a[N], b[N], c[N];

    double *dev_a, *dev_b, *dev_c;

    fill_array(a, N);
    fill_array(b, N);

    cudaMalloc(&dev_a, N * sizeof(double));
    cudaMalloc(&dev_b, N * sizeof(double));
    cudaMalloc(&dev_c, N * sizeof(double));

    cudaMemcpy(a, dev_a, N * sizeof(double),
               cudaMemcpyHostToDevice);
    cudaMemcpy(b, dev_b, N * sizeof(double),
               cudaMemcpyHostToDevice);

    add_vectors<<<N, 1>>>(N, dev_a, dev_b, dev_c);

    cudaMemcpy(c, dev_c, N * sizeof(double),
               cudaMemcpyDeviceToHost);

    output_results(c, N);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

FIGURE 2.2. CUDA Vector Addition Host Code

```
(module
  (define (main)
    (let* ((N 10000)
           (a (load-vector N))
           (b (load-vector N))
           (c (kernel ((a_i a) (b_i b))
                (+ a_i b_i))))
      (output-results c))
```

FIGURE 2.3. The Harlan equivalent to the vector addition program in Figure 2.1 and Figure 2.2.

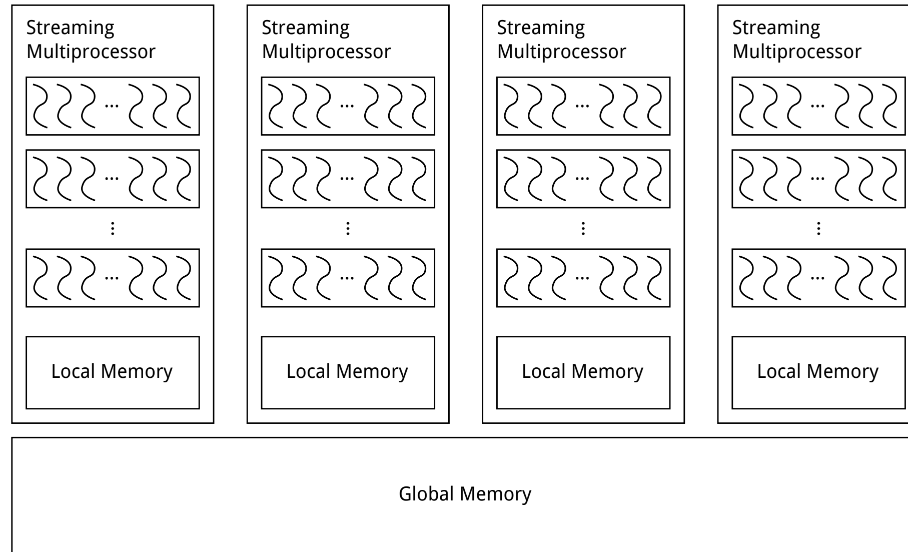


FIGURE 2.4. CUDA Processing Architecture

small local memory. All of the SMs share a single global memory. In terms of CUDA programs, each SM executes one block at a time.

In the current generation of GPUs, 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 is 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 architectures of CPUs and GPUs are becoming more similar. Newer generations of CPUs include more powerful vector processing capabilities, such as the AVX-512 instruction set in Intel's Skylake CPUs. Likewise, GPUs have adopted features that were once CPU-only, such as the hardware-managed L2 caches that appeared in NVIDIA's Fermi-class GPUs. 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. In other words, the programmer should be able to use the same language for both CPU and GPU code, with no restrictions on either. 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.

2.3. 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 RBMM as a general garbage collection strategy [**80**]. In this system, objects were automatically assigned to regions based on their inferred lifetimes. An important safety property is that a region must outlive all references to any object contained within it.

Below is an example of what a program in a Scheme-like language with explicit regions might look like.

```
(let-region r1
  (let ((f (lambda (z)
             (let-region r2
               (let ((x (ref r1 5))
                     (y (ref r2 6)))
                 (let ((p (cons y x)))
                   (cdr p)))))))
    (deref (f f))))
```

This example creates a region `r1` and creates a function `f` which creates its own local region then allocates a number into region `r1` and another into region `r2`. Finally, `f` returns the reference in `r1` after shuffling it through a pair. Upon return, the program dereferences the reference and the whole program evaluates to the number 5.

Figure 2.5 shows what the memory layout would be before returning from `f`. The box labeled `r1` is the region created outside of the function, while the box labeled `r2` is the region that is local to function `f`. The pair `p` is shown with pointers into each region. Regions follow a stack discipline, so upon exiting `f`, region `r2` would be destroyed. The allocation by `f` into region `r1` can be thought of allocating from the caller's stack frame.

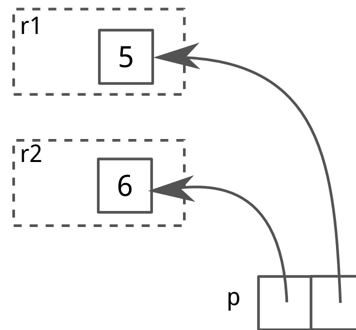


FIGURE 2.5. An example arrangement of regions and values contained within them.

The region system can verify that this program is safe, even without a tracing garbage collector, by ensuring that no references to a region outlive the region.

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 projects such as the Apache Web Server.

Another use of regions might more accurately be called *lifetimes*. This version of regions appears in programming languages like Cyclone [**31**] and Rust [**58**] to enforce safety properties by ensuring that an object outlives any references to the object. Thus, 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.

CHAPTER 3

Related Work

Harlan and its region system extend a large body of related work. In this chapter, we survey work from several areas and discuss how it relates to Harlan.

3.1. GPU Applications and Algorithms

Some of the initial inspiration for this project came from EigenCFA [**67**]. EigenCFA is an algorithm for control flow analysis (CFA) on the GPU that achieves up to a 72× speedup over a CPU implementation. This project showed that GPUs are applicable to more than just traditional scientific computing problems, but also highlighted the limitations on the expressiveness in current GPU languages. In order to achieve this performance, the authors had to recast CFA in terms of linear algebra. Rather than using more traditional rules for evaluating λ-Calculus expressions, the authors developed a way of encoding programs and runtime environments into matrices and using matrix multiplication to simulate their evaluation. While this is no doubt clever, ideally a better programming language would allow the programmer to express CFA in a more direct style.

Some of the earliest GPGPU instances were programmed as graphics applications that solved a non-graphics problem. One early matrix multiplication algorithm "computed by literally visualizing the computations of a simple parallel processing algorithm" [**52**]. Effectively, computation occurred by rendering an image and interpreting the results as the solution to a matrix multiplication problem.

GPUs are now regularly used to solve a variety of problems. These include graph algorithms such as breadth first search (BFS) and single source shortest paths (SSSP) [**41**, **57**], various rendering algorithms such as ray tracing, KD trees and ray marching [**44**]. GPUs have even be used to accelerate cryptocurrency mining [**21**, **60**].

3.2. Data Parallelism

GPUs are designed for data parallel computation—that is, applying the same relatively simple computation to a large amount of data. This is evident in the fact that GPUs have wide SIMD lanes and high memory bandwidth. As such, much of the prior work on data parallel programming languages is relevant to GPU programming as well. The NESL programming language is one of the first practical implementations of a data parallel programming language [**9**]. This language built on earlier work by Guy Blelloch on the scan-vector model [**8**].

NESL was partially inspired by the Connection Machine Lisp [**76**]. Connection Machine Lisp was built around a new data structure called a *xapping*. Xappings are distributed associative arrays. Each processor is responsible for some subset of the keys in the array, and operations on xappings are automatically applied to values with matching keys. Harlan draws inspiration from Connection Machine Lisp as well, in that both languages are designed to solve problems beyond pure numerical computation.

More recently, NESL has been successfully ported to the GPU [**6**]. This work used the existing NESL compiler and provided a new implementation of the VCODE interpreter that executes vector operations on the GPU. The new VCODE interpreter is also responsible for performing kernel fusion, which is critical for good performance on the GPU. Particular care is given to the representation of vector segment descriptors, as different representations have different performance characteristics. Similarly, Harlan is concerned with developing efficient representations of interesting data structures.

Data parallelism in the presence of trees often suffers from unbalanced computation, that is, some processors receive too much work to do while others receive too little. Lazy tree splitting is a technique that addresses this by dynamically partitioning workloads based on when compute resources are available [**5**]. This is in contrast to Eager Binary Splitting, which is used by Cilk [**27**] and Intel's Threading Building Blocks [**45**]. Eager binary splitting divides the work up into chunks of fixed sizes ahead of time. This leads to

the Goldilocks problem, that is, balancing the chunk size between having so much parallelism that scheduling overhead destroys the benefits of parallelism and having such large blocks that not all compute devices are busy. Lazy tree splitting divides the remaining work when there is reason to believe that there are idle processors, and otherwise keeps processing the current chunk sequentially. Key to this technique is designing data structures such that the system can efficiently partition the remaining work in half. The authors accomplish this by using ropes and zippers.

Several themes in this work relate to Harlan. One is selecting underlying data structures to enable efficient implementation. This is related to Harlan's use of regions to manage the runtime representation of data. The lazy tree splitting paper is also performed with performance portability, which is currently a weakness of Harlan. Harlan has many parameters that need tuned, which depend on the characteristics both of the program and the hardware. Harlan's use of OpenCL makes it a natural vehicle to explore performance portability, as it can trivially use the CPU, GPU and other devices like the Intel's Xeon PHI.

3.3. GPU Programming Languages

In the early 2000s, GPU had become programmable enough that practitioners began applying them to more areas than just graphics. A variety of techniques developed to convert scientific computation problems into graphics problems suitable for execution on a GPU. Many of these techniques are surveyed in [**62**]. These techniques showed the promise of general purpose computation on graphics hardware and soon general purpose computation was supported directly in both graphics hardware and new programming languages.

One early programming language for GPGPU programming is Brook [**12**]. Brook is an extension to C that adds the concepts of streams, kernels and reductions. Streams are analogous to Harlan's vectors, kernels are functions that can be mapped over streams, and reductions combine multiple values. Prior to Brook, GPGPU computation required to programmer to be extremely familiar with the latest graphics APIs and hardware. Brook

still compiled to graphics APIs, since more general purpose APIs such as CUDA had not yet been released.

The advent of CUDA allowed programmers to work with data without massaging it into an image. Instead, programmers would write in CUDA-C, a dialect of C that includes support for executing kernels on an NVIDIA GPU. CUDA's programming environment was still very low level. One effort at alleviating this is Copperhead [**13**]. Copperhead is a domain specific language (DSL) embedded in Python for GPU computing. Functions can be compiled for the GPU by applying an `@cu` decorator. Data parallelism is accomplished using operators like `map`. Copperhead allows `lambda` for defining mapping functions, but Copperhead's `lambda` is not as general as Harlan's. Copperhead's concept of places allows programs to manage the location of data and code using Python's `with` statement. Copperhead also supports arbitrarily nested data structures, using the same techniques used in the scan-vector model of parallelism [**8**].

Accelerate is a similar language for GPU computing embedded in Haskell [**14**]. Accelerate uses the operator overloading technique to embed itself in Haskell and makes heavy use of Haskell's type system to statically eliminate errors such as indexing into an array with an incorrect shape. By contrast, Copperhead detects many of these same issues through runtime program analysis. Programs are compiled into CUDA code using skeletons. Skeletons represent the outline of a data parallel operator that can be filled in with the specific computation. Later work on Accelerate has enabled optimizations like sharing recovery and array fusion to reduce code duplication and eliminate intermediate results [**55**].

One key challenge to efficient GPU programming is managing the communication between the host and device memories. [**46**] addresses this problem with a fully automatic system. It is implemented as several compiler passes and a runtime library. The compiler detects points at which communication must happen and inserts the appropriate code to do so. This includes several optimizations, including map promotion, alloca promotion and glue kernels. Alloca promotion is similar to Harlan's let lifting (Section 5.3.2) while glue kernels are similar to remote vector allocation in Harlan (Section 5.3.4) The

goal of these optimizations is to transform a program from consisting of mostly *cyclic* communication–that is, copying all data to the GPU, launching a kernel and then copying all the data back from the GPU–into a program consisting of mostly *acyclic* communication, where the data is copied to the GPU at the start, then several kernels execute and finally the relevant results are copied back. Harlan attempts to do something similar dynamically through lazy data transfer (Section 5.3.3) Their system copies data at the granularity of allocation units, thereby allowing double indirection and arbitrary pointer arithmetic. These allocation units are quite similar to Harlan's regions, although Harlan's region system supports arbitrary levels of pointer indirection due to its static type system.

Although CUDA now supports the use of recursive functions in kernels, earlier versions did not and OpenCL still does not. There are well-known techniques, however, for overcoming these limitations. OptiX, a DSL for ray tracing on the GPU, allows shaders to use recursion to follow rays reflecting off surfaces [**63**]. This is done by applying a continuation passing style (CPS) and trampolining technique to remove the recursion. On the other hand, [**87**] describes a technique for implementing recursion by explicitly manage stacks. Harlan implements recursion using a similar technique.

The latest release of CUDA includes support for a number of C++ 11 features, including lambda functions [**36**]. While CUDA's lambda functions open up new programming styles, they are not as powerful as Harlan's lambdas. Harlan lambdas are independent of their device, while CUDA lambdas cannot be applied on both the host and device.

Work on compiling Mozilla's Rust for the GPU is similar to this in its focus on enabling higher-level abstractions in GPU kernels [**40**]. Rust for the GPU supports some `enum` types (Rust's version of ADTs), but cannot handle pointer structures like those possible in Harlan.

NOVA [**16**] 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.

3.4. Regions

Regions have been used in programming languages for a variety of purposes, including garbage collection, reasoning about the safety of pointers, and reasoning about the safety of sharing data between parallel processes.

The ML Kit compiler used RBMM as a general garbage collection strategy [**80**]. In this system, objects were automatically assigned to regions based on their inferred lifetimes. An important safety property is that a region must outlive all references to any object contained within it.

Several programming languages like Cyclone [**31**] and Rust [**58**] use a version of regions to enforce safety properties by ensuring that an object outlives any references to the object. These languages analyze the lifetime of various data statically and provided the lifetimes are proven safe, they do not have any cost at runtime.

Regions are also used for safe parallelism. Legion is one project that uses a logical region system to ensure safety in parallel programs while providing control of data movement through the memory hierarchy to the programmer [**2**]. Function types are annotated with region parameters and permissions for each region (for example, read, write and reduce). Regions can be subdivided using a coloring, and this subdivision of regions forms the basis of parallelism. Noninterference of parallel processes is checked at runtime at the granularity of regions, which is significantly less overhead than per-access checking as in software transactional memory (STM), but also significantly more expressive than a purely static system. Because of noninterference, regions also indicate portions of a computation that can be dynamically scheduled to run in parallel on different resource. The Legion system has been tested on the Keeneland machine, a large CPU/GPU cluster, and appears to do a good job of automatically utilizing available compute devices. However, the type system appears to impose a heavy annotation burden on the programmer which may limit is applicability in practice.

Harlan's region system differs significantly from Legion's 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 [**11**]. 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 [**47**]. 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 [**46**]. These allocation units are similar to Harlan's regions. Many of the communication optimizations in this work would apply to Harlan. For example, Harlan already uses let lifting (Section 5.3.2) and remote vector allocation (Section 5.3.4), which are similar to *alloca* promotion and glue kernels from CGCM.

Regions in Harlan bear some similarity to places in X10, which can also be compiled for the GPU [**20**]. 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.

Although not explicitly a form of RBMM, Compact Normal Form (CNF) is similar in spirit to Harlan's use of regions [**86**]. CNF is a modification to Haskell that allows data to be completely evaluated into a contiguous region of memory in order to easily transfer data structures between processes. This work is in the context of a Haskell program distributed over a cluster, but is similar to Harlan's need to produce compact data structures to transfer between CPU and GPU memory. One difference is that the programmer is explicitly in control of CNF, while Harlan handles the assignment of data structures to regions automatically and transparently.

3.5. Semantics

There has been relatively little work on formal semantics for GPUs. One example is [**37**], which models the behavior of CUDA-C programs using the $\mathbb{K}$ semantic framework [**70**], building on an existing semantics for C [**22**]. This system is closely related to what is needed for Harlan because it models the whole program, including the fact that memory references may be accessible by either the CPU or the GPU. The language of CUDA-C is in some ways more expressive than Harlan, due to it exposing mutable memory to the programmer. Still, CUDA-C lacks Harlan's advanced features such as first class procedures.

One earlier work on the semantics of GPU languages focuses on modeling the Parallel Thread Execution (PTX) virtual assembly language used by NVIDIA GPUs [**32**]. Unfortunately, this work does not model the interaction of the GPU code with the host CPU program. This interaction is the focus of the Harlan semantics work (Chapter 6).

Work on optimizing NESL includes a full operational semantics that includes cost measures [**10**]. This allowed the implementation of NESL to have provable space and time bounds.

Other semantics for languages with parallelism are typically based on futures or fork-join parallelism. The future construct allows programmers start a computation at one point and time and request its value later [**24**]. This allows the language to execute the future in parallel with the main execution. While this approach could scale to the large number of threads in GPU parallelism, this seems unwieldy.

The semantics of region-based memory management has been explored in several works. Tofte and Talpin [**80**] is on of the most comprehensive treatments. This work describes a source language and its semantics without region annotations and then provides a translation from source terms to a region-annotated target language. The proof of soundness proceeds by a relation between source and target language terms and shows that evaluation of terms in the two languages preserves the relation.

Other treatments of regions have shown that they can implemented as a monad and therefore be encoded in System F [**26**], [**51**]. Among other things, implementing regions as monads enables their use in languages such as Haskell for more precise control over resources like file handles.

The semantics and type system we will develop in Chapter 6 relies on a version of separation logic [**69**] in order to reason about how the locations of regions evolve during program execution. Harlan is mostly side-effect free, but one notable exception is in the movement of regions. Separation logic is a powerful tool to reason about changes to the heap.

CHAPTER 4

Exploring Regions with the Harlan Language

The Harlan programming language combines features from functional programming languages like Scheme and ML and targets data-parallel compute devices like GPUs. The initial observation that led to the development of Harlan was that while the programming languages we have now do a decent job of specifying computational code, they leave much to be desired when it comes to managing the movement of data between disjoint device memories. From the beginning, Harlan has managed the movement of data for the programmer. The language includes a rich set of features, including support for trees, first class procedures and recursive kernels. The language is statically typed and uses a region system to manage the movement of pointer-based data structures between the CPU and GPU memory. Though the language is statically typed, it makes heavy use of type inference and thus explicit type annotations are rare. Due to its S-Expression syntax, programming in Harlan feels similar to programming in Scheme, though there are some significant semantic differences. The Harlan compiler compiles programs to C++ and OpenCL, which allows Harlan to target both GPUs and CPUs, as well as other OpenCL-capable devices.

In this chapter, we see an overview of the language. We start by introducing the language's forms to show what tools are available to Harlan programmers (Section 4.1) and then we explore the Harlan language in more detail. In particular, we will see how Harlan's region system works through a series of examples (Section 4.2).

4.1. A User's View of Harlan

In this section we will see how to write and understand programs in Harlan. We will start with Harlan's general purpose programming features (Section 4.1.1) and then focus on parallel programming in Harlan (Section 4.1.2).

4.1.1. Basics. Harlan can be seen as a general purpose functional programming language extended with data parallelism constructs. Harlan's syntax resembles Scheme, and in general programming in Harlan is meant to feel like programming in Scheme. There are key differences, most important of which is that Harlan has a static type system that is much more in the style of ML, although without type polymorphism.

Following in the Scheme tradition, Harlan has a relatively small number of primitive forms that can be easily extended with macros. The grammar for Harlan's core forms is shown in Figure 4.1. Some of Harlan's non-primitive forms are shown in Figure 4.2. The *Ident* nonterminal represents an identifier such as a variable name and generally follows Scheme's rules for identifiers. The *Literal* nonterminal represents literal values such as integers, floating point numbers, booleans, strings or characters. *BinOp*s include arithmetic and relational operators, such as +, −, < or =.

Programs in Harlan consist of a *Module* including some number of declarations, or *Decl*s. For standalone programs, as opposed to libraries, the module must define a function called `main`. Such a declaration would look as follows:

```
(define (main)
  ...)
```

The `main` function takes no arguments and returns an integer. The returned integer becomes the program's exit status, following C conventions. Successful programs should return 0.

The remaining declaration types are used to `import` libraries, make an `extern`al function available to a Harlan program (Section 7.5), define macros (Section 4.1.3) and define algebraic data types (Section 4.1.6).

Functions consist of one or more expressions, which include standard operations such as displaying information to the user (`print` and `println`), conditional, variable binding, procedure creation and application, arithmetic, et cetera.

Forms such as `vector`, `iota` and `kernel` are used to create and manipulate collections of data in parallel. We will see these in more depth in Section 4.1.2.

```
      Module  ::=  (module Decl ...)
        Decl  ::=  (import Ident)
               |   (define (Ident Ident ...) Expr ... Expr)
               |   (extern Ident Type)
               |   (define-macro Ident (Ident ...) MacroRule ...)
               |   (define-datatype Ident Constructor ...)
        Expr  ::=  (assert Expr)
               |   (print Expr)
               |   (println Expr)
               |   (if Expr Expr)
               |   (if Expr Expr Expr)
               |   (begin Expr ... Expr)
               |   (let-region Ident Expr)
               |   (let ((Ident Expr) ...) Expr ... Expr)
               |   Literal
               |   Ident
               |   (vector Expr ...)
               |   (vector-r Ident Expr ...)
               |   (iota Expr)
               |   (iota-r Ident Expr)
               |   (vector-ref Expr Expr)
               |   (length Expr)
               |   (lambda (Ident ...) Expr ... Expr)
               |   (kernel ((Ident Expr) ...) Expr ... Expr)
               |   (kernel-r Ident ((Ident Expr) ...) Expr ... Expr)
               |   (match Expr (MatchPattern Expr ... Expr) ...)
               |   (error! String)
               |   (BinOp Expr Expr)
               |   (Expr Expr ...)
        Type  ::=  Ident
               |   (ptr Type)
               |   (vec Type)
               |   (closure (Type ...) -> Type)
               |   ((Type ...) -> Type)
   MacroRule  ::=  (SExpr SExpr)
 Constructor  ::=  (Ident Type ...)
MatchPattern  ::=  (Ident Ident ...)
```

FIGURE 4.1. The grammar of Harlan's core forms. Harlan programs consist of a module which should define a function called `main`. The `main` function is not necessary for libraries. Only Harlan's primitive forms are listed here; Harlan forms that are not listed here are implemented as macros or library functions.

```
Expr  +=  (reduce Expr Expr)
       |  (kernel* ((Ident Expr)...)
       |  (map2d Expr Expr)
```

FIGURE 4.2. A selection of Harlan's non-core forms. These are implemented either as macros or functions in Harlan's standard library. Many of these are described in more detail in Section 4.1.2.

There are additional forms with the `-r` suffix that are used along with `let-region` to gain some explicit control over regions. These are not used in general but can be used to experiment with region assignment strategies before they are implemented in the compiler.

In general, regions and types are inferred and not something the programmer must be overtly concerned with. Types, especially, become much more explicit when declaring new data types and when interacting with foreign code.

4.1.2. Data Parallel Programming. Programmers indicate data parallel portions of code with the `kernel` form. Kernels generally run on the GPU, but the language design reserves the right to schedule a kernel on whatever device is the best fit for the kernel's computational requirements. The example below shows a simple vector addition kernel written in Harlan.

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

In this example, `xs` and `ys` are both vectors of numeric values. The two vectors must have the same length to pass them to this kernel. The kernel then launches one thread for each of the elements in `xs` and `ys`, binding the variable `x` to an element in `xs` and `y` to and element in `ys`. Each thread then performs the computation `(+ x y)`. The result of the kernel is another vector containing the element-wise sum of `xs` and `ys`.

Kernels can also be nested. The example below shows how to compute the outer product of two vectors.

```
(kernel ((x xs))
  (kernel ((y ys))
```

```
(let ((xs (vector 1 2 3 4))
      (ys (vector 5 6 7 8)))
  (kernel ((x xs)
           (y ys))
    (+ x y)))

=> (vector 6 8 10 12)
```

(A) Element-wise sum

```
(let ((xs (vector 1 2 3 4))
      (ys (vector 5 6 7 8)))
  (kernel ((x xs)
           (y ys))
    (* x y)))

=> (vector 5 12 21 32)
```

(B) Element-wise product

```
(let ((xs (vector 1 2 3 4))
      (ys (vector 5 6 7 8)))
  (kernel* ((x xs)
            (y ys))
    (* x y)))

=> (vector
     (vector 5  6   7  8)
     (vector 10 12 14 16)
     (vector 15 18 21 24)
     (vector 20 24 28 32))
```

(C) Outer product

FIGURE 4.3. Several simple example kernels and their output.

```
    (* x y)))
```

The result is a two dimensional array, which is represented in Harlan as a vector of vectors.

Harlan's `kernel` form effectively performs a parallel map (Scheme terminology) or parallel zipWith (Haskell terminology). That is, all input vectors must match in size and the result will have the same length. The body of the kernel is applied to corresponding elements from each input array. Harlan provides a variant of `kernel` called `kernel*` which performs outer products. This `kernel*` form is an equivalent shorthand for nested kernels. For example, the kernel in Figure 4.3c could be written as follows:

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

Figure 4.3 shows each of these kernels and the result of applying them to sample inputs.

In each of these examples there is no explicit movement of data. Harlan is responsible for mapping the compute kernels onto the GPU or other compute devices, and for making sure the requisite data is in the correct place at the correct time.

Harlan uses region-based memory management to automatically move data. The type system and compiler are responsible for ensuring that related data structures are located in a single contiguous region of memory. Thus, rather than explicitly serializing the data structure before transferring it, the entire region can be transferred as a single operation. Transferring entire regions at once enables Harlan to ensure the entire data structure is accessible when it is needed. Because there is no way of referring to only a portion of a region, each region and therefore and data the region contains must fit in GPU memory. There is currently no way to transfer part of a region. Relaxing this restriction is an interesting avenue for future work. Note that this constraint only applies to the regions accessed by a given kernel. The program's entire working set may exceed the GPU memory size provided portions that are needed at once fit in GPU memory.

The kernel form is Harlan's only primitive for describing parallelism, but a number of additional parallel operators are provided for convenience. Harlan's core kernel form maps well onto what is natively supported in GPU hardware, and thus more accurately exposes the cost of an operation to the programmer. Implementing additional operators in terms of forms natively supported by the GPU make the operators amenable to compiler optimizations, leading to the potential for higher performance.

One of these additional operators is `kernel*`, which we have already seen. Harlan's library also defines `reduce`, which is used to combine many values into one. The example below shows a how to add up the numbers 1 through 5.

```
(reduce + (vector 1 2 3 4 5))
```

The reduction operator (+ in the example above) can be any associative and commutative function of two arguments. While the compiler cannot statically enforce these properties, the compiler relies on them to ensure determinism when parallelizing the reduction.

An additional non-primitive parallel operator is `map2d`, which is shown below.

```
(map2d (lambda (x) (+ 1 x)) xs)
```

In this example, `xs` must be a two dimensional array. The `map2d` operator applies the supplied function to each element in the array. It is a shorthand for nested kernels.

These parallel operators are implemented as macros that expand into kernels. Harlan's macro system is discussed in more detail in the next section. This style of implementation simplifies the compiler requiring it to process a smaller number of forms, and also allows optimizations to apply more generally.

4.1.3. Macros. Harlan follows Scheme's philosophy of defining a small core language and then enabling more advanced syntactic forms with a powerful macro system. This reduces the number of forms the compiler must handle, and also allows the language to develop more rapidly. The resulting code is still efficient due to generally-applicable compiler optimizations.

Harlan's macro system is based on hygienic pattern-based rewrite rules, similar to Scheme's `syntax-rules` system [**1**]. In Harlan, macros are introduced using the `define-macro` form. An example of a simple `or` macro is given below.

```
(define-macro or ()
  [(_ a b)
   (if a a b)])
```

Macro definitions include a name of the macro (`or` in this case), a set of optional keywords (this example uses no keywords), and then a list of input and output patterns. This `or` macro has only one input and output pair. The input pattern, `(_ a b)`, matches expressions like `(or #t #f)`. The output pattern states that this input should be replaced by `(if #t #t #f)`.

Macros can include multiple patterns. For example, we could extend the `or` macro to accept any number of arguments:

```
(define-macro or ()
  [(_)
   #f]
```

```
  [(_ a)
   a]
  [(_ a b)
   (if a a b)]
  [(_ a b ...)
   (if a a (or b ...))])
```

This macro has patterns that match zero, one, two and one or more arguments. The last pattern shows an example of the ellipsis operator. On the input side, the ellipsis modifies the previous pattern to match zero or more times. On the output side, it repeats the output pattern as many times as the bound pattern variable matched. So, a macro application such as `(or #f #f #t #f)` would expand into `(if #f #f (or #f #t #f))`. Notice that macros expand in an outside-in fashion until all uses of a macro are eliminated.

The following example shows a `cond` macro that makes use of the keyword facility.

```
(define-macro cond (else)
  ((_ (else body ...))
   (begin body ...))
  ((_ (test body ...))
   (if test (begin body ...)))
  ((_ (test body ...) rest ...)
   (if test (begin body ...) (cond rest ...))))
```

This macro is useful for a sequence of conditionals. It examples into a series of nested `if` expressions. The `else` keyword is used to indicate what to do in case no other patterns match. Declaring `else` as a keyword, by listing it after the `cond` macro's name, indicates that the `else` symbol should match literally, rather than binding it as a pattern variable.

An important feature of Harlan's macro system is that it is *hygienic*. Macros often introduce variable bindings, and without proper hygiene these variable bindings could cause problems. Hygiene essentially means that bindings from outside of a macro invocation remain hidden from the macro, and variables bound by a macro expansion remain invisible

to the surrounding code. There are essentially two categories of hygiene errors. The first is the most common, where a macro may introduce a binding that had already been used. Consider the following variation of the `or` macro:

```
(define-macro or ()
  [(_ a b)
   (let ((t a))
     (if t t b))])
```

This version evaluates whatever expression was match as `a` in a temporary variable. This is useful if `a` included side effects, as without this explicit staging the side effects would be evaluated twice. But consider the naive expansion of `(let ((t #t)) (or #f t))`. This expression should evaluate to `#t`, but look at the naive expansion of this term:

```
(let ((t #t))
  (let ((t #f))
    (if t t t)))
```

In this version it is clear that it will return `#f` no matter what!

The second case is more subtle and needs a slightly longer example. Consider the following Harlan module.

```
(module
  (define (foo x)
    (+ 1 x))

  (define-macro double-foo ()
    [(_ n) (foo (foo n))])

  (define (main)
    (let ((foo (lambda (n) (+ 2 n))))
      (double-foo 0))))
```

This program includes a function, `foo`, which simply adds one to its argument. This function is referenced by the macro, `double-foo`, which expands into two successive calls to `foo`. Yet, before the macro is applied in `main` the `foo` binding is shadowed by a local definition that adds two instead of one. A properly hygienic macro system should ensure that the original `foo` function is used, as it is the one that was in scope when the macro was defined. This property makes it easy to reason about macros in isolation, as it is harder for surrounding code to affect their behavior.

Harlan ensures hygiene by carefully renaming variables during macro expansion. Although Harlan does not support polymorphism, macros can often be used to emulate polymorphic functions.

4.1.4. Arbitrarily nested arrays. The only built-in array type in Harlan is the one dimensional `vector` type, which is an ordered collection of elements of a uniform type. These elements, however, can be any other type, including other vectors. Thus, Harlan represents N-dimensional arrays as vectors nested to depth N. Each nested vector can have a different size, leading to arrays of nonuniform shape, or "ragged arrays." This is in contrast to languages like Accelerate **[14]** or Pochoir **[78]**, where multidimensional arrays must be rectangular. Figure 4.4 shows several examples of vectors that would be allowed and disallowed by Harlan.

4.1.5. In-kernel recursion. Recursion is a staple of any functional programming language, yet calling recursive functions from inside a kernel is explicitly disallowed by the OpenCL specification.[1] This is directly in conflict with the goal of allowing Harlan programs to use the same language inside kernels as outside. Harlan avoids OpenCL's restriction on recursive functions by creating an explicit stack and combining functions that may recursively call each other into a single function that uses `goto` for control flow.

[1]At least one OpenCL implementation uses inlining to entirely remove function calls, leading to unbounded code growth in the presence of recursive functions

```
(vector
  1
  (vector 2.0)
  (vector (vector 3)))
```

(A) This is not allowed because the elements do not have uniform types.

```
(vector
  (vector 1 2 3)
  (vector 4 5 6)
  (vector 7 8 9))
```

(B) This is allowed because the elements are all of the same type. The data structure happens to be rectangular as well.

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

(C) This is allowed because the elements are all of the same type. The data structure is not rectangular.

FIGURE 4.4. Several examples of vectors in Harlan.

4.1.6. Algebraic data types. In the interest of supporting rich data structures, Harlan supports algebraic data types. The `define-datatype` form is used to define custom tree-like types. Below is an example of how λ-Calculus expressions might be represented.

```
(define-datatype Expr
  (Variable int)
  (Lambda Expr)
  (App Expr Expr))
```

This defines a data type called `Expr` with constructors `Variable`, `Lambda` and `App`. The `match` form is used to inspect instances of an ADT through case analysis. The example below shows a single-step evaluator for λ-Calculus expressions.

```
(define (stepe e)
  (match e
    ((variable n) (variable n))
    ((Lambda e) (Lambda e))
    ((app e1 e2)
     (match e1
       ((variable n) (app (variable n) e2))
```

```
((Lambda e)
 (subst e 0 e2))
((app e1 e2)
 (app (stepe e1) e2))))))
```

4.1.7. First class procedures. While the existence of ADTs already gives Harlan high expressiveness, the Harlan compiler uses them internally to implement first class procedures. The `lambda` form creates new functions at runtime, and these functions can be passed as data even between kernel and non-kernel code.

Harlan compiles first class procedures using defunctionalization [**68**]. All lambdas that might flow to the same call site are combined into an ADT to describe the data contained in the lambda's closure and a dispatch function that applies the correct lambda body for each closure variant. Since lambdas are statically typed, the type of the procedure is used to determine which procedures may flow to a given call site. After all, two procedures of different types cannot be called in the same location.

4.2. 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.

4.2.1. Region Inference. Region inference happens alongside type inference using an approach similar to that of [**80**]. 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 inference algorithm 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. This this cause, 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 inference algorithm 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 inference algorithm will assign `v2` the type `(vec` ρ_2 `int)`.

Things are slightly more complicated for `v`. The type inference algorithm 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 inference algorithm 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:

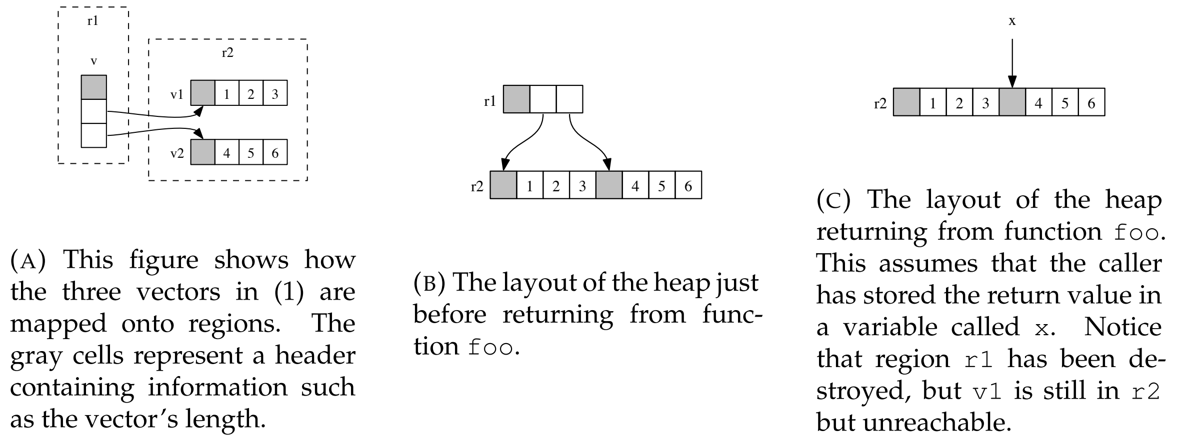
(1)

```
(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))))
```

Figure 4.5a shows how these vectors are grouped into regions. Figure 4.5b shows a diagram of the heap immediately before returning from `foo`, while Figure 4.5c 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.

4.2.2. 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



(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 that the caller has stored the return value in a variable called `x`. Notice that region `r1` has been destroyed, but `v1` is still in `r2` but unreachable.

FIGURE 4.5. Examples showing how data is assigned to and arranged within regions.

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 determined 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:

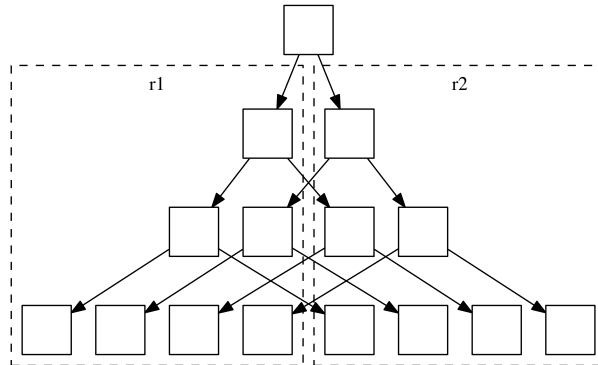


FIGURE 4.6. An example of a possible data structure if ADTs could take multiple region parameters. The leftward nodes are in region `r1` and the rightward nodes are in region `r2`.

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

An ADT requires a region parameter if it is immediately recursive, or if any of its fields are region-allocated.

As a simplification, ADTs contain exactly zero or one region parameters. There are no known 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))
```

Figure 4.6 shows a visual representation of a binary tree where all the leftward nodes are in one region and the rightward nodes are in another.

The final class of types that are affected by regions are closures.[2] 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.

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

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. The region inference process ensures this by allocating both vectors from the same region, rather than by inserting a copy operation to move each vector into the same region. Also, the Harlan compiler allows additional regions to appear in the argument types, meaning vectors that are passed into a procedure (rather than captured) can reside in different regions.

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.

4.2.3. 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 7.6.1), 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.

CHAPTER 5

Harlan Implementation

We will now turn our attention to Harlan's implementation. Harlan is compiled from its Scheme-like source language into C++ with OpenCL. Section 5.1 explores the compilation in more detail. Special care is needed in handling ADTs, first class procedures and using recursive functions from kernels. In Section 5.2, we will explore the details of the region system's implementation, as well as the rationale behind several design decisions and tradeoffs. Section 5.3 looks at some of the optimizations that are performed by the Harlan compiler. Finally, Section 5.4 shows how Harlan is able to provide more detailed error handling than many other GPU programming languages.

5.1. Compilation

The Harlan compiler is written in Scheme as a Nanopass-style compiler [**72**]. 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.

Harlan compiles to a relatively high level target language which simplifies some aspects of the compiler. For example, we will see in Sections 5.1.2 and 5.1.3 that some Harlan constructs compile into C structs and unions. Obviously it is helpful to rely on an existing compiler's support for these features rather than re-implementing it. There may be some benefit in compiler to a lower-level target, such as LLVM, PTX or SPIR. These targets

would give more precise control over the machine and could possibly allow better low level optimizations. For example, the code Harlan generates for region references seems likely to defeat the C++ compiler's attempts to analyze it.

The passes are roughly divided into a front end, a middle end and the back end. Figure 5.1 shows a detailed list of each of the passes in the compiler.

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 5.3). Finally, the back end generates a C++ program from the generated abstract syntax tree and invokes the C++ compiler to produce an executable program.

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

5.1.1.1. *Macro Expansion.* One of the first things done by the compiler is to expand macros into core syntax forms. The macro expander proceeds definition by definition adding new macro definitions to its environment. Each of these macro definitions consists of a number of match and rewrite patterns. The macro expander proceeds in the usual outside-in fashion detecting macro invocations and then rewriting each matched pattern according to its output template. In doing so, the expander must track where identifiers come from and rename as necessary to preserve hygiene.

5.1.1.2. *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

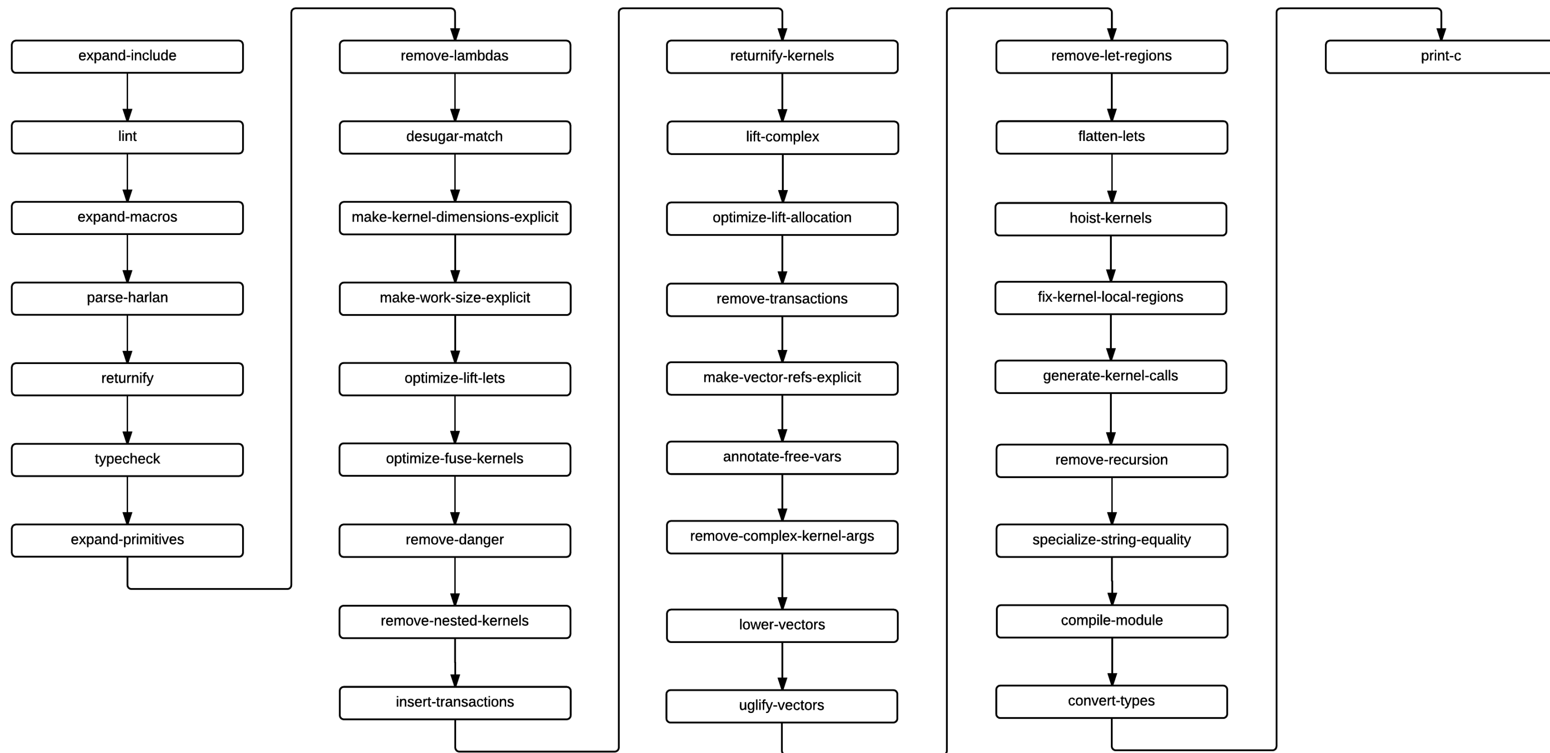


FIGURE 5.1. Harlan Compiler Passes.

`match` expressions into a chain of `if` statements that check the tag and bind the given variables to each field in the variant.

5.1.1.3. *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 5.3).

5.1.1.4. *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 [**71**]. More analysis could be used to determine the best point in a kernel nest to keep parallel.

5.1.1.5. *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.

5.1.1.6. *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 [**48**]. 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.

5.1.2. Algebraic Data Types. Once the frontend tasks like type inference are complete, there are three main things the Harlan compiler must do to implement ADTs.

(1) Generate data types
(2) Generate constructors
(3) Desugar `match` expressions

These three things are accomplished during a single compiler pass called `desugar-match`. Consider the following type definition:

```
(define-datatype Foo
  (Foo1)
  (Foo2 int int)
  (Foo3 Foo)
```

After type inference, this type will have a region parameter, because it is recursive. The type would then look something like this:

```
(define-datatype (Foo r)
  (Foo1)
  (Foo2 int int)
  (Foo3 (Foo r))
```

The Harlan compiler lowers this definition into a tagged union, which ultimately becomes a struct of unions in C. A tag field is added to determine which variant the given instance represents. The Harlan abstract syntax tree (AST) representation of this lower level data type looks something like

```
(define-datatype Foo
  (struct
```

```
    (tag int)
    (data (union
            (Foo1 (struct))
            (Foo2 (struct (f0 int) (f1 int))
            (Foo3 (struct (f0 (region_ptr))))))))
```

while the C version of the type looks like this:

```
struct Foo {
  int tag;
  union {
    struct {} Foo1;
    struct {
      int f0;
      int f1;
    } Foo2;
    struct {
      region_ptr f0;
    } Foo3;
  } data;
};
```

Since fields are never accessed by name, they must be given generated names by the Harlan compiler.

The next component of ADT implementation is creating the constructors for each variant of a type. Constructors in Harlan are invoked like any function, which is convenient because it allows to compiler to generate a function for each constructor. For each variant, the compiler creates a function that takes the data associated with the variant and then builds and returns an instance of the given type. Constructors for recursive ADTs will also take a region parameter which will be used to allocate any child nodes.

The final piece is to remove all `match` expressions and replace them with equivalent lower level expressions. Harlan converts `match` expressions into a sequence of nested `if` expressions that each check the tag of an ADT instance. Under each matched variant, the compiler inserts `let` bindings to extract the data from the ADT instance and make it available to the body expression. Consider the following expression, which uses the `Foo` data type we saw previously.

```
(match item
  ((Foo1)        (do-something))
  ((Foo2 a b)    (do-something-else a b))
  ((Foo3 item^) (do-something-recursive item^)))
```

This expression would expand into something similar to this:

```
(let ((tag (tag-of item)))
  (if (= tag Foo1-tag)
      (do-something)
      (if (= tag Foo2 tag)
          (let ((a (field (field item Foo2) f0))
                (b (field (field item Foo2) f1)))
            (do-something-else a b))
          ;; tag must be Foo3-tag at this point.
          (let ((item^ (field (field item Foo3) f0)))
            (do-something-recursive item^)))))
```

This code snippet includes two new intermediate language forms, `tag-of` and `field`. The `field` form accesses fields in a structure. For example, `(field item Foo3)` translates to `item.Foo3` in C. The `tag-of` operator returns the tag of an ADT. In this case, `(tag-of item)` is shorthand for `(field item tag)`.

5.1.3. First Class Procedures. We implement first class procedures by defunctionalization [**68**]. 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.

5.1.4. 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 [**79**] to find the strongly connected components. As an example, Figure 5.2 shows the call graph for a λ-Calculus interpreter written in Harlan. 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

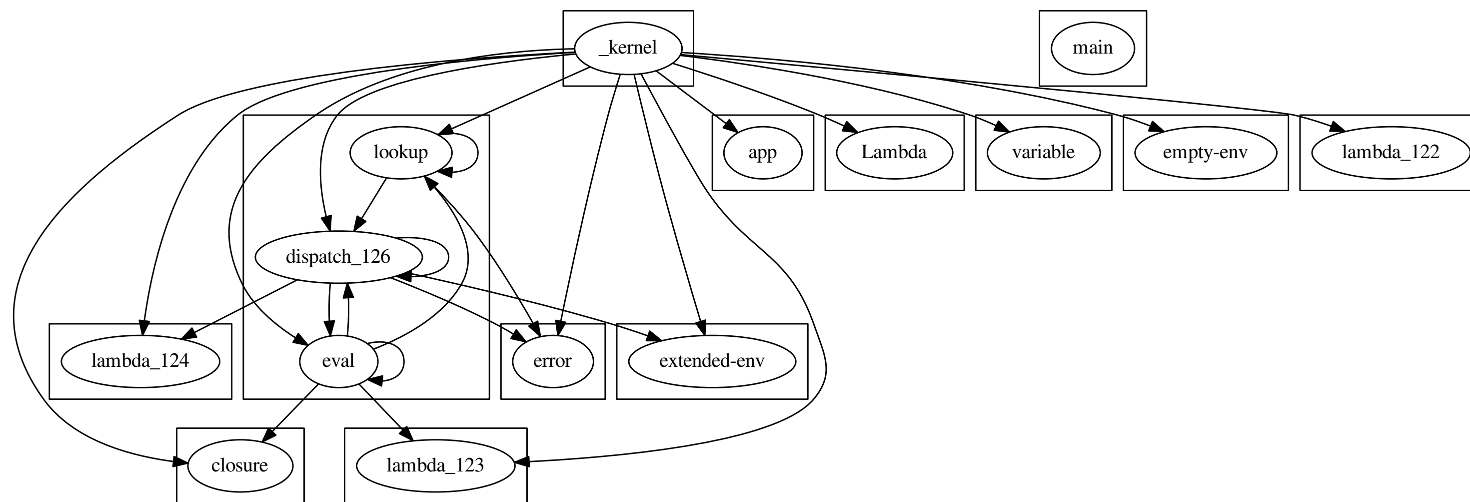


FIGURE 5.2. The call graph for a λ-Calculus interpreter written in Harlan. The SCCs are indicated by squares, while ovals represent functions and arrows indicate that a function may call another. Note that most of these functions are generated internally by the compiler.

```
(module
  (define (main)
    (let* ((add1 (lambda (x) (+ 1 x)))
           (add2 (lambda (x) (add1 (add1 x)))))
      (kernel ((i (iota 1)))
        (add1 i)))
    0))
```

FIGURE 5.3. An example of false recursion.

continuations being allocated from the same region, so instead we opted for an explicitly managed stack.

5.1.4.1. *False Recursion.* 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 that 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. Figure 5.3 shows a program that triggers this behavior. This makes having compiler support for recursion all the more important, as it may not be obvious or even under the programmer's control that a program would appear recursive to OpenCL. Alternatively, more precise control flow analysis could eliminate most of these cases.

5.2. Implementation of the Regions System

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 [**19**].

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.

5.3. 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.

5.3.1. 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)))
```

5.3.2. 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 [**17**]. 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.

5.3.3. 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 transferred. This approach does 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 advantage of this to more precisely transfer only the changed portions of a region.

5.3.4. Remote Vector Allocation. In order to avoid unnecessary round trips between the CPU and GPU memory, Harlan uses a different vector allocation path when allocating on the CPU from a region that is on the GPU. This is particularly important before executing a kernel, as space for the kernel's results must be allocated before launching the kernel. Originally the modification to the allocation pointer would trigger a region move from the GPU to the CPU. Instead, Harlan launches a small kernel that performs the allocation on the GPU, thereby allowing the region to remain on the GPU.

5.4. In-kernel Error Handling

In-kernel error handling is accomplished by adding an implicit argument to kernels, affectionately known as the *danger vector*. The danger vector includes a boolean entry for each type of error that might occur in a kernel, such as an allocation failure or an index out of bounds error. When a kernel thread encounters an error, it sets the appropriate entry in the danger vector to true. Upon exiting the kernel, the host code checks if any errors occurred and reports them to the user.

Several other strategies are possible which each have different implications for error reporting granularity and performance. Currently we have a small, predefined set of possible errors. Instead, each point in the kernel that can produce an error could be assigned a unique error identifier. This means Harlan could report the exact location of the error rather than simply reporting a generic class of error. This comes at the cost of a larger danger vector.

At the extreme end, the danger vector could include an entry for each kernel thread. Under this system, if a kernel encounters an error, the erroneous thread would fill its entry in the danger vector with an identifier for the error that occurred. This has the effect of providing very detailed information about kernel failures by showing which threads failed and why. We initially implemented this approach, but the memory overhead of having one entry in the danger vector for each thread led to an unacceptable performance penalty. This strategy may still be useful as an option for debugging applications.

5.4.1. Recovering from allocation failures. One extremely common error is not having enough space available in a region to allocate the results of a kernel. Harlan uses two strategies to handle these conditions. The first is to attempt to reserve enough space ahead of time for the results of a kernel, while the second is to automatically restart kernels with a larger region.

In the first case, many kernels have very predictable size requirements. Consider the kernel below.

```
(kernel ((i (iota 1000)))
```

```
(kernel ((j (iota 1000)))
  (* i j)))
```

This kernel produces a 1000×1000 array of integers, which can easily be seen by inspecting the kernel arguments. Since Harlan compiler can, in this case, determine the dimensions of the result statically, the compiler could reserve enough free space to ensure all allocations within a kernel complete successfully. This technique obviously cannot be applied in general, as kernels may allocate space based on arbitrary computation. Harlan currently does not implement this technique at all, but it could be added with minor changes to the compiler.

On the other hand, Harlan implements a dynamic checkpoint and retry strategy. The second method relies heavily on the fact that data in Harlan is immutable. Before launching the kernel, Harlan saves the allocation pointer of each of the regions written by a kernel. If the kernel completes unsuccessfully due to a lack of available space in the region, Harlan can simply roll the allocation pointer back to its previous value, increase the size of the region and try the kernel again. This checkpoint and retry process is introduced by the insert-transactions and remove-transactions passes (Figure 5.1). Note that these passes take place after nested kernels have been converted to sequential code by the remove-nested-kernels pass, so the checkpoint and retry process can only happen surrounding true kernels. This is necessary because trying to enlarge a region inside of a kernel runs into the same issue of not being able to allocate from the GPU.

CHAPTER 6

Region Semantics for Multi-memory Systems

This thesis claims that region-based memory is an effective way of enabling high-level language features when targeting machines with multiple disjoint memories. There is included within this an implicit safety claim, which stated vaguely is that programs should do what they are supposed to do. This chapter will make this safety claim more explicit by investigating a formal semantics for a representative model of the Harlan language. In this we will see what safety guarantees a language such as Harlan can provide as well as what is required of the type and region system to ensure these safety guarantees.

One key aspect of Harlan's region system is that regions move between memory spaces. This leads to an important safety consideration—programs must only access data in a region if that region is in the currently active memory space. This is the primary safety property that we will prove in this chapter.

The general approach will be to implement an interpreter for a representative subset of Harlan. This interpreter will then be used to derive an abstract machine. This machine will form the basis for an operational semantics that will then be used to explore Harlan's region system.

There are several tasks included as part of developing a semantics:

(1) Implement an interpreter for a core fragment of Harlan (Section 6.2.1)
(2) Derive an abstract machine from the interpreter (Section 6.2.2).
(3) Develop a static semantics for Harlan that describes the type and region rules (Section 6.4).
(4) Define and prove a meaningful safety property (Section 6.5).

The type system developed in Section 6.4 relies heavily on separation logic to simplify the handling of region locations as the execution of a program progresses, so we include an overview of separation logic in Section 6.3.

6.1. Core Harlan

Rather than consider the semantics of the whole Harlan language, we will consider a small subset. We are primarily concerned with Harlan's region system as it relates to multiple memory locations. Thus, our language will consist of a small functional language augmented with the ability to allocate and read from regions, move regions between locations, and move computation between locations. The system considered here includes two locations, called CPU and GPU, but this system could easily generalize to more memory locations.

While regions do not generally appear in full Harlan source language, Core Harlan explicitly mentions regions. Core Harlan represents an intermediate language in which the compiler has already inserted region annotations as needed. The semantics presented in this chapter defines a set of invariants that Harlan's type and region inference algorithm must maintain.

Figure 6.1 shows the syntax for the fragment of Harlan we will consider. The `()` term is read as "unit" and constructs a value. This is similar to a base value from the Simply Typed Lambda Calculus [**33**]. The x term is a variable reference, while `(`e_1 e_2`)` represents the application of e_1 to e_2. References are created by `(ref` **r** e`)` allocates space from region r and stores the value of e in this location. The result is a pointer that can be used in `(deref` **r** e`)` to read a value from the region. The `(push` **r**`)` expression moves a region from the CPU to the GPU (or leaves the region on the GPU if it is already there) and `(pull` **r**`)` moves a region from the GPU to the CPU (similarly leaving the region unchanged if it is already on the CPU). There is not a similar construct to `push` and `pull` in the Harlan source language, but these operations are inserted by the compiler. Regions are created with `let-region`, where `(let-region` **r** e`)` creates a region **r** which is available during the evaluation of the body, e.

$$
\begin{array}{rcl}
e & ::= & \texttt{()} \\
 & | & x \\
 & | & \texttt{(spawn } e\texttt{)} \\
 & | & \texttt{(let-region } \mathbf{r}\ e\texttt{)} \\
 & | & \texttt{(}e_1\ \ e_2\texttt{)} \\
 & | & \texttt{(ref } \mathbf{r}\ e\texttt{)} \\
 & | & \texttt{(deref } \mathbf{r}\ e\texttt{)} \\
 & | & \texttt{(push } \mathbf{r}\texttt{)} \\
 & | & \texttt{(pull } \mathbf{r}\texttt{)} \\
 & | & \texttt{(lambda } l\ \ \texttt{[}\mathbf{r} : l \ldots\texttt{]}\ \ \texttt{(}x\texttt{)}\ \ e\texttt{)} \\
 & & \\
l & ::= & \texttt{CPU} \mid \texttt{GPU}
\end{array}
$$

FIGURE 6.1. The syntax for Core Harlan. This is a small model of the Harlan language which we will use to study its semantics. The e non-terminal represents Harlan expressions. Variables are indicated by x, while $\mathbf{r}$ indicates a region variable. l represents a location, either CPU or GPU.

The `lambda` construct is rather more complicated than traditional `lambda`s. The first argument, l, specifies what location the resulting function may be called from. Functions labeled as `GPU` may be called from the CPU as well, but `CPU`-labeled functions may only be used on the CPU. The reason for this distinction is that some operations, such as `push` and `pull`, may only be used on the CPU. The next list of $\mathbf{r} : l$ pairs specifies which regions the function will use, and what location the function expects them to be in. Finally, x specifies the name of the formal parameter and e specifies the body of the function.

Core Harlan uses `spawn` to perform a computation on the GPU. The `spawn` operation represents spawning a kernel, and like kernels in Harlan, `spawn` expressions may be nested. In this case, only the outermost `spawn` expression changes the computation location; any nested `spawn`s remain on the GPU. Unlike kernels in Harlan, there is no parallelism involved with `spawn`. This may seem surprising, but the goal of the semantics is to show Harlan is safe with respect to moving data between different memory spaces, which is independent of the parallelism associated with either device.

Several features of the full Harlan language, such as ADTs and vectors, are not included in this core fragment in order to simplify the reasoning about the semantics. These can be emulated with the constructs included in Core Harlan, or easily added if needed.

```
(define (eval-harlan e env)
  (match e
    ('() '())
    (x #:when (symbol? x)
     (lookup x env))
    (`(lambda ,l [(,r* : ,l*) ...] (,x) ,e)
     `(closure ,x ,env ,e))
    (`(,e1 ,e2)
     (match-let ((`(closure ,x ,env^ ,e) (eval-harlan e1 env))
                 (arg  (eval-harlan e2 env)))
       (eval-harlan e (extend env^ x arg))))))
```

FIGURE 6.2. The baseline interpreter for the λ-Calculus. Procedures (`lambda`) have been extended with location and region requirement annotations that are part of Core Harlan but not standard λ-Calculus.

6.2. Operational Semantics

We will now develop an operational semantics for Core Harlan. We start with an interpreter and then derive an abstract machine. Starting with an interpreter makes it easy to test and experiment with the language, ensuring the programs behave as expected. Once this is completed, we will use the interpreter to derive an abstract machine. The abstract machine consists of a machine configuration combined with rules for how to derive the next configuration. The resulting machine will have the property that the transition function is a total and non-recursive function, a fact which will prove especially for mechanizing the type safety proof (Section 6.7).

6.2.1. Developing the interpreter. We start with a standard "three-line interpreter" for the λ-Calculus written in Racket **[25]**. This interpreter is shown in Figure 6.2.

The `lookup` and `extend` functions are used to find values in the environment and add new values to the environment. One possible implementation of these two procedures is given below.

```
(define (lookup x env)
  (cdr (assq x env)))


(define (extend env x value)
```

```
  (cons (cons x value) env))
```

Obviously, this isn't quite a three line interpreter, as it also includes a match clause for unit expressions. Otherwise, it is a standard interpreter for the λ-Calculus. The variable line simply pulls the appropriate value out of the runtime environment using `lookup`. We have omitted the error-handling code here; this interpreter assumes programs will not try to access unbound variables. The `lambda` line just captures the current runtime environment and the body of the procedure in a closure data structure. Notice that we ignore the location and region requirement annotations. These are needed only for type checking but are not used at runtime. The type system is responsible for ensuring that the procedure's region requirements are always met, but a more robust interpreter could check these dynamically for better debugging support.

The interpreter as it stands now does not support the majority of Core Harlan. The remaining forms affect the region system, so we will extend the interpreter with stores. One key difference is that we have two stores, `s1` and `s2`, which serve as the *primary* and *alternate* stores. The primary store is the store that is active on the currently executing device, while the alternate store contains the data on the alternate device. This will become more clear when we see how the interpreter handles the `spawn` form. Figure 6.3 shows the interpreter extended with stores and also a location variable that indicates which device is currently executing. Because stores are mutated during program execution, we must be careful with the order of evaluation and thread the stores through so that effects will not be lost. Notice that the interpreter now returns three values instead of just one. The first is the value to which the first expression was evaluated. The second and third are the updated primary and alternate stores.

We can now add match classes each of the missing forms in turn. We will start with `spawn`:

```
[`(spawn ,e)
 (match loc
   ['CPU (let-values ([(v s2 s1) (eval-harlan e env 'GPU s2 s1)])
```

```
(define (eval-harlan e env loc s1 s2)
  (match e
    ('() (values '() s1 s2))
    (x #:when (symbol? x)
       (values (lookup x env) s1 s2))
    (`(lambda ,l [(,r* : ,l*) ...] (,x) ,e)
     (values `(closure ,x ,env ,e) s1 s2))
    (`(,e1 ,e2)
     (match-let*-values
      ([(`(closure ,x ,env^ ,e) s1 s2)
         (eval-harlan e1 env loc s1 s2)]
       [(arg s1 s2)
         (eval-harlan e2 env loc s1 s2)])
      (eval-harlan e (extend env^ x arg) loc s1 s2)))))
```

FIGURE 6.3. The interpreter from Figure 6.2 modified to thread store and location variables throughout the execution.

```
((r1 . (v0 v2 v3))
 (r2 . (v4))
 (r3 . ())
 (r4 . (v5 v6 v7 v8)))
```

FIGURE 6.4. An example store.

```
          (values v s1 s2))]
   ['GPU (eval-harlan e env loc s1 s2)]))]
```

Spawn simulates executing a kernel, but without the parallelism from Harlan's `kernel` form. Like `kernel` in Harlan, `spawn` can be nested. The `spawn` form indicates code should execute on the GPU, but if the program is already executing on the GPU then `spawn` simply passes through the evaluation of its subexpression. This is the reason for matching on `loc`. Notice that in the CPU case, where the execution switches from the CPU to the GPU, the two stores are swapped on entrance to the recursive call to `eval-harlan` and swapped again on its return.

The remaining forms directly interaction with the stores. We will represent stores as an association list from region identifiers to a list of values. The contents of the region itself are indexed by a number that can be thought of as the memory address. Figure 6.4 gives an example of what a store may look like.

Given this representation, the `let-region` form simply adds an empty region to the primary store:

```
[`(let-region ,r ,e)
 (eval-harlan e env loc (add-new-region r s1) s2)]
```

The definition of `add-new-region` can be seen in Figure 6.5.

Values are added to the store using `ref`. In this case, the interpreter first evaluates the subexpression to a value and then the `alloc-in-region` helper defined in Figure 6.5 appends the value to the appropriate region. The return values from `alloc-in-region` include an index which the interpreter saves in a label that is returned as the value of the `ref` expression. The code for the `ref` case is given below.

```
[`(ref ,r ,e)
 (let-values ([(v s1 s2) (eval-harlan e env loc s1 s2)])
   (let-values ([(s1 i) (alloc-in-region r v)])
     (values `(label ,i) s1 s2)))]
```

For the symmetric operation, `deref`, the interpreter first evaluates its argument to a label and then uses this to lookup the resulting value from the store. This is shown below.

```
[`(deref ,r ,e)
 (match-let-values ([(`(label ,i) s1 s2)
                     (eval-harlan e env loc s1 s2)])
    (values (lookup-store s1 r i) s1 s2))]
```

Push and pull are similar, so we will consider them together. These simply return `()`, but their importance is in the effect of moving a region between the primary and alternate store. The `push` and `pull` cases are shown below.

```
[`(push ,r)
 (match loc
   ['CPU (let-values ([(s1 s2) (move-region r s1 s2)])
           (values '() s1 s2))]
   ['GPU (error 'eval-harlan
```

```
                  "Push can only be used on the CPU")]))]
[`(push ,r)
 (match loc
   ['CPU (let-values ([(s1 s2) (move-region r s1 s2)])
           (values '() s1 s2))]
   ['GPU (error 'eval-harlan
                  "Pull can only be used on the CPU")]))]
```

Unlike our other cases, this case does some error checking. The reason is that in keeping with our model of CPU/GPU computing, region movement can only be initiated on the CPU. In the CPU case, we simply use `move-region` (defined in Figure 6.5) to move the given region between the two stores. In the case of `push`, the interpreter moves the region from the primary store to the alternate store. Likewise, for `pull`, the interpreter moves the region from the GPU store to the CPU store.

Having discussed each of these cases individually, we now present the complete interpreter in Figure 6.6.

6.2.2. Developing the abstract machine. Given our interpreter, we can derive an abstract machine through a relatively straightforward series of transformations. The gist of these transformations is that we first need to convert the interpreter to CPS and then replace all of the continuations with explicit data structures. We shall elide most of these details and instead focus on the finished product in more formal notation.

The abstract machine has four different kinds of configurations, which are:

(1) $\langle v, s_1, s_2 \rangle$
(2) $\perp$
(3) $\langle e, \rho, s_1, s_2, k, l \rangle$
(4) $\langle k, v, \rho, s_1, s_2, l \rangle$

Configuration (1) is simply a value, meaning the machine terminated with the given result. Possible values are shown in Table 6.1. The value machine configuration includes two stores, s_1 and s_2 because v may contain references into the source. Being able to type

```
(define (add-new-region r store)
  (cons (list r) store))

(define (alloc-in-region r store value)
  (if (eq? r (caar store))
      (values (cons (append (car store) (list value))
                    (cdr store))
              (length (cadr store)))
      (cons (car store)
            (alloc-in-region r (cdr store) value))))

(define (lookup-store s r i)
  (list-ref (cdr (assq r s)) i))

(define (move-region r s1 s2)
  (let ((r (assq r s1))
        (s1 (let loop ((s1 s1))
              (if (eq? (caar s1) r)
                  (cdr s1)
                  (cons (car s1) (loop (cdr s1)))))))
    (values s1 (if r (cons r s2) s2))))
```

FIGURE 6.5. Region manipulation functions.

assign a type to these values requires a store that is consistent with the type, so we include a store in the machine configuration. Configuration (2) means an error occurred, such as attempting to apply a non-procedure. We will refer to type (3) as an expression configuration, which contains an expression to be evaluated. Here, e is the expression under evaluation, ρ is a lexical environment mapping variable names onto values. The active store is given by s_1, which maps region names onto regions, which in turn map labels onto values. Similarly, s_2 represents the alternate store, which is the memory on the device that is not currently computing. The machine exchanges s_1 and s_2 when `spawn` changes the compute location. This is the behavior we saw in the `spawn` case of the interpreter and will see again in Table 6.3. Finally, k gives the continuation to be applied once e has been evaluated to a value and l is the location where evaluation is currently taking place.

Configuration (4) is a continuation configuration, which contains a continuation and the value (v) to which it is applied. The variables k, ρ, s_1, s_2 and l are as they were for expression configurations.

```
(define (eval-harlan e env loc s1 s2)
  (match e
    ('() (values '() s1 s2))
    (x #:when (symbol? x)
       (values (lookup x env) s1 s2))
    [`(spawn ,e)
     (match loc
       ['CPU (let-values ([(v s2 s1)
                           (eval-harlan e env 'GPU s2 s1)])
               (values v s1 s2))]
       ['GPU (eval-harlan e env loc s1 s2)])]
    [`(let-region ,r ,e)
     (eval-harlan e env loc (add-new-region r s1) s2)]
    [`(ref ,r ,e)
     (let-values ([(v s1 s2) (eval-harlan e env loc s1 s2)])
       (let-values ([(s1 i) (alloc-in-region r s1 v)])
         (values `(label ,i) s1 s2)))]
    [`(deref ,r ,e)
     (match-let-values ([(`(label ,i) s1 s2)
                         (eval-harlan e env loc s1 s2)])
        (values (lookup-store s1 r i) s1 s2))]
    [`(push ,r)
     (match loc
       ['CPU (let-values ([(s1 s2) (move-region r s1 s2)])
               (values '() s1 s2))]
       ['GPU (error 'eval-harlan
                    "Push can only be used on the CPU")])]
    [`(push ,r)
     (match loc
       ['CPU (let-values ([(s2 s1) (move-region r s2 s1)])
               (values '() s1 s2))]
       ['GPU (error 'eval-harlan
                    "Pull can only be used on the CPU")])]
    (`(lambda ,l [(,r* : ,l*) ...] (,x) ,e)
     (values `(closure ,x ,env ,e) s1 s2))
    (`(,e1 ,e2)
     (match-let*-values
      ([(`(closure ,x ,env^ ,e) s1 s2)
        (eval-harlan e1 env loc s1 s2)]
       [(arg s1 s2)
        (eval-harlan e2 env loc s1 s2)])
      (eval-harlan e (extend env^ x arg) loc s1 s2)))))
```

FIGURE 6.6. The full interpreter for Core Harlan.

Value	Description
`()`	Unit
`(closure` x ρ e`)`	A closure. The formal parameter name is given by x, the captured environment is ρ and the body of the procedure is e.
`(label` i`)`	A memory address which points into a particular region.

TABLE 6.1. The syntax of values.

Error Configuration		
$\bot$	$\rightsquigarrow$	$\bot$
Value Configuration		
$\langle v, s_1, s_2 \rangle$	$\rightsquigarrow$	$\langle v, s_1, s_2 \rangle$

TABLE 6.2. Core Harlan transition function (Error and Value configurations).

6.2.3. Transition Function. Now that we have seen the syntax of Core Harlan and the associated values and machine configurations, we can develop the operational semantics. We use a small step operational semantics **[66]**. One key feature is that the step function we develop here is total. This is to aid in developing a mechanized proof (Section 6.7). Tables 6.2 to 6.4 give the definition of the relation $M \rightsquigarrow M'$, meaning that machine configuration M steps to machine configuration M'.

These rules make use of a number of auxiliary functions and notational shorthands. We treat runtime environments, ρ, as partial functions from variable names to values. Thus, $\rho(x)$ corresponds to `(lookup x` ρ`)` in the interpreter and $\mathrm{dom}(\rho)$ gives the set of names that are defined in the environment. Environments are written as a list of $[x : v]$ pairs but are extended by simply juxtaposing pairs, as in $[x : v]\rho$.

Stores are treated similarly. Stores are partial functions from region names to regions, which in turn are partial functions that map integers from 0 to some n to values. We use $s(\mathbf{r})$ to return the entire region $\mathbf{r}$ and $s(\mathbf{r}, i)$ as a shorthand for $s(\mathbf{r})(i)$. Like environments, stores are written as a list of $[\mathbf{r} : v \ldots]$ pairs but are be concatenated using the :: operator. As with environments, we use $\mathrm{dom}(s)$ to get the set of region names included in the store. We use the notation $s - \{\mathbf{r}\}$ to denote the restriction of s, which is a new store that is identical to s except without $\mathbf{r}$ in its domain.

Expression Configurations		
$\langle \texttt{()}, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle k, \texttt{()}, \rho, s_1, s_2, l \rangle$
$\langle x, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle k, \rho(x), \rho, s_1, s_2, l \rangle$
		if $x \in \text{dom}(\rho)$
	$\rightsquigarrow$	$\bot$ otherwise
$\langle \texttt{(spawn } e\texttt{)}, \rho, s_1, s_2, k, \texttt{CPU} \rangle$	$\rightsquigarrow$	$\langle e, \rho, s_2, s_1, \text{Unspawn}(k, \texttt{CPU}), \texttt{GPU} \rangle$
$\langle \texttt{(spawn } e\texttt{)}, \rho, s_1, s_2, k, \texttt{GPU} \rangle$	$\rightsquigarrow$	$\langle e, \rho, s_1, s_2, \text{Unspawn}(k, \texttt{GPU}), \texttt{GPU} \rangle$
$\langle \texttt{(}e_1\ \ e_2\texttt{)}, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle e_1, \rho, s_1, s_2, \text{Rator}(e_2, k), l \rangle$
$\langle \texttt{(ref } \mathbf{r}\ \ e\texttt{)}, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle e, \rho, s_1, s_2, \text{Alloc}(\mathbf{r}, k), l \rangle$
$\langle \texttt{(deref } \mathbf{r}\ \ e\texttt{)}, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle e, \rho, s_1, s_2, \text{Deref}(\mathbf{r}, k), l \rangle$
$\langle \texttt{(push } \mathbf{r}\texttt{)}, \rho, s_1, s_2, l, \texttt{CPU} \rangle$	$\rightsquigarrow$	$\langle k, \texttt{()}, \rho, s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r}), \texttt{CPU} \rangle$
		if $\mathbf{r} \in \text{dom}(s_1)$
	$\rightsquigarrow$	$\langle k, \texttt{()}, \rho, s_1, s_2, \texttt{CPU} \rangle$
		if $\mathbf{r} \notin \text{dom}(s_1)$ and $\mathbf{r} \in \text{dom}(s_2)$
	$\rightsquigarrow$	$\bot$ otherwise
$\langle \texttt{(pull } \mathbf{r}\texttt{)}, \rho, s_1, s_2, l, \texttt{CPU} \rangle$	$\rightsquigarrow$	$\langle k, \texttt{()}, \rho, s_1 :: s_2(\mathbf{r}), s_2 - \{\mathbf{r}\}, \texttt{CPU} \rangle$
		if $\mathbf{r} \in \text{dom}(s_2)$
	$\rightsquigarrow$	$\langle k, \texttt{()}, \rho, s_1, s_2, \texttt{CPU} \rangle$
		if $\mathbf{r} \notin \text{dom}(s_2)$ and $\mathbf{r} \in \text{dom}(s_1)$
	$\rightsquigarrow$	$\bot$ otherwise
$\langle \texttt{(let-region } \mathbf{r}\ \ e\texttt{)}, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle e, \rho, [\mathbf{r} : \cdot] :: s_1, s_2, \text{Dealloc}(\mathbf{r}, k), l \rangle$
$\langle \texttt{(lambda } l\ \texttt{[}\mathbf{r} : l \ldots\texttt{]}\ \ (x)\ \ e\texttt{)}, \rho, s_1, s_2, k, l \rangle$	$\rightsquigarrow$	$\langle k, \texttt{(closure } x\ \ \rho\ \ e\texttt{)}, \rho, s_1, s_2, l \rangle$

TABLE 6.3. Core Harlan transition function (Expression configurations).

Continuation Configurations		
$\langle \text{Empty}, v, \rho, s_1, s_2, \texttt{CPU} \rangle$	$\rightsquigarrow$	$\langle v, s_1, s_2 \rangle$
$\langle \text{Unspawn}(k, \texttt{CPU}), v, \rho, s_1, s_2, \texttt{GPU} \rangle$	$\rightsquigarrow$	$\langle k, v, \rho, s_2, s_1, \texttt{CPU} \rangle$
$\langle \text{Unspawn}(k, \texttt{GPU}), v, \rho, s_1, s_2, \texttt{GPU} \rangle$	$\rightsquigarrow$	$\langle k, v, \rho, s_1, s_2, \texttt{GPU} \rangle$
$\langle \text{Rator}(e, k), v, \rho, s_1, s_2, l \rangle$	$\rightsquigarrow$	$\langle e, \rho, s_1, s_2, \text{Rand}(v, k), l \rangle$
$\langle \text{Rand}(\texttt{(closure } x\ \ \rho'\ \ e\texttt{)},,) v, \rho, s_1, s_2, l \rangle$	$\rightsquigarrow$	$\langle e, [x : v]\rho', s_1, s_2, \text{Return}(\rho, k), l \rangle$
$\langle \text{Alloc}(\mathbf{r}, k), v, \rho, s_1, s_2, l \rangle$	$\rightsquigarrow$	$\langle k, \texttt{(label } i\texttt{)}, \rho, s_1[r := s_1(r) :: v], s_2, l \rangle$
		where $i = \text{length}(s_1)$ and $\mathbf{r} \in \text{dom}(s_1)$
$\langle \text{Deref}(\mathbf{r}, k), \texttt{(label } i\texttt{)}, \rho, s_1, s_2, l \rangle$	$\rightsquigarrow$	$\langle k, s_1(\mathbf{r}, i), \rho, s_1, s_2, l \rangle$
		if $\mathbf{r} \in \text{dom}(s_1)$ and $i < \text{length}(s_1(\mathbf{r}))$
$\langle \text{Dealloc}(\mathbf{r}, k), v, \rho, s_1, s_2, l \rangle$	$\rightsquigarrow$	$\langle k, v, \rho, s_1 - \{\rho\}, s_2 - \{\rho\}, l \rangle$
$\langle \text{Return}(\rho, k), v, \rho', s_1, s_2, l \rangle$	$\rightsquigarrow$	$\langle k, v, \rho, s_1, s_2, l \rangle$
Otherwise		
M	$\rightsquigarrow$	$\bot$ otherwise

TABLE 6.4. Core Harlan transition function (Continuation and Otherwise configurations).

The transition function rules generally follow the behavior of the interpreter. One key difference is that all erroneous behavior has been made explicit. Also, notice in the Unspawn continuation cases that the stores are reversed when switching from the GPU to the CPU. This mirrors the behavior of the interpreter.

6.3. A Separation Logic Primer

As we will see in Section 6.4, the type system for Harlan depends strongly on the locations of regions. Dereferencing a pointer into a region requires the region to be on the current device. In order to ensure memory safety, the type system must guarantee that when a pointer is dereferenced the target region is in the correct location.

The most obvious way to do this is to thread a mapping of regions to locations throughout the typing rules that is updated to reflect region creation and movement. This quickly becomes unwieldy, as the region mapping contains many regions that are irrelevant to any given operation.

This problem is similar to reasoning about the types of memory locations, especially in the presence of mutable references. Separation logic is a technique that simplifies reasoning about the state of memory [**69**]. Separation logic works in terms of assertions about a heap, where a heap is a mapping from keys, such as labels or addresses, to values or types. An example assertion is $l \mapsto \tau$, which states that a heap maps a location l to a value of type τ. More interesting is the separating conjunction operator, $\star$. The statement $\Delta \star \Delta'$ states that the heap can be split or separated into two disjoint parts, one which satisfies Δ and another which satisfies Δ'.

An extremely powerful tool in separation logic is the frame rule:

$$\frac{\{P\}C\{Q\}}{\{R \star P\}C\{R \star Q\}}$$

Intuitively, the frame rule means that if a statement is true in one heap, that statement remains true in any extension of that heap and it leaves the extended portion of the heap unchanged. Crucially, the frame rule allows for local reasoning about portions of a program.

Harlan uses a version of separation logic in its typing rules. The rules include a statement that must be true about the heap in which the program runs. For the purposes of Harlan's type system, a heap is a mapping from regions onto locations, either CPU or GPU.

6.3.1. Heap Assertions. The typing rules state conditions about the locations of regions using assertions inspired by separation logic. The letter Δ is used to represent a heap assertion, and can be one of the following.

- $\bullet$ An empty heap assertion. This assertion makes no claims about the heap.
- $[\mathbf{r} \mapsto l]$ A singleton heap. This assertion claims only that the region $\mathbf{r}$ is in location l.
- $\Delta_1 \star \Delta_2$ Separating conjunction of two heaps. This is the union of Δ_1 and Δ_2. The domains of Δ_1 and Δ_2 must be disjoint.

For the purposes of the semantics of Core Harlan, we must define what it means for a heap assertion to describe a certain pair of stores. We use the judgment $\Delta; l \vdash s_1, s_2$ to indicate this. The l parameter is the location of s_1, while s_2 is the store that is in the alternate location. Thus judgment is defined formally below.

DEFINITION 1 (Heap Consistency). *We say* $\Delta; l \vdash s_1, s_2$ *if and only if whenever there exists a* Δ' *such that* $\Delta = \Delta' \star [\mathbf{r} \mapsto l]$ *then* $\mathbf{r} \in \mathrm{dom}(s_1)$ *and whenever there exists a* Δ' *such that* $\Delta = \Delta' \star [\mathbf{r} \mapsto \mathrm{other}(l)]$ *then* $\mathbf{r} \in \mathrm{dom}(s_2)$

Note that the $\star$ operator is commutative and associative. We also treat $\bullet$ as an identity for $\star$. That is, for any Δ, $\Delta = \Delta \star \bullet = \bullet \star \Delta$.

6.4. Type System

We shall now develop a type system that will allow us to state and prove the desired safety properties for Harlan. These rules use the following parameters:

- Γ - This is a traditional type environment. It maps variable names onto types. In addition, Γ contains the region identifiers that are in scope.

$$\begin{array}{rcl} \tau & ::= & () \\ & | & (\text{ref } \mathbf{r}\ \tau) \\ & | & \tau \xrightarrow{\Delta;\Delta'}_{l} \tau' \end{array}$$

FIGURE 6.7. Syntax of types for Core Harlan.

$$\text{TM-VALUE}\quad \frac{\Sigma \vdash s_1, s_2 \qquad \Sigma \vdash v : \tau}{\langle v, s_1, s_2 \rangle : \tau}$$

$$\text{TM-EXPR}\quad \frac{\Sigma \vdash \Gamma \sim \rho \qquad \Gamma;\Delta;l \vdash e : \tau';\Delta' \qquad \Sigma \vdash s_1, s_2 \qquad \Delta;l \vdash s_1, s_2 \qquad \Gamma;\Delta';\Sigma;l \vdash k : \tau' \to \tau}{\langle e, \rho, s_1, s_2, k, l \rangle : \tau}$$

$$\text{TM-CONT}\quad \frac{\Sigma \vdash \Gamma \sim \rho \qquad \Delta;l \vdash s_1, s_2 \qquad \Sigma \vdash s_1, s_2 \qquad \Gamma;\Delta;\Sigma;l \vdash k : \tau' \to \tau \qquad \Sigma \vdash v : \tau}{\langle k, v, \rho, s_1, s_2, l \rangle : \tau}$$

FIGURE 6.8. Machine typing rules.

- Δ - This is a heap assertion (Section 6.3.1), which states facts about the location of the regions used by an expression.
- Σ - This is a store typing, which makes pairs of region names and indices, $(\mathbf{r}, i)$, to types.

In Figure 6.7, we see the syntax for types in Core Harlan. We use () as a base type. References have type (ref $\mathbf{r}$ τ), which indicates a pointer into region $\mathbf{r}$ to a value of type τ. Function types, $\tau \xrightarrow{\Delta;\Delta'}_{l} \tau'$, indicate a function that takes a value of type τ and returns a value of type τ'. The l annotation indicates whether the function may execute on the CPU or GPU. Δ indicates the function's requirements on the state of the heap upon entry and Δ' reflects the state of the heap upon the function's exit.

The typing rules are presented in a top down fashion, starting with the typing for machine configurations.

6.4.1. Machine Typing Rules. The judgment $M : \tau$ is presented in Figure 6.8. It is read "machine configuration M has type τ."

The TM-VALUE rule states that a final value configuration of a machine has a given type. This is established by showing that the return value has a given type, using the rules in Figure 6.13. However, values are meaningless apart from their associated stores, so the value typing judgment relies on a store typing environment, Σ. In turn, we must show that Σ accurately reflects the state of the stores. This statement is captured by the $\Sigma \vdash s_1, s_2$ condition, which is defined below.

DEFINITION 2 (Well-typed Stores). $\Sigma \vdash s_1, s_2$ *if and only if* $\forall (r,i) \in \mathrm{dom}(\Sigma), \Sigma \vdash s_1(r,i) : \Sigma(r,i)$ *or* $\Sigma \vdash s_2(r,i) : \Sigma(r,i)$.

The well-typed stores condition needs to consider both the primary and alternate stores, but it is not concerned with the location of regions between these two stores. In other words, moving a region between devices does not affect the type of any values.

The TM-EXPR rule requires a number of preconditions in order to show that an expression evaluation machine configuration has the correct type. The two key conditions are that the continuation k will produce a value of the correct type ($\Gamma; \Delta'; \Sigma; l \vdash k : \tau' \to \tau$) and that evaluation of the expression will produce a value of the correct type for the continuation ($\Gamma; \Delta; l \vdash e : \tau'; \Delta'$). It is important to notice how the heap assertions evolve throughout the computation. The expression typing judgment includes both an initial heap assertion, Δ, and an outgoing heap assertion Δ'. This is to account for the changes to the location of regions that can result from evaluating e, such as when e contains push and pull expressions. Because evaluating the argument to the continuation completes entirely before executing the continuation, the continuation typing judgment uses the outgoing heap assertion, Δ'.

There are several other conditions in TM-EXPR that are necessary to ensure everything is consistent. As before, we need to show that there is a store typing, Σ, that reflects the state of the stores s_1 and s_2. Similarly, we must show that the heap assertion is consistent with the current execution location and the stores, stated as $\Delta; l \vdash s_1, s_2$. Finally, we must show that the typing environment Γ is consistent with the values in the runtime environment ρ, stated as $\Sigma \vdash \Gamma \sim \rho$. These two conditions are defined formally below.

DEFINITION 3 (Well-typed Environments). *We say an environment ρ is well-typed with respect to a type environment Γ and a store typing Σ, written $\Sigma \vdash \Gamma \sim \rho$, if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\rho)$ and $\forall x \in \mathrm{dom}(\rho), \Sigma \vdash \rho(x) : \Gamma(x)$.*

DEFINITION 4 (Consistent Stores). *Two stores s_1 and s_2 are consistent with Δ from location l, written $\Delta; l \vdash s_1, s_2$, if for any $\mathbf{r}$, there exists a Δ' such that $\Delta = \Delta' \star [\mathbf{r} \mapsto l]$ if and only if $\mathbf{r} \in \mathrm{dom}(s_1)$ and there exists another Δ' such that $\Delta = \Delta' \star [\mathbf{r} \mapsto \mathrm{other}(l)]$ if and only if $\mathbf{r} \in \mathrm{dom}(s_2)$.*

Finally, we consider the TM-CONT rule. For this machine configuration, we have a continuation k being applied to a value v. The $\Gamma; \Delta; \Sigma; l \vdash k : \tau' \to \tau$ tells us that k expects a value of type τ' to produce a value of type τ, which is provided by $\Sigma \vdash v : \tau$. As before, we need everything to be consistent, which we get from $\Sigma \vdash \Gamma \sim \rho$, $\Delta; l \vdash s_1, s_2$ and $\Sigma \vdash s_1, s_2$ conditions.

6.4.2. Expression Typing Rules. We have already seen the expression typing judgment in the previous section. The $\Gamma; \Delta; l \vdash e : \tau; \Delta'$ judgment is defined in Figure 6.9. The judgment can be read as "in typing environment Γ, with heap assertion Δ and executing in location l, e has type τ with an updated heap assertion Δ'."

TE-VAR simply requires the variable x to be defined in the type environment and does not have any effect on the heap assertion. Similarly, TE-NIL has no requirements as `()` immediately evaluates to a value of type (), again with no effect on the heap assertion.

In TE-SPAWN, we allow the spawn expression to execute in any location l, but its subexpression e must be able to execute on the GPU. Finally, the subexpression must evaluate to a reference because data can only move between the CPU and GPU through region transfers.

In both TE-PUSH and TE-PULL, we require the region in question to exist in some location, indicated by the $\Delta \star [\mathbf{r} \mapsto l]$ condition. Once the execution completes, the region is guaranteed to be on the GPU in the case of push and on the CPU in the case of pull. Note that these rules do not require the region to be in a particular location to start with. This model's Harlan's behavior, where regions are only moved if needed.

$$\textrm{TE-VAR}\quad \frac{\Gamma(x) = \tau}{\Gamma;\Delta;l \vdash x : \tau;\Delta}$$

$$\textrm{TE-NIL}\quad \frac{}{\Gamma;\Delta;l \vdash \texttt{()} : ();\Delta}$$

$$\textrm{TE-SPAWN}\quad \frac{\Gamma;\Delta;\texttt{GPU} \vdash e : (\text{ref } \mathbf{r}\ \tau);\Delta'}{\Gamma;\Delta;l \vdash \texttt{(spawn } e\texttt{)} : (\text{ref } \mathbf{r}\ \tau);\Delta'}$$

$$\textrm{TE-PUSH}\quad \frac{}{\Gamma;\Delta \star [\mathbf{r} \mapsto l];\texttt{CPU} \vdash \texttt{(push } \mathbf{r}\texttt{)} : ();\Delta \star [\mathbf{r} \mapsto \texttt{GPU}]}$$

$$\textrm{TE-PULL}\quad \frac{}{\Gamma;\Delta \star [\mathbf{r} \mapsto l];\texttt{CPU} \vdash \texttt{(pull } \mathbf{r}\texttt{)} : ();\Delta \star [\mathbf{r} \mapsto \texttt{CPU}]}$$

$$\textrm{TE-APP}\quad \frac{\Gamma;\Delta_1;l \vdash e_1 : \tau' \xrightarrow{\Delta;\Delta'}_{l'} \tau;\Delta_2 \qquad \Gamma;\Delta_2;l \vdash e_2 : \tau';\Delta_3 \star \Delta \qquad l' <: l}{\Gamma;\Delta_1;l \vdash \texttt{(}e_1\ \ e_2\texttt{)} : \tau;\Delta_3 \star \Delta'}$$

$$\textrm{TE-}\lambda\quad \frac{\Gamma[x:\tau'];\Delta';l' \vdash e : \tau;\Delta'' \qquad \Delta' = [\mathbf{r_1} \mapsto l_1] \star \ldots \star [\mathbf{r_n} \mapsto l_n] \qquad r_i \in \Gamma,\ \text{for } 1 \le i \le n}{\Gamma;\Delta;l \vdash \texttt{(lambda } l'\ \ \texttt{[}\mathbf{r_1} : l_1, \ldots, \mathbf{r_n} : l_2\texttt{]}\ \ \texttt{(}x\texttt{)}\ \ e\texttt{)} : \tau' \xrightarrow{\Delta';\Delta''}_{l'} \tau;\Delta}$$

$$\textrm{TE-LETREGION}\quad \frac{\Gamma,\mathbf{r};\Delta \star [\mathbf{r} \mapsto l];l \vdash e : \tau;\Delta' \star [\mathbf{r} \mapsto l'] \qquad \mathbf{r} \notin \text{frv}(\tau) \qquad \mathbf{r} \notin \Gamma}{\Gamma;\Delta;l \vdash \texttt{(let-region } \mathbf{r}\ \ e\texttt{)} : \tau;\Delta'}$$

$$\textrm{TE-REF}\quad \frac{\Gamma;\Delta;l \vdash e : \tau;\Delta' \star [\mathbf{r} \mapsto l] \qquad \mathbf{r} \in \Gamma}{\Gamma;\Delta;l \vdash \texttt{(ref } \mathbf{r}\ \ e\texttt{)} : (\text{ref } \mathbf{r}\ \tau);\Delta' \star [\mathbf{r} \mapsto l]}$$

$$\textrm{TE-DEREF}\quad \frac{\Gamma;\Delta;l \vdash e : (\text{ref } \mathbf{r}\ \tau);\Delta' \star [\mathbf{r} \mapsto l] \qquad \mathbf{r} \in \Gamma}{\Gamma;\Delta;l \vdash \texttt{(deref } \mathbf{r}\ \ e\texttt{)} : \tau;\Delta' \star [\mathbf{r} \mapsto l]}$$

FIGURE 6.9. Typing rules for expressions.

TE-APP is somewhat more complicated because we must take care with the evolution of the heap between the evaluation of the operator e_1 and the argument e_2. Furthermore, as we saw in the syntax of types in Figure 6.7, procedures carry their own region requirements which must be respected. As is standard, TE-APP requires that e_1 evaluate to a procedure and that e_2 evaluate to a value of the correct type for the procedure. Because each of these expressions may affect the location of regions, this rule enforces a left-to-right evaluation order. First, e_1 is evaluated under the conditions of Δ_1, producing an updated heap assertion, Δ_2. This heap assertion is used to evaluate e_2, which may make arbitrary changes to the heap provided it respects the requirements of the result of e_1. This is shown

$$\frac{}{\texttt{CPU} <: \texttt{CPU}} \qquad \frac{}{\texttt{GPU} <: \texttt{CPU}} \qquad \frac{}{\texttt{GPU} <: \texttt{GPU}}$$

FIGURE 6.10. Location compatibility rules. Intuitively, $l <: l'$ means that code that runs in location l can also run in location l'.

in the fact that e_2's updated heap is $\Delta_3 \star \Delta$. Finally, the procedure is applied to the argument under the conditions of $\Delta_3 \star \Delta$, producing a new heap assertion $\Delta_3 \star \Delta'$ that reflects the changes incurred by the procedure. Additionally, we require that the current execution location is compatible with the procedure. This is shown by the $l' <: l$ condition, whose definition is given in Figure 6.10.

TE-λ is similar to the λ rule for Simply Typed Lambda Calculus (STLC), but with some extra conditions due to the region annotations. In particular, we require the body of the procedure to run under a heap assertion constructed solely from the region location annotations on the `lambda` expression. In other words, the expression must make all of its region location assumptions explicit. For this reason, the closure type is somewhat more complicated, as it must capture a heap assertion that must hold on entry to the function, another that reflects the location of the regions when the function returns, and a location annotation that describes where a procedure may execute. For example, procedures that push or pull regions cannot execute on the GPU because region transfers may only be initiated from the CPU.

For TE-LETREGION, we need to make sure that the subexpression e can be typed with a new region $\mathbf{r}$ added to the type environment and heap assertion. We do not allow regions to be shadowed, hence the $\mathbf{e} \notin \Gamma$ requirement. Additionally, we need to ensure that region references do not escape their region,[1] so we require that $\mathbf{r}$ is not in the free region variables of the return type ($\mathbf{r} \notin \mathrm{frv}(\tau)$). See Figure 6.12 for the definition of frv.

TE-REF and TE-DEREF are similar. In the case of TE-REF, the argument must produce a value of type τ and then the full `(ref` $\mathbf{r}$ e`)` expression produces a value of type (ref $\mathbf{r}$ τ). Symmetrically, TE-DEREF requires the argument to produce an appropriate reference and

[1]This is not strictly necessary, as the dereference rule also requires the region to be in scope. It is safe for a reference to outlive its region provided that the program never dereferences it.

$$\text{TK-EMPTY}\quad \frac{}{\Gamma;\Delta;\Sigma;\texttt{CPU} \vdash \text{Empty} : \tau \to \tau}$$

$$\text{TK-UNSPAWN}\quad \frac{\Gamma;\Delta;\Sigma;l' \vdash k : (\text{ref}\ \mathbf{r}\ \tau') \to \tau \qquad l <: l'}{\Gamma;\Delta;\Sigma;l \vdash \text{Unspawn}(k,l') : (\text{ref}\ \mathbf{r}\ \tau') \to \tau}$$

$$\text{TK-RATOR}\quad \frac{\Gamma;\Delta''\star\Delta_2;\Sigma;l \vdash k : \tau'' \to \tau \qquad \Gamma;\Delta;l \vdash e : \tau';\Delta'\star\Delta_2}{\Gamma;\Delta;\Sigma;l \vdash \text{Rator}(e,k) : \left(\tau' \xrightarrow{\Delta';\Delta''}_{l} \tau''\right) \to \tau}$$

$$\text{TK-RAND}\quad \frac{\Sigma \vdash v : \tau' \xrightarrow{\Delta;\Delta'}_{l'} \tau'' \qquad \Gamma;\Delta_0\star\Delta';\Sigma;l \vdash k : \tau'' \to \tau \qquad l <: l'}{\Gamma;\Delta_0\star\Delta;\Sigma;l \vdash \text{Rand}(v,k) : \tau' \to \tau}$$

$$\text{TK-ALLOC}\quad \frac{\Gamma;\Delta\star[\mathbf{r}\mapsto l];\Sigma;l \vdash k : (\text{ref}\ \mathbf{r}\ \tau') \to \tau}{\Gamma;\Delta\star[\mathbf{r}\mapsto l];\Sigma;l \vdash \text{Alloc}(\mathbf{r},k) : \tau' \to \tau}$$

$$\text{TK-DEREF}\quad \frac{\Gamma;\Delta\star[\mathbf{r}\mapsto l];\Sigma;l \vdash k : \tau' \to \tau}{\Gamma;\Delta\star[\mathbf{r}\mapsto l];\Sigma;l \vdash \text{Deref}(\mathbf{r},k) : (\text{ref}\ \mathbf{r}\ \tau') \to \tau}$$

$$\text{TK-DEALLOC}\quad \frac{\Gamma;\Delta;\Sigma-\{\mathbf{r}\};l \vdash k : \tau' \to \tau \qquad \mathbf{r} \notin \text{frv}(\tau') \qquad \mathbf{r} \notin \Gamma}{\Gamma;\Delta\star[\mathbf{r}\mapsto l'];\Sigma;l \vdash \text{Dealloc}(\mathbf{r},k) : \tau' \to \tau}$$

$$\text{TK-RETURN}\quad \frac{\Sigma \vdash \Gamma' \sim \rho \qquad \Gamma';\Delta;\Sigma;l \vdash k : \tau' \to \tau}{\Gamma;\Delta;\Sigma;l \vdash \text{Return}(\rho,k) : \tau' \to \tau}$$

FIGURE 6.11. Continuation typing rules.

then the `(deref` **r** *e*`)` expression returns a value stripped of its reference type. In both cases, the region **r** is required to be in scope ($\mathbf{r} \in \Gamma$), and the region must be in the current location upon completion of the argument evaluation.

6.4.3. Continuation Typing Rules. The rules for typing continuations are given in Figure 6.11. These rules define a judgment $\Gamma;\Delta;\Sigma;l \vdash k : \tau_1 \to \tau_2$, which states that within a typing environment Γ, heap assertion Δ, store typing Σ and location l, the continuation k will, given a value of type τ_1, produce a value of type τ_2.[2]

We use Empty as the initial continuation, so in some ways Empty corresponds to the identity function. As such, it simply passes its argument on, producing a value of type τ when given a value of type τ. We require the location to be `CPU`, to enforce that Core Harlan programs begin and end execution on the CPU. These conditions are reflected in the TK-EMPTY rule.

[2]Modulo nontermination, of course.

The Unspawn continuation is used to possibly swap stores if necessary to signify that execution is moving from the GPU to CPU. As such, this continuation passes its argument through unchanged. The typing rule, TK-UNSPAWN requires the argument to be a reference in order to model the fact that Harlan only passes data between devices by reference.

For TK-RATOR and TK-RAND, recall how the evaluation of procedure applications proceeds. Because side effects such as moving regions are involved, we must encode the order of evaluation both in the transition function and the type rules. Given an expression $(e_1 \;\; e_2)$, the evaluator first evaluates e_1 while saving a Rator continuation on the stack (Table 6.3). The Rator continuation expects to receive a closure value and then proceeds evaluating the procedures argument, e_2. Finally, the Rand continuation saves the procedure and receives its argument evaluated to a value and then commences execution of the procedure (Table 6.4).

This order of evaluation imposes several constraints on the heap assertions used at each step. In TK-RATOR, we require the argument to the continuation to have a procedure type, $\tau' \xrightarrow{\Delta';\Delta''}_l \tau''$. Since the next step is to evaluate the saved argument expression, e, we require that we can type e using the heap assertion Δ. Upon return from evaluating e, we require $\Delta' \star \Delta_2$ to hold. This means that the heap assertion meets the requirements to apply the procedure, Δ', in addition to some other statements about the heap, Δ_2. Then, we require the saved continuation to have the correct type given $\Delta'' \star \Delta_2$, which reflects the changes made to the heap by applying the procedure combined with the portion of the heap that remained unchanged. This rule can be thought of has having a frame rule built in.

TK-RAND requires the saved procedure v to be a procedure type. Furthermore, the heap assertion must satisfy the procedure's requirements, Δ. The saved continuation k must then be well-typed under the changes to the heap introduced by applying the procedure, Δ', combined with the separate portion of the heap that does not concern the applied procedure, Δ_0. Finally, we require that the procedure is able to execute in the current location, indicated by $l <: l'$.

$$\begin{aligned}
\text{frv}(\texttt{()}) &= \{\} \\
\text{frv}((\text{ref } \mathbf{r}\ \tau)) &= \{\mathbf{r}\} \cup \text{frv}(\tau) \\
\text{frv}(\tau \xrightarrow{\Delta;\Delta'}_l \tau') &= \text{frv}(\tau) \cup \text{frv}(\tau') \cup \text{dom}(\Delta) \cup \text{dom}(\Delta')
\end{aligned}$$

$$\begin{aligned}
\text{other}(\texttt{CPU}) &= \texttt{GPU} \\
\text{other}(\texttt{GPU}) &= \texttt{CPU}
\end{aligned}$$

FIGURE 6.12. Auxiliary functions.

$$\text{TV-NIL}\ \frac{}{\Sigma \vdash \texttt{()} : ()} \qquad \text{TV-REF}\ \frac{\Sigma(\mathbf{r}, i) = \tau}{\Sigma \vdash \texttt{(label } i\texttt{)} : (\text{ref } \mathbf{r}\ \tau)} \qquad \text{TV-PROC}\ \frac{\Sigma \vdash \Gamma \sim \rho \qquad \Gamma[x : \tau']; \Delta; l \vdash e : \tau; \Delta'}{\Sigma \vdash \texttt{(closure } x\ \rho\ e\texttt{)} : \tau' \xrightarrow{\Delta;\Delta'}_l \tau}$$

FIGURE 6.13. Typing rules for values.

For TK-ALLOC, we require that the region being allocated from is in the current execution location and that the next continuation k accepts a reference type. Similarly, TK-DEREF requires the dereferenced region to be in the current execution location and that the next continuation accept a value of the type of the reference.

In TK-DEALLOC, we require that the region being deallocated—that is, destroyed—not appear in the type of the argument value in order to prevent escaping region references.

Finally, TK-RETURN requires us to supply a new typing environment Γ' that matches the saved environment for the stack frame being restored.

6.4.4. Value Typing Rules. Figure 6.13 defines the judgment $\Sigma \vdash v : \tau$, stating that a value v has type τ under the store typing environment Σ. For TV-NIL, the unit expression `()` always has type (). TV-REF states that a label has type (ref $\mathbf{r}$ i) if dereferencing the pointer in $\mathbf{r}$ yields a value of type τ. Finally, TV-PROC states that a closure has a function type $\tau' \xrightarrow{\Delta;\Delta'}_l \tau$ if we can find a Γ such that the body expression e has the correct type.

6.5. Type Safety

THEOREM 1 (Type Safety). *If $M : \tau$ then there exists an M' such that $M \rightsquigarrow M'$ and $M' : \tau$.*

We will devote the rest of this section to proving this theorem. The proof proceeds by induction on the structure of the typing judgment, $M : \tau$. The proof is broken down into three main sections, according to the three rules from the machine typing judgment (Figure 6.8). These rules are TM-VALUE, TM-EXPR and TM-CONT. The TM-VALUE case is small and straightforward, but the other two cases are further broken down according to the expression and continuation typing rules (Figures 6.9 and 6.11).

6.5.1. Case TM-VALUE. In this case we have $M : \tau$ because of the TM-VALUE rule. This means $M = \langle v, s_1, s_2 \rangle$ for some v, s_1 and s_2. Furthermore, we know from TM-VALUE that there is some Σ such that $\Sigma \vdash s_1, s_2$ and $\Sigma \vdash v : \tau$.

Using these facts, we must find a new machine, M', such that $M \rightsquigarrow M'$ and $M' : \tau$. The transition function states that $\langle v, s_1, s_2 \rangle \rightsquigarrow \langle v, s_1, s_2 \rangle$, so we will let $M' = M$. Because the machine state has not changed, we can conclude $M' : \tau$ trivially from the induction hypothesis.

6.5.2. Case TM-EXPR. In this case, we have $M = \langle e, \rho, s_1, s_2, k, l \rangle$. From the TM-EXPR rule, we know the following:

$$\Sigma \vdash \Gamma \sim \rho \tag{2}$$

$$\Gamma; \Delta; l \vdash e : \tau'; \Delta' \tag{3}$$

$$\Sigma \vdash s_1, s_2 \tag{4}$$

$$\Delta; l \vdash s_1, s_2 \tag{5}$$

$$\Gamma; \Delta'; \Sigma; l \vdash k : \tau' \rightarrow \tau \tag{6}$$

$$\langle e, \rho, s_1, s_2, k, l \rangle : \tau \tag{7}$$

We will further break this case down by expression types, giving us ten cases to consider–one for each rule in Figure 6.9. In each case, we need show how to find an M' such that $\langle e, \rho, s_1, s_2, k, l \rangle \rightsquigarrow M'$ and $M' : \tau$.

6.5.2.1. TE-VAR. In this case, we have $e = x$ for some x. From the TE-VAR rule, we also know the following:

(8) $$\Gamma(x) = \tau'$$

The TE-VAR rule further refines (3) to

(9) $$\Gamma;\Delta;l \vdash x : \tau';\Delta$$

Importantly, we see that $\Delta' = \Delta$. In other words, evaluating a variable reference does not affect the location of any regions.

In order to apply the step function and not step to an error configuration, we need to show that $\rho(x)$ is defined. We know from (2) that $\rho(x)$ is defined precisely when $\Gamma(x)$ is defined. We know $\Gamma(x)$ is defined from (8). Let $v = \rho(x)$, then from the step function we have:

(10) $$\langle e,\rho,s_1,s_2,k,l\rangle \rightsquigarrow \langle k,v,\rho,s_1,s_2,l\rangle$$

Now we need to show $\langle k,v,\rho,s_1,s_2,l\rangle : \tau'$. To apply TM-CONT, we have several obligations to meet:

- $\Sigma \vdash \Gamma \sim \rho$, which is just (2)
- $\Delta;l \vdash s_1,s_2$, which is (5)
- $\Gamma;\Delta';\Sigma;l \vdash k : \tau' \to \tau$, which is (6)
- $\Sigma \vdash v : \tau'$

We conclude $\Sigma \vdash v : \tau'$ using (2) and (8). Thus, we can use TM-CONT to conclude:

$$\langle k,v,\rho,s_1,s_2,l\rangle : \tau'$$

6.5.2.2. TE-NIL. From the TE-NIL rule, we know that $e = ()$ and $\tau' = ()$. The step function gives us $M' = \langle k, (),\rho,s_1,s_2,l\rangle$. We now need to use TM-CONT to show $M' : \tau'$, that is, $\langle k, (),\rho,s_1,s_2,l\rangle : ()$.

As in the TE-VAR case, most of the conditions to use TM-CONT are exactly contained within the induction hypotheses. However, we still must show $\Sigma \vdash \texttt{()} : ()$, which follows straightforwardly from TV-NIL. Thus, we use TM-CONT to conclude $M' : \tau'$.

6.5.2.3. TE-SPAWN. This case has $e = \texttt{(spawn } e'\texttt{)}$. The TE-SPAWN rule tells us that $\tau' = (\text{ref } \mathbf{r}\ \tau'')$ and allows us to refine the induction hypothesis as follows.

$$\Gamma;\Delta;l \vdash \texttt{(spawn } e'\texttt{)} : (\text{ref } \mathbf{r}\ \tau'');\Delta' \tag{11}$$

$$\Gamma;\Delta;\texttt{GPU} \vdash e' : (\text{ref } \mathbf{r}\ \tau'');\Delta' \tag{12}$$

There are two possible cases of the step function to apply, depending on whether $l = \texttt{CPU}$ or $l = \texttt{GPU}$. We will consider these two separately.

Case $l = \texttt{CPU}$. Using the step function, we have $M' = \langle e', \rho, s_2, s_1, \text{Unspawn}(k, \texttt{CPU}), \texttt{GPU}\rangle$. It is important to notice that the two stores have swapped and we have moved from executing on the CPU to the GPU. We must now attempt to use TE-EXPR to show $\langle e', \rho, s_2, s_1, \text{Unspawn}(k, \texttt{CPU}), \texttt{GP}$ τ. Most of the prerequisites are exactly among the induction hypotheses, but we still must show $\Sigma \vdash s_2, s_1$, $\Delta;\texttt{GPU} \vdash s_2, s_1$ and $\Gamma;\Delta';\Sigma;\texttt{GPU} \vdash \text{Unspawn}(k, \texttt{CPU}) : \tau' \to \tau$.

We show $\Sigma \vdash s_2, s_1$ using Lemma 1. Similarly, we use Lemma 2 to show $\Delta;\texttt{GPU} \vdash s_2, s_1$.

To show $\Gamma;\Delta';\Sigma;\texttt{GPU} \vdash \text{Unspawn}(k, \texttt{CPU}) : \tau' \to \tau$ we need to use TK-UNSPAWN, which has the further obligation of $\Gamma;\Delta';\Sigma;\texttt{CPU} \vdash k : (\text{ref } \mathbf{r}\ \tau'') \to \tau$. This obligation is simply (6), since $\tau' = (\text{ref } \mathbf{r}\ \tau'')$ and $l = \texttt{CPU}$.

Case $l = \texttt{CPU}$. Here, the step function gives us $M' = \langle e', \rho, s_1, s_2, \text{Unspawn}(k, \texttt{GPU}), \texttt{GPU}\rangle$. We need to show $\langle e', \rho, s_1, s_2, \text{Unspawn}(k, \texttt{GPU}), \texttt{GPU}\rangle : \tau$ by using TM-EXPR. The prerequisites for TM-EXPR are among the induction hypotheses, except for

$$\Gamma;\Delta';\Sigma;\texttt{GPU} \vdash \text{Unspawn}(k, \texttt{GPU}) : \tau' \to \tau$$

which follows directly from TK-UNSPAWN and (6).

6.5.2.4. TE-PUSH. The TE-PUSH rule additionally gives us the following refinement of (3), which is to say that $e = \texttt{(push } \mathbf{r}\texttt{)}$, $\Delta = \Delta'' \star [\mathbf{r} \mapsto l']$ and $\Delta' = \Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}]$ for some

Δ''.

$$\Gamma;\Delta'' \star [\mathbf{r} \mapsto l'];\texttt{CPU} \vdash \texttt{(push } \mathbf{r}\texttt{)} : ();\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] \tag{13}$$

Furthermore, we see that $\tau' = ()$ and that $l = \texttt{CPU}$.

Now there are three subcases to consider: when $\mathbf{r} \in \text{dom}(s_1)$, when $\mathbf{r} \notin \text{dom}(s_1)$ but $\mathbf{r} \in \text{dom}(s_2)$, and when $\mathbf{r}$ is in neither $\text{dom}(s_1)$ nor $\text{dom}(s_2)$.

Case $\mathbf{r} \in \text{dom}(s_1)$. From the step function, we have $M' = \langle k, \texttt{()}, \rho, s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r}), \texttt{CPU}\rangle$.

We want to use TM-CONT to show $M' : \tau$. We will use the same store typing, type environment and runtime environments: Σ, Γ, and ρ. TM-CONT also needs a heap assertion, which we will take to be the result of evaluating `(push r)`, which is $\Delta' = \Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}]$. Applying TM-CONT gives us the following proof obligations.

- $\Sigma \vdash \Gamma \sim \rho$
- $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}];\texttt{CPU} \vdash s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r})$
- $\Sigma \vdash s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r})$
- $\Gamma;\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}];\Sigma;\texttt{CPU} \vdash k : () \to \tau$
- $\Sigma \vdash v : ()$

The first obligation is just (2). The fourth obligation is just (6) with the appropriate substitutions applied. The fifth obligation follows directly from TV-NIL. The second through fourth obligations essentially amount to showing (5) and (4) still hold under the new heap assertion.

Recall from Definition 1 that in order to show $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}];\texttt{CPU} \vdash s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r})$ we must show two things:

(a) If there exists a Δ''' such that $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] = \Delta''' \star [\mathbf{r}' \mapsto \texttt{CPU}]$ then $\mathbf{r}' \in \text{dom}(s_1 - \{\mathbf{r}\})$.

(b) If there exists a Δ''' such that $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] = \Delta''' \star [\mathbf{r}' \mapsto \texttt{GPU}]$ then $\mathbf{r}' \in \text{dom}(s_2 :: s_1(\mathbf{r}))$.

For Item (a), consider whether $\mathbf{r} = \mathbf{r}'$. If $\mathbf{r} = \mathbf{r}'$ then we have a contradiction, since the heap assertion maps $\mathbf{r}$ to both $\texttt{CPU}$and $\texttt{GPU}$. This cannot happen without violating the disjointness requirement for $\star$. Now consider if $\mathbf{r} \neq \mathbf{r}'$. We know from (5) that $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}]; l \vdash s_1, s_2$, which means if there exists a $\Delta^{(4)}$ such that $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] = \Delta^{(4)} \star [\mathbf{r}' \mapsto \texttt{CPU}]$ that $\mathbf{r}' \in \mathrm{dom}(s_1)$. We can use Δ''' as the $\Delta^{(4)}$ we are looking for, so therefore $\mathbf{r}' \in \mathrm{dom}(s_1)$.

For Item (b), we will also consider the cases for $\mathbf{r} = \mathbf{r}'$ and $\mathbf{r} \neq \mathbf{r}'$ separately. In either case, we know that $\mathrm{dom}(s_2 :: s_1(\mathbf{r})) = \{\mathbf{r}\} \cup \mathrm{dom}(s_2)$. If $\mathbf{r} = \mathbf{r}'$ then it is obvious that $\mathbf{r} \in \{\mathbf{r}\} \cup \mathrm{dom}(s_2)$. Now consider when $\mathbf{r} \neq \mathbf{r}'$. Using (5), we know that if there exists a $\Delta^{(4)}$ such at $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] = \Delta^{(4)} \star [\mathbf{r}' \mapsto \texttt{GPU}]$ then $\mathbf{r}' \in \mathrm{dom}(s_2)$. Using Lemma 4, we know that since $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] = \Delta''' \star [\mathbf{r}' \mapsto \texttt{GPU}]$ then there is some $\Delta^{(5)}$ such that $\Delta''' = \Delta^{(5)} \star [\mathbf{r} \mapsto \texttt{GPU}]$.

Choose $\Delta^{(4)} = \Delta^{(5)} \star [\mathbf{r} \mapsto \texttt{GPU}]$. We need to show:

$$\Delta'' \star [\mathbf{r} \mapsto \texttt{CPU}] = \Delta^{(5)} \star [\mathbf{r} \mapsto \texttt{CPU}] \star [\mathbf{r}' \mapsto \texttt{GPU}]$$

We know $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}] = \Delta^{(4)} \star [\mathbf{r} \mapsto \texttt{GPU}] \star [\mathbf{r}' \mapsto \texttt{GPU}]$. We can replace $[\mathbf{r} \mapsto \texttt{GPU}]$ on both sides of the equation with $[\mathbf{r} \mapsto \texttt{CPU}]$, giving the result we wanted. Therefore, we conclude that $\mathbf{r}' \in \mathrm{dom}(s_2)$ and therefore in $\mathrm{dom}(s_2 :: s_1(\mathbf{r}) ::)$.

To show $\Sigma \vdash s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r})$, we observe that Definition 2 holds as long as either store maps each $(\mathbf{r}, i)$ pair to a value of the correct type. In other words, the typing is independent of the values' locations and thus moving a region from one store to another does not affect the typing. Thus, $\Sigma \vdash s_1 - \{\mathbf{r}\}, s_2 :: s_1(\mathbf{r})$ holds.

Having met these proof obligations, we conclude for this subcase that $M' : \tau$.

Case $\mathbf{r} \notin \mathrm{dom}(s_1)$ *but* $\mathbf{r} \in \mathrm{dom}(s_2)$. In this case, M steps to $M' = \langle k, \texttt{()}, \rho, s_1, s_2, \texttt{CPU} \rangle$. We want to show this has type τ using TM-CONT. Since the step function left the stores unchanged, we will similarly leave the heap assertion, store typing and type environment the same.

Once again, TM-CONT gives us the following obligations:

- $\Sigma \vdash \Gamma \sim \rho$
- $\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}]; \texttt{CPU} \vdash s_1, s_2$
- $\Sigma \vdash s_1, s_2$

- $\Gamma;\Delta'' \star [\mathbf{r} \mapsto \texttt{GPU}];\Sigma;\texttt{CPU} \vdash k : () \to \tau$
- $\Sigma \vdash v : ()$

These obligations follow respectively from (2), (5), (4), (6) and TV-NIL. Thus we conclude $M' : \tau$.

Case $\mathbf{r} \notin \mathrm{dom}(s_1)$ *and* $\mathbf{r} \notin \mathrm{dom}(s_2)$. In this case the step function would give us $\perp$, which cannot be typed. Fortunately, we shall find a contradiction between this case's assumptions and the induction hypothesis.

Using (5), we know that if $l' = \texttt{CPU}$ then $\mathbf{r} \in \mathrm{dom}(s_1)$ and if $l' = \texttt{GPU}$ then $\mathbf{r} \in \mathrm{dom}(s_2)$. In both cases, we contradict our assumption that $\mathbf{r} \notin \mathrm{dom}(s_1)$ and $\mathbf{r} \notin \mathrm{dom}(s_2)$, which means this case is impossible.

6.5.2.5. TE-PULL. This case is analogous to the TE-PUSH case.

6.5.2.6. TE-APP. The TE-APP rule refines (3) to $\Gamma;\Delta;l \vdash \texttt{(}e_1\ \ e_2\texttt{)} : \tau';\Delta_3 \star \Delta^{(2)}$. Furthermore, we gain the following new pieces of information.

$$\Gamma;\Delta;l \vdash e_1 : \tau'' \xrightarrow{\Delta^{(1)};\Delta^{(2)}}_l \tau';\Delta_2 \tag{14}$$

$$\Gamma;\Delta_2;l \vdash e_2 : \tau'';\Delta_3 \star \Delta^{(1)} \tag{15}$$

Applying the step function to M gives us $M' = \langle e_1, \rho, s_1, s_2, \mathrm{Rator}(e_2,k), l\rangle$. In order to apply TM-EXPR, we will first show the following:

- $\Sigma \vdash \Gamma \sim \rho$
- $\Gamma;\Delta;l \vdash e_1 : \tau'';\Delta_3 \star \Delta^{(1)}$
- $\Sigma \vdash s_1, s_2$
- $\Delta;l \vdash s_1, s_2$
- $\Gamma;\Delta_3 \star \Delta^{(1)};\Sigma;l \vdash \mathrm{Rator}(e_2,k) : \tau'' \to \tau$

The first four are (2), (14), (4) and (5), respectively.

To show the final obligation, we use TK-RATOR, which gives us two new obligations:

- $\Gamma;\Delta_3 \star \Delta^{(1)};\Sigma;l \vdash k : \tau' \to \tau$
- $\Gamma;\Delta_2;l \vdash e_2 : \tau'';\Delta_3 \star \Delta^{(1)}$

These are just (6) and (15).

Thus we have met all our obligations and therefore conclude $M' : \tau$.

6.5.2.7. TE-λ. We have the following from TE-λ.

$$e = \texttt{(lambda}\ l'\ \ \texttt{[}\mathbf{r_1} : l_1 \ldots \mathbf{r_n} : l_n\texttt{]}\ \ \texttt{(}x\texttt{)}\ \ e'\texttt{)} \tag{16}$$

$$\Delta' = \Delta \tag{17}$$

$$\Delta'' = [\mathbf{r_1} \mapsto l_1] \star \ldots \star [\mathbf{r_n} \mapsto l_n] \tag{18}$$

$$\Gamma[x : \tau_1]; m; \Delta'' \vdash l' : e'; \tau_2 \Delta''' \tag{19}$$

$$\tau' = \tau_1 \xrightarrow{\Delta'';\Delta'''}_{l'} \tau_2 \tag{20}$$

The step function gives $M' = \langle k, \texttt{(closure}\ x\ \ \rho\ \ e'\texttt{)}, \rho, s_1, s_2, l\rangle$.

We want to show $M' : \tau$ using TM-CONT. Most of the prerequisites for TM-CONT are in the induction hypotheses, but we must still show that the value being passed to the continuation has the right type. That is:

$$\Sigma \vdash \texttt{(closure}\ x\ \ \rho\ \ e'\texttt{)} : \tau'$$

We can do this using TV-PROC, which requires showing $\Sigma \vdash \Gamma \sim \rho$, which is just (2), and $\Gamma[x : \tau_1]; \Delta''; l' \vdash e' : \tau_2; \Delta'''$, which is just (19). Therefore, we conclude $M' : \tau$.

6.5.2.8. TE-LETREGION. From TE-LETREGION, we have:

$$e = \texttt{(let-region}\ \mathbf{r}\ \ e'\texttt{)} \tag{21}$$

$$\Gamma; \Delta \star [\mathbf{r} \mapsto l]; l \vdash e' : \tau'; \Delta' \star [\mathbf{r} \mapsto l'] \tag{22}$$

$$\mathbf{r} \notin \mathrm{frv}(\tau') \tag{23}$$

$$\mathbf{r} \notin \Gamma \tag{24}$$

The step function gives us the following next state for the machine.

$$M' = \langle e', \rho, [r : \cdot] :: s_1, s_2, \mathrm{Dealloc}(\mathbf{r}, k), l\rangle$$

To show this has the correct type, we must use TM-EXPR, which gives us the following proof obligations:

- $\Sigma \vdash \Gamma \sim \rho$
- $\Gamma; \Delta \star [\mathbf{r} \mapsto l]; l \vdash e' : \tau'; \Delta \star [\mathbf{r} \mapsto l']$
- $\Sigma \vdash [r : \cdot] :: s_1, s_2$
- $\Delta \star [\mathbf{r} \mapsto l]; l \vdash [r : \cdot] :: s_1, s_2$
- $\Gamma; \Gamma' \star [\mathbf{r} \mapsto l']; \Sigma; l \vdash \text{Dealloc}(\mathbf{r}, k) : \tau' \to \tau$

The first obligation is just (2). The second obligation is (22). The remaining obligations take a little more care.

To show $\Sigma \vdash [r : \cdot] :: s_1, s_2$, recall from Definition 2 that we need to show $\forall (\mathbf{r}, i) \in \text{dom}(\Sigma)$, $\Sigma \vdash s_1(\mathbf{r}, i) : \Sigma(\mathbf{r}i)$ or $\Sigma \vdash s_2(\mathbf{r}, i) : \Sigma(\mathbf{r}i)$. Because we have just extended s_1 with an empty region and we knew that $\mathbf{r} \notin \text{dom}(\Sigma)$, we that $\Sigma \vdash [r : \cdot] :: s_1, s_2$ still holds.

$\Delta \star [\mathbf{r} \mapsto l]; l \vdash [r : \cdot] :: s_1, s_2$ holds because we have $\mathbf{r} \in \text{dom}([r : \cdot] :: s_1)$ and the rest holds from (5).

Finally, we must show $\Gamma; \Gamma' \star [\mathbf{r} \mapsto l']; \Sigma; l \vdash \text{Dealloc}(\mathbf{r}, k) : \tau' \to \tau$ using TK-DEALLOC. This rule has the following obligations:

- $\Gamma; \Delta' \star [\mathbf{r} \mapsto l']; \Sigma - \{\mathbf{r}\}; l \vdash k : \tau' \to \tau$
- $\mathbf{r} \notin \text{frv}(\tau')$
- $\mathbf{r} \notin \Gamma$

For the first obligation, we already know (6.5.2.8), and furthermore because $\mathbf{r} \notin \text{dom}(\Sigma)$, we know that $\Sigma - \{\mathbf{r}\} = \Sigma$. Therefore, we conclude $\Gamma; \Gamma' \star [\mathbf{r} \mapsto l']; \Sigma - \{\mathbf{r}\}; l \vdash k : \tau' \to \tau$. The remaining obligations are among the induction hypotheses, so we conclude k has the appropriate type and go on to conclude $M' : \tau$.

6.5.2.9. TE-REF. From TE-REF, we know:

(25) $$e = (\texttt{ref}\ \ \mathbf{r}\ \ e')$$

(26) $$\tau' = (\text{ref}\ \mathbf{r}\ \tau'')$$

(27) $$\Delta' = \Delta'' \star [\mathbf{r} \mapsto l]$$

(28) $$\Gamma; \Delta; l \vdash e' : \tau''; \Delta'' \star [\mathbf{r} \mapsto l]$$

The step function yields the following.

$$M' = \langle e', \rho, s_1, s_2, \text{Alloc}(\mathbf{r}, k), l \rangle$$

As usual, we want to use TM-EXPR to show $M' : \tau$. Most of the prerequisites are included in the induction hypotheses. However, we must use TK-ALLOC to show $\Gamma; \Delta'; \Sigma; l \vdash \text{Alloc}(\mathbf{r}, k) : \tau'' \to \tau$.

TK-ALLOC requires us to show $\Gamma; \Delta'; \Sigma; l \vdash k : (\text{ref } \mathbf{r}\ \tau'') \to \tau$, which is just (6) with the appropriate substitutions applied. Therefore, we conclude $M' : \tau$.

6.5.2.10. TE-DEREF. This case proceeds analogously to the TE-REF case. From TE-DEREF, we have:

(29) $$e = (\texttt{deref}\ \ \mathbf{r}\ \ e')$$

(30) $$\Delta' = \Delta'' \star [\mathbf{r} \mapsto l]$$

(31) $$\Gamma; \Delta; l \vdash e' : (\text{ref } \mathbf{r}\ \tau''); \Delta'' \star [\mathbf{r} \mapsto l]$$

The step function yields $M' = \langle e', \rho, s_1, s_2, \text{Deref}(\mathbf{r}, k), l \rangle$. To show $M' : \tau$, we use TM-EXPR. Most of the prerequisites for TM-EXPR are among the induction hypotheses, but we must still show the following.

$$\Gamma; \Delta'' \star [\mathbf{r} \mapsto l]; \Sigma; l \vdash \text{Deref}(\mathbf{r}, k) : (\text{ref } \mathbf{r}\ \tau'') \to \tau$$

As before, this is just (6) with the appropriate substitutions applied. Therefore, $M' : \tau$.

6.5.3. Case TM-CONT. In these subcases, we have $M : \tau$ by TM-CONT, which means we also gain the following pieces of information:

(32) $$\Sigma \vdash s_1, s_2$$

(33) $$\Sigma \vdash \Gamma \sim \rho$$

(34) $$\Delta; l \vdash s_1, s_2$$

(35) $$\Gamma; \Delta; \Sigma; l \vdash k : \tau' \to \tau$$

(36) $$\Sigma \vdash v : \tau'$$

We will continue by inversion on (35), considering each of the rules from Figure 6.11.

6.5.3.1. TK-EMPTY. From TK-EMPTY, we have $l = \texttt{CPU}$, $k = \text{Empty}$ and $\tau' = \tau$. The step function yields $M' = \langle v, s_1, s_2 \rangle$. We can directly apply TM-VALUE using (32) and (36) to show $M' : \tau$.

6.5.3.2. TK-UNSPAWN. We know the following from TK-UNSPAWN.

$$k = \text{Unspawn}(k', l') \tag{37}$$

$$\tau' = (\text{ref}\ \mathbf{r}\ \tau'') \tag{38}$$

$$\Gamma; \Delta; \Sigma; l' \vdash k' : (\text{ref}\ \mathbf{r}\ \tau'') \to \tau \tag{39}$$

$$l <: l' \tag{40}$$

Since v is of reference type, we also know there is some i such that $v = \texttt{(label}\ i\texttt{)}$.

We have to consider $l' = \texttt{GPU}$ and $l' = \texttt{CPU}$ separately. When $l' = \texttt{GPU}$ we also know that $l = \texttt{GPU}$ because $l <: l'$. The step function gives us $M' = \langle k, \texttt{(label}\ i\texttt{)}, \rho, s_1, s_2, l' \rangle$. Using TM-CONT, we conclude $M' : \tau$.

When $l' = \texttt{CPU}$, we still know that $l = \texttt{GPU}$ because there is no way to spawn an expression onto the CPU. In Table 6.3, we see that the only two right hand sides in which an Unspawn continuation appear on the right hand side also have the execution location set to $\texttt{GPU}$. Since $l = \texttt{GPU}$, we match a case of the transition function in Table 6.4 and get $M' = \langle k, \texttt{(label}\ i\texttt{)}, \rho, s_2, s_1, l \rangle$. Notice that the stores have been swapped because we have transitioned from executing on the GPU to the CPU.

We want to show $M' : \tau$ using TM-CONT, which requires showing the following.

- $\Sigma \vdash s_2, s_1$
- $\Delta; l' \vdash s_2, s_1$
- $\Gamma; \Delta; \Sigma; l' \vdash k : \tau' \to \tau$
- $\Sigma \vdash v : \tau'$

We show the first obligation using Lemma 1 with (32) and similarly we show the second obligation with Lemma 2 with (34). The last two are just (39) and (36), respectively.

6.5.3.3. TK-RATOR. We gain the following new information from TK-RATOR:

(41) $$\tau' = \tau_1 \xrightarrow{\Delta_1;\Delta_2}_l \tau_2$$

(42) $$k = \text{Rator}(e, k')$$

(43) $$\Gamma;\Delta;l \vdash e : \tau_1;\Delta'$$

(44) $$\Gamma;\Delta' \star \Delta_2;\Sigma;l \vdash k' : \tau_2 \to \tau$$

The step function yields $M' = \langle e, \rho, s_1, s_2, \text{Rand}(v, k'), l\rangle$. To show $M' : \tau$, we will use TM-EXPR. Most of TM-EXPR's requirements are already among the induction hypotheses, but we still must show that $\text{Rand}(v, k')$ has the correct type:

$$\Gamma;\Delta';\Sigma;l \vdash \text{Rand}(v, k') : \tau_1 \to \tau$$

We will show this using TK-RAND, but to do this we must show:

- $\Sigma \vdash v : \tau_1 \xrightarrow{\Delta_1;\Delta_2}_l \tau_2$
- $\Gamma;\Delta_0 \star \Delta_2;\Sigma;l \vdash k' : \tau_2 \to \tau$, for some Δ_0.

The first obligation is just (36). We can choose $\Delta_0 = \Delta'$, which means the second obligation is just (44).

Thus, we conclude $M' : \tau$.

6.5.3.4. TK-RAND. From TK-RAND, we know:

(45) $$\Delta' = \Delta_0 \star \Delta_1$$

(46) $$k = \text{Rand}(v', k')$$

(47) $$\Sigma \vdash v' : \tau' \xrightarrow{\Delta_1;\Delta_2}_{l'} \tau''$$

(48) $$l' <: l$$

(49) $$\Gamma;\Delta_0 \star \Delta_2;\Sigma;l \vdash k' : \tau'' \to \tau$$

Since v' has a closure type, we know that $v' =$ `(closure` $x\ \ \rho'\ \ e$`)` for some x, ρ' and e.

Thus, by the step function, we have $M' = \langle e, [x : v]\rho', s_1, s_2, \text{Return}(\rho, k'), l\rangle$.

We want to show $M' : \tau$ using TM-EXPR, but we will have to establish a few things before we can do this. We know that v' is well typed and must be well typed by TV-PROC. Therefore we know that there is some Γ' such that the following hold.

$$\Sigma \vdash \Gamma' \sim \rho' \tag{50}$$

$$\Gamma'[x:\tau];\Delta_1;l \vdash e : \tau'';\Delta_2 \tag{51}$$

In order to show $M' : \tau$ using TM-EXPR, we must establish the following.

(a) $\Sigma \vdash \Gamma'[x:\tau'] \sim [x:v]\rho'$

(b) $\Gamma'[x:\tau];\Delta_0 \star \Delta_1;l \vdash e : \tau'';\Delta_0 \star \Delta_2$

(c) $\Sigma \vdash s_1, s_2$

(d) $\Delta_0 \star \Delta_1;l \vdash s_1, s_2$

(e) $\Gamma';\Delta_0 \star \Delta_2;\Sigma;l \vdash \text{Return}(\rho,k') : \tau'' \to \tau$

A couple of these are obvious. Item (c) is just (32). Item (d) is (34), since $\Delta' = \Delta_0 \star \Delta_1$.

For Item (a), it's clear from (36) that $\Sigma \vdash \Gamma'(x) : \rho(x)$. We know the remainder, $\Sigma \vdash \Gamma' \sim \rho'$, from (50).

We can show Item (b) using (51) and Lemma 3.

Finally, we show Item (e) using TK-RETURN with (33) and (49).

Therefore, we have established $M' : \tau$

6.5.3.5. TK-ALLOC. From TK-ALLOC, we know the following:

$$\Delta = \Delta' \star [\mathbf{r} \mapsto l] \tag{52}$$

$$k = \text{Alloc}(\mathbf{r},k') \tag{53}$$

$$\Gamma;\Delta' \star [\mathbf{r} \mapsto l];\Sigma;l \vdash k' : (\text{ref } \mathbf{r}\ \tau') \to \tau \tag{54}$$

Because of (52) and the fact that $\Delta;l \vdash s_1, s_2$, we know that $\mathbf{r} \in \text{dom}(s_1)$, which means we can apply the step function to get:

$$M' = \langle k', \texttt{(label } i\texttt{)}, \rho, s_1[r := s_1(r) :: v], s_2, l\rangle \tag{55}$$

$$i = \text{length}(s_1(\mathbf{r})) \tag{56}$$

We now want to show $M' : \tau$ using TM-CONT. Let $\Sigma' = \Sigma :: [(\mathbf{r}, i) : \tau']$. We have the following obligations in order to use TM-CONT:

(a) $\Sigma' \vdash s_1[r := s_1(r) :: v], s_2$

(b) $\Sigma' \vdash \Gamma \sim \rho$

(c) $\Delta' \star [\mathbf{r} \mapsto l]; l \vdash s_1[r := s_1(r) :: v], s_2$

(d) $\Gamma; \Delta' \star [\mathbf{r} \mapsto l]; \Sigma'; l \vdash k' : (\text{ref } \mathbf{r}\ \tau') \to \tau$

(e) $\Sigma' \vdash \texttt{(label } i\texttt{)} : (\text{ref } \mathbf{r}\ \tau')$

For Item (a), we have chosen i such that $s_1[r := s_1(r) :: v](i) = v$ and we know from (36) that $\Sigma \vdash v : \tau'$. Since we have only added new information to go from Σ to Σ', we know that $\Sigma' \vdash v : \tau'$ holds as well. Therefore, we can also conclude $\Sigma' \vdash s_1[r := s_1(r) :: v], s_2$.

Item (b) holds by a similar argument. Recall Definition 3. The domain of Σ' is a superset of the domain of Σ, so for every $x \in \text{dom}(\rho)$ we still have $\Sigma' \vdash \rho(x) : \Gamma(x)$.

Item (c) holds because although we have updated the primary store, we have not changed the location of any region and therefore the heap assertion is still in tact.

Item (d) holds by a similar argument to Item (b).

Item (e) is shown by a straightforward application of TV-REF.

Therefore, we conclude $M' : \tau$.

6.5.3.6. TK-DEREF. From TK-DEREF, we have:

$$k = \text{Deref}(\mathbf{r}, k') \tag{57}$$

$$\tau' = (\text{ref } \mathbf{r}\ \tau'') \tag{58}$$

$$\Delta = \Delta' \star [\mathbf{r} \mapsto l] \tag{59}$$

$$\Gamma; \Delta' \star [\mathbf{r} \mapsto l]; \Sigma; l \vdash k' : \tau'' \to \tau \tag{60}$$

Since $\Sigma \vdash v : (\text{ref } \mathbf{r}\ \tau'')$, we know from TV-REF that $v = \texttt{(label } i\texttt{)}$ for some i and that $\Sigma(\mathbf{r}, i) = \tau''$.

We can conclude from (59) and (34) that $\mathbf{r} \in \text{dom}(s_1)$ and furthermore we know that $i < \text{length}(s_1(\mathbf{r})$ because $\texttt{(label } i\texttt{)}$ has type $(\text{ref } \mathbf{r}\ \tau'')$ in Σ. Using these facts, we know the

step function will give us:

$$M' = \langle k', s_1(\mathbf{r}, i), \rho, s_1, s_2, l \rangle$$

At this point, we also have all the information we need for a straightforward application of TM-CONT to conclude $M' : \tau$.

6.5.3.7. TK-DEALLOC. In this case, we know the following.

$$k = \text{Dealloc}(\mathbf{r}, k') \tag{61}$$

$$\Delta = \Delta' \star [\mathbf{r} \mapsto l'] \tag{62}$$

$$\mathbf{r} \notin \text{frv}(\tau') \tag{63}$$

$$\mathbf{r} \notin \Gamma \tag{64}$$

$$\Gamma; \Delta'; \Sigma - \{\mathbf{r}\}; l \vdash k' : \tau' \to \tau \tag{65}$$

The step function yields:

$$M' = \langle k', v, \rho, s_1 - \{\mathbf{r}\}, s_2 - \{\mathbf{r}\}, l \rangle$$

As usual, we want to use TM-CONT to show $M' : \tau$ which obliges us to show the following.

(a) $\Sigma - \{\mathbf{r}\} \vdash s_1 - \{\mathbf{r}\}, s_2 - \{\mathbf{r}\}$
(b) $\Sigma - \{\mathbf{r}\} \vdash \Gamma \sim \rho$
(c) $\Delta'; l \vdash s_1 - \{\mathbf{r}\}, s_2 - \{\mathbf{r}\}$
(d) $\Gamma; \Delta'; \Sigma - \{\mathbf{r}\}; l \vdash k' : \tau' \to \tau$
(e) $\Sigma - \{\mathbf{r}\} \vdash v : \tau'$

To show Item (a), notice that we already have $\Sigma \vdash s_1, s_2$. In this case, we have just removed $\mathbf{r}$ from each of the pieces of the judgment, so therefore Item (a) will still hold.

For Item (b), we know from (64) that $\mathbf{r} \notin \Gamma$, meaning nothing in this typing judgment refers to $\mathbf{r}$. Therefore, removing $\mathbf{r}$ from Σ will not affect the judgment.

For Item (c), we know from (62) that $\mathbf{r} \notin \Delta'$. Since we also have removed $\mathbf{r}$ from s_1 and s_2, then we can conclude $\Delta'; l \vdash s_1 - \{\mathbf{r}\}, s_2 - \{\mathbf{r}\}$.

Finally, Item (d) is just (65) and we know Item (e) because (63) tells us that τ' cannot refer to $\mathbf{r}$ and therefore it makes no difference whether $\mathbf{r}$ is in the domain of the store typing.

Therefore, we conclude that $M' : \tau$.

6.5.3.8. TK-RETURN. In this case, we have the following additional information:

$$k = \text{Return}(\rho', k') \tag{66}$$

$$\Sigma \vdash \Gamma' \sim \rho' \tag{67}$$

$$\Gamma'; \Delta; \Sigma; l \vdash k' : \tau' \to \tau \tag{68}$$

From the step function, we have:

$$M' = \langle k', v, \rho', s_1, s_2, l \rangle$$

To show $M' : \tau$, we once again use TM-CONT, which carries the following obligations.

(a) $\Sigma \vdash s_1, s_2$

(b) $\Sigma \vdash \Gamma' \sim \rho'$

(c) $\Delta; l \vdash s_1, s_2$

(d) $\Gamma'; \Delta; \Sigma; l \vdash k' : \tau' \to \tau$

(e) $\Sigma \vdash v : \tau'$

These are all directly contained within the induction hypotheses, so we conclude $M' : \tau$.

6.5.4. A Problematic Example. Consider the following example. This is based on a similar example in **[26]**.

```
(let-region r
  (let ((x (ref r ())))
    (lambda CPU [] (y)
      (let ((z x))
        ())))))
```

We have used `let` here for clarity even though it is not in the semantics we are considering. Let binding can be implemented using an application of a `lambda`.

This example illustrates a difficulty with the semantics presented here. The program is initially well typed but reduces to the following value configuration:

$$\langle \texttt{(closure } y\ \rho\ \texttt{(let ((}z\ \ x\texttt{)) ()))}, \cdot, \cdot \rangle$$

The environment, ρ, will consist of a mapping of x to some label. The label originally had type (ref **r** ()), but now there is no way to assign a type to the label because no regions are active. Thus we will not be able to produce a Γ to use with TV-PROC. The dangling reference captured in x is obviously benign since it is never dereferenced, and attempts to dereference it would cause the original expression to fail to typecheck. Still, this case presents a difficulty for the semantics.

There are several ways to remedy this. Fluet and Morrisett solve it by replacing references to destroyed regions with a special dead region marker [**26**]. It might also be possible to fix this issue by adding a small restriction to the TE-VAR rule. In any case, solutions to this issue are well known and are independent of the focus in this work on regions moving between different compute devices.

6.6. Auxiliary Lemmas

LEMMA 1 (Store Typing Commutativity). *If* $\Sigma \vdash s_1, s_2$ *then* $\Sigma \vdash s_2, s_1$.

PROOF. This lemma follows directly from the definitions of $\Sigma \vdash s_1, s_2$ and $\Sigma \vdash s_2, s_1$. □

LEMMA 2 (Heap Swapping). $\Delta; \mathit{CPU} \vdash s_1, s_2$ *if and only if* $\Delta; \mathit{GPU} \vdash s_2, s_1$.

PROOF. This lemma follows directly from the definition of $\Delta; \texttt{CPU} \vdash s_1, s_2$ and $\Delta; \texttt{GPU} \vdash s_2, s_1$. □

LEMMA 3 (Frame Rule for Expressions). *If* $\Gamma; \Delta; l \vdash e : \tau; \Delta'$ *then* $\Gamma; \Delta \star \Delta''; l \vdash e : \tau; \Delta' \star \Delta''$

PROOF. By induction on the structure of the derivation of $\Gamma; \Delta; l \vdash e : \tau; \Delta'$. □

LEMMA 4 (Heap Factoring). *If* $\Delta_1 \star [\mathbf{r_1} \mapsto l_1] = \Delta_2 \star [\mathbf{r_2} \mapsto l_2]$ *and* $r_1 \neq r_2$ *then there exists a* Δ_3 *such that* $\Delta_1 \star [\mathbf{r_1} \mapsto l_1] = \Delta_3 \star [\mathbf{r_1} \mapsto l_1] \star [\mathbf{r_2} \mapsto l_2]$.

PROOF. We can rewrite both Δ_1 and Δ_2 as a sequence of separating conjunctions, such as $[\mathbf{r'_1} \mapsto l'_1] \star \ldots \star [\mathbf{r'_n} \mapsto l'_n]$. Because the two sides of the equation are equal, we know that the expansion will have the same set of factors on either side. Choose the factors that are common between Δ_1 and Δ_2 to build Δ_3. □

6.7. Designing for Proof Mechanization

Although the type safety proof from Section 6.5 has not yet been mechanized, the semantics was designed in such a way as to make mechanization easier in the future. The discussion in this chapter is in the context of the Coq Proof Assistant [**54**], but should be applicable to proof assistants in general.

One common challenge in mechanized proofs about programming languages is handling identifier names. To be fair, naming has proved challenging in other contexts as well. The rules for capture-avoiding substitution in the λ-Calculus are subtle. The name resolution rules for C++ are still under study [**84**].

Several techniques have been developed to avoid the difficulties with α-equivalence and capture avoiding substitution in mechanized proof assistants. Among these are the locally nameless representation, in which bound variables are represented as De Bruijn indices while free variables retain their names [**15**], and higher order abstract syntax (HOAS), which uses the binding structures from the host language to implement the binding structures in the language being modeled [**64**].

These naming issues are one of the main reasons the semantics presented in this chapter is built around an interpreter. The interpreter implements binding through a well-defined, mechanical process of looking up variables in an environment. The runtime environment is explicitly represented in the machine state, making it clear what value a variable refers to. Building the semantics around an interpreter has the added benefit of being able to execute test programs to verify that the semantics captures the desired intuition.

The interpreter by itself introduces some new problems. Coq requires all functions be total, meaning in particular that all functions must terminate. Requiring the interpreter to terminate would have the undesirable side effect of requiring all programs in Core Harlan

to terminate. This is the primary motivation for transforming the interpreter to an abstract machine. Instead of having a potentially non-terminating recursive evaluator function, we instead represent the complete state of the machine—basically, the expression being evaluated and the continuation—as a data structure. We then transform the interpreter to be a function that applies one step of the computation. Each step is then guaranteed to terminate, and evaluation can continue with a potentially unbounded number of steps of the interpreter.

Termination alone is not enough to guarantee totality in the evaluator. The result of executing some Core Harlan programs is undefined, such as if a program attempts to dereference a pointer into a non-existent region. While the type safety theorem guarantees that this will not happen for well-typed programs, not all syntactically correct Core Harlan programs are well-typed. We ensure totality then by explicitly representing an erroneous machine configuration, $\perp$. This way, the interpreter can still return a value that says the machine is undefined if an error occurs.

Describing the semantics in terms of a total step function has another benefit. Type safety proofs are traditionally presented using a combination of progress and preservation. Progress says that if a machine configuration is well-typed, then there is another configuration that the evaluation can step to. Preservation says that the type of each intermediate configuration remains the same. In our case, progress is essentially free. Because the step function is total, any configuration has a next step. To prove type safety then, we just have to show that each of these steps retains the same type as before.

CHAPTER 7

Harlan Case Studies

Having discussed the design, implementation and semantics of the Harlan language, we will now explore several applications to show Harlan's utility on several classes of problems.

We will consider three main applications:

- Dense Matrix Multiplication (Section 7.2)
- Breadth first search and strongly connected components (Section 7.4)
- Ray tracing (Section 7.3)

In addition to these three main applications, we will also explore how Harlan programs can interact with programs written in other languages (Section 7.5) and a number of microbenchmarks to characterize more specific details of Harlan's performance (Section 7.6).

These case studies have been selected to show that Harlan is expressive enough to implement several traditional GPU computing applications while also simplifying some applications that are difficult to implement in existing GPU programming systems. Benchmark results are provided to characterize the performance of Harlan programs as well as to inform future optimization efforts.

7.1. Benchmarking Methodology

This chapter includes several benchmarks comparing programs written in Harlan with similar versions in other languages. All of the benchmarks presented here were performed on Tesla, which consists of two Intel Xeon E5-2670 v3 CPUs at 2.30GHz, 32GB RAM and two NVIDIA Tesla K40c GPUs. Each GPU has 12GB RAM and 2880 CUDA cores.

Each benchmark is run multiple times. The graphs show the average of the results from each trial. Additionally, the graphs include error bars marking the 95% confidence

```
(define (mat-mul A B)
    (let* ((N (length A))
           (Bt (kernel* ((i (iota N))
                         (j (iota N)))
                 (vector-ref (vector-ref B j) i))))
      (kernel* ((a A)
                (b Bt))
        (reduce + (kernel ((a a) (b b)) (* a b))))))
```

FIGURE 7.1. The core dense matrix multiplication kernel. This assumes we are multiplying two square matrices, A and B.

interval computed using Student's T-Test. Unless otherwise mentioned, each benchmark was run ten times.

7.2. Dense Matrix Multiplication

Dense matrix multiplication is a staple of many scientific computation problems. Matrix multiplication is a well studied problem with highly tuned solutions. Thus, it is also useful for characterizing Harlan's raw numeric performance.

The core matrix multiply kernel is shown in Figure 7.1. The code fragment works on two square matrices, `A` and `B` of size `N`. The body of the outer kernel expression computes a dot product of a row in `A` with one column in `B`. The columns of `B` are extracted by first computing the transpose of `B` and storing it in `Bt`.

Figure 7.2 compares the execution time of several implementations of dense matrix multiply at a variety of sizes. The results displayed here are the average of five runs. The Harlan benchmark is based on the kernel showed in Figure 7.1. The OpenCL version is a simple OpenCL kernel without a lot of hand tuning applied. This is meant to represent "equivalent programmer effort" to the Harlan program. The code for the OpenCL matrix multiplication kernel is shown in Figure 7.3. Finally, the CuBLAS version represents a highly tuned, state of the art implementation of dense matrix multiply (DMM).

The CuBLAS version is by far the fastest, as would be expected. The OpenCL version is roughly two to ten times faster than the Harlan version. One possible reason is that Harlan does memory allocation within the kernel, while the OpenCL version allocates all

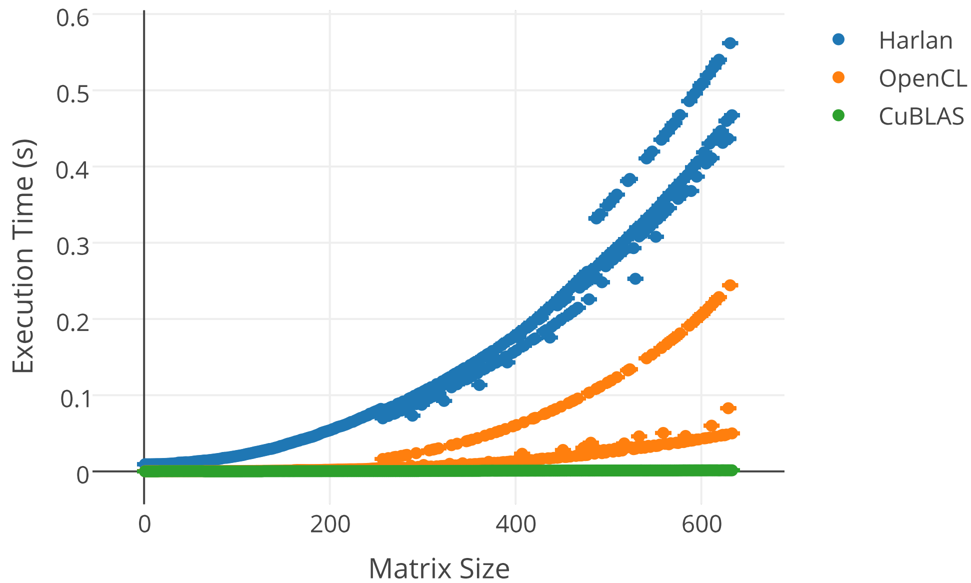


FIGURE 7.2. Dense matrix multiplication performance.

```
__kernel
void dmm(int N,
         __global float *A,
         __global float *B,
         __global float *C)
{
int i = get_global_id(0);
int j = get_global_id(1);

#define ref(A, i, j) ((A)[(i) * N + (j)])

float acc = 0;
for(int k = 0; k < N; ++k) {
acc += ref(A, i, k) * ref(B, k, j);
}

ref(C, i, j) = acc;
}
```

FIGURE 7.3. OpenCL dense matrix multiplication kernel.

of its memory up front. Harlan does not yet have a scalable memory allocation algorithm and instead all threads trying to allocate contend on doing atomic operations on a single word in memory.

One interesting feature of the OpenCL data, and to a lesson extent the Harlan data, is that the execution times fall into discrete bands. Manual inspection of the data suggests that the slower times are when the matrix size is a prime number. The OpenCL program is written so that it launches exactly the right number of work items and lets the OpenCL runtime choose how to divide these into blocks. The block size must evenly divide the total number of work items, which is obviously impossible if there is a prime number of work items. The result is that much of the GPU's parallelism goes unused. By contrast, CUDA programs are usually written to over-approximate the number of work items and have threads that fall outside the desired range drop out.

Inspection of the Harlan data suggests a similar explanation for the banding there as well.

In an earlier version of this benchmark, the Harlan program was written with direct references into `B` when needed rather than precomputing the transpose. This resulted in significantly poorer performance. Precomputing the transpose improves the memory access pattern, which makes up for the cost in doing the precomputation. Harlan's optimizer is almost powerful enough to do the transformation from explicitly representing the transpose to computing it inline, which means future work could explore autotuning to automatically find the best matrix multiplication algorithm.

7.3. 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. Rather than approximate a sphere as a triangle mesh, for example, 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.

```
(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 7.4. 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.

A portion of the ray tracer code is given in Figure 7.4. 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 at 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 created 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 7.3.1.

7.3.1. Ray Tracing Performance. 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. Indeed, Hong et al's work on GPU graph algorithms was specifically designed to minimize thread divergence by programming using a warp-centric model [**41**]. To measure this penalty, we ran two versions of the ray tracing program from the previous section. 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.

Figure 7.5 shows the results of running five iterations of this benchmark on Tesla. There is not a significant difference in the performance for either variant. This is likely because

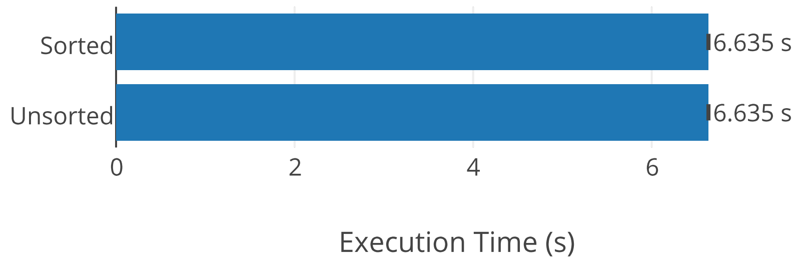
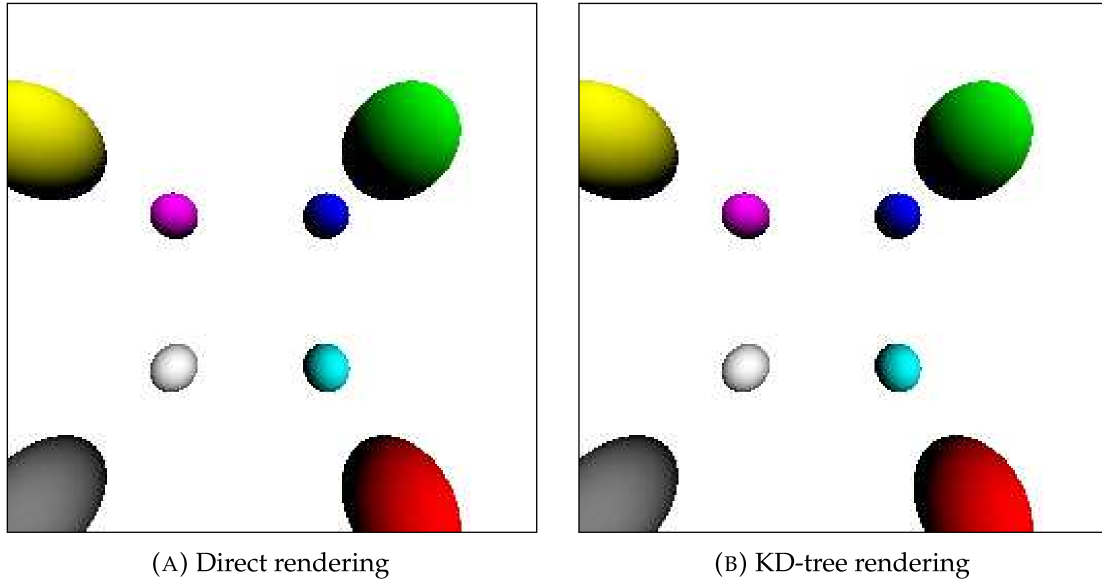


FIGURE 7.5. The effect of thread divergence on ray tracing performance. In this particular case, thread divergence does not make a significant impact on the overall execution time.

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, it may be possible to structure them to minimize these effects. This is analogous to how specializing compilers for languages such as JavaScript can gain impressive speedups based on the observation that programs are often far less irregular than the language semantics allows [**28**].

7.3.2. Ray Tracing with KD-Trees. The previous section defined an extremely naive ray tracing algorithm. One particular problem is that each ray must test for an intersection with all surfaces in the scene. Many of these checks can be avoided, and KD-trees are one way to do this. KD-trees are an instance of a binary space partition (BSP) algorithm. BSP algorithms work by recursively dividing a space into half spaces that each contain a portion of the objects in the scene. A KD-tree is a special case of a binary space partition in which the faces of the subspaces are all aligned with either the X, Y or Z axis. For a ray to intersect a surface contained within one of the volumes in the KD-tree it must also intersect one of the faces of the prism. This gives us an easy way to eliminate large numbers



(A) Direct rendering

(B) KD-tree rendering

FIGURE 7.6. An image rendered by ray tracing using a direct style and KD trees. Importantly, they are identical.

of objects, since if the ray does not intersect the objects' containing volume then there is no reason to consider that object at all.

Implementing this algorithm leads to a code where all ray traverse the same tree structure in parallel, which is well-suited to Harlan's support for ADTs.

The implementation of KD-trees in this section is by no means the first GPU-enabled KD-tree implementation. For a discussion of several implementation techniques, see [**65**]. The discussion here is meant to emphasize how Harlan's high level features simplifies implementing a KD-tree.

As a preview, Figure 7.6 shows an example of an image rendered using both a direct style algorithm, like the one shown in Figure 7.4, and the KD-tree algorithm discussed here. It is important that they look the same, as the algorithms should not produce any visual differences.

There are two main phases to KD-tree rendering. First, we must build the tree from a given scene. Second, we perform the actual rendering. The tree only needs to be built once per scene and then it can be reused for many subsequent images.

There are many possible strategies to build the tree. At first glance, it seems the ideal strategy is to try to balance the number of objects in each subtree. This is in practice less than ideal as it leads to often having to traverse a high number of tree nodes. Instead, a good strategy is to try to build very large volumes that are completely empty with very small volumes that contain objects in the scene. In this way, the algorithm can often prune very large portions of the space very quickly. One popular technique is called the *surface area heuristic*, which is built on the observation that the probability a ray will intersect with a box is proportional to its surface area. At each step in building the tree, the algorithm may choose to split along the X, Y or Z axis. The surface area heuristic computes a cost of each potential split as follows.

$$C = k_{traversal} + NAk_{intersection}$$

Here C is the cost of a potential volume, $k_{traversal}$ is a constant representing the fixed cost of reading a node from memory, N is the number of objects contained within the volume, A is the surface area of that volume, and $k_{intersection}$ is the cost of computing an intersection of a ray with an object. The two constants, $k_{traversal}$ and $k_{intersection}$, should be empirically determined.

A set of objects in a volume gives a number of potential split points, which are each of the edges of each object's bounding boxes. We build the tree by estimating the cost resulting from each of these splits and then choosing the best one. The core of the Harlan code to do this is shown in Figure 7.7. This code evaluates splits on each axis and then chooses the best one of the three and also compares against the cost if it were to just leave the volume in tact. If the algorithm decides to split the volume, then it recursively builds the tree of objects on the two subvolumes.

Figure 7.8 shows a dump of the tree data structure produced by running this algorithm on a scene consisting of 8 spheres of radius 3 positioned on the corners of a $20 \times 20 \times 20$ cube. Figure 7.9 shows a graphical depiction of a 2D projection of this scene and tree.

Having built the tree, the next step is to render an image. The algorithm is as follows.

```
(define (build-tree shapes dims)
  (if (> (length shapes) 0)
      (let ((nosplit-cost (* (box-surface-area dims)
                             (* (int->float (length shapes))
                                (intersection-cost))))
            (best-x (find-split dims shapes (XAxis)))
            (best-y (find-split dims shapes (YAxis)))
            (best-z (find-split dims shapes (ZAxis))))
        (match (best-split best-x (best-split best-y best-z))
          ((SplitCost axis plane cost)
           (if (< cost nosplit-cost)
               (let ((lefts (filter (lambda (shape)
                                      (left-of? axis plane shape))
                                    shapes))
                     (rights (filter (lambda (shape)
                                       (right-of?
                                        axis plane shape))
                                     shapes)))
                 (match (split-box dims axis plane)
                   ((BoxPair left-box right-box)
                    (Split axis plane
                           (build-tree lefts left-box)
                           (build-tree rights right-box)))))
               (Leaf shapes)))))
      (Leaf (vector))))
```

FIGURE 7.7. Harlan code to build a KD-tree.

(1) If the node is a leaf node, fall back on the direct rendering algorithm for the objects contained at this node.
(2) If the node is a split node, determine which subspaces intersect the view ray and recursively traverse each space that is intersected.

The code to do this in Harlan is given in Figure 7.10.

The traversal code represents the view ray parametrically, where the ray is defined as:

$$\mathbf{x} = \mathbf{x_0} + t\mathbf{d}$$

The traversal tracks two variables, t_0 and t_1, which represent the point at which the ray enters the bounding volume and the point at which it exits. Then, for a given split the

```
(Split (ZAxis) -7
       (Split (YAxis) -7
              (Split (XAxis) -7
                     (Leaf [(Sphere -10 -10 -10 3)])
                     (Split (XAxis) 7
                            (Leaf [])
                            (Leaf [(Sphere 10 -10 -10 3)])))
              (Split (YAxis) 7
                     (Leaf [])
                     (Split (XAxis) 7
                            (Split (XAxis) -7
                                   (Leaf [(Sphere -10 10 -10 3)])
                                   (Leaf []))
                            (Leaf [(Sphere 10 10 -10 3)]))))
       (Split (ZAxis) 7
              (Leaf [])
              (Split (YAxis) 7
                     (Split (YAxis) -7
                            (Split (XAxis) 7
                                   (Split
                                     (XAxis) -7
                                     (Leaf
                                       [(Sphere -10 -10 10 3)])
                                     (Leaf []))
                                   (Leaf [(Sphere 10 -10 10 3)]))
                            (Leaf []))
                     (Split (XAxis) 7
                            (Split (XAxis) -7
                                   (Leaf [(Sphere -10 10 10 3)])
                                   (Leaf []))
                            (Leaf [(Sphere 10 10 10 3)])))))
```

FIGURE 7.8. A KD-Tree produced by Harlan.

traversal finds the t for which the ray intersections the split plane,

$$t = \frac{s - x_0}{d}$$

where s is the coordinate for the dividing plane, x_0 represents the component of $\mathbf{x_0}$ in the axis of the split and d is the component of $\mathbf{d}$ in the axis of the split.

There are three possibilities for t in relation to t_0 and t_1, which are illustrated in Figure 7.11. These possibilities illustrate the criteria our traversal code uses to decide which

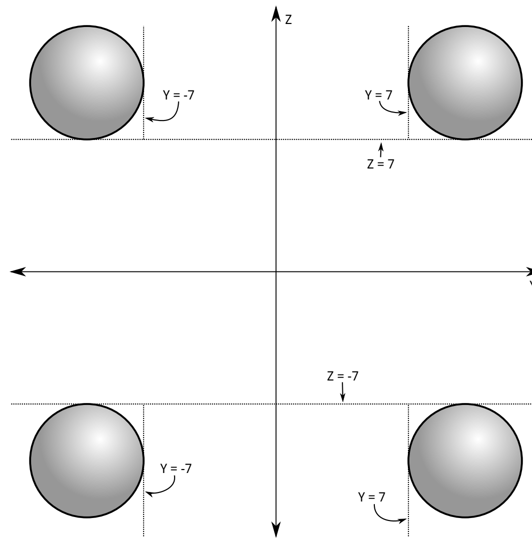


FIGURE 7.9. A 2D projection of the scene and tree from Figure 7.6.

```
(define (traverse-kd-tree tree origin dir ray-bounds)
  (match tree
    ((Leaf shapes) (select-point shapes origin dir))
    ((Split axis plane left right)
     (let ((reverse (< (select-coord axis dir) 0)))
       (let ((left  (if reverse right left))
             (right (if reverse left right)))
         (let ((t (plane-intersection axis plane origin dir)))
           (match ray-bounds
             ((FloatPair t0 t1)
              (if (< t t0)
                  (traverse-kd-tree right origin dir ray-bounds)
                  (if (<= t t1)
                      (select-closest
                        (traverse-kd-tree left origin dir
                                          (FloatPair t0 t))
                        (traverse-kd-tree right origin dir
                                          (FloatPair t t1)))
                      (traverse-kd-tree right
                                        origin
                                        dir
                                        ray-bounds))))))))))
```

FIGURE 7.10. Harlan code to traverse a KD tree.

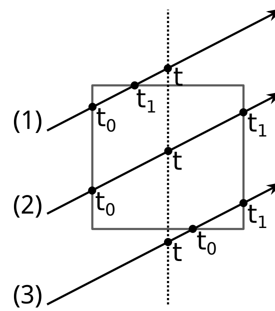


FIGURE 7.11. Possible intersections of a ray and two bounding volumes.

subspaces to traverse. In (1), we have $t > t_1$, which means the ray intersects the dividing plane outside of the current bounding volume. In this case the algorithm only has to traverse the left subspace. In (2), $t_0 < t < t_1$, which means the ray passes through both subspaces. In this case, the traversal algorithm then needs to recur on both the left and right halves. Finally, (3) has $t < t_0$, which means the ray only intersects the right half-space and we can avoid traversing the left subspace.

One final detail is that the ray might be oriented opposite to the left and right subspaces. That is, Figure 7.11 assumes the ray travels from left to right. If the ray is traveling the opposite direction, we simply need to reverse the left and right subspaces. This is accomplished in the tree traversal code by checking whether the direction of the ray is negative in the component of the dividing axis, shown in the test for `(< (select-coord axis dir) 0)` in Figure 7.10.

Harlan's support for high level functional language features, especially ADTs, greatly simplified the implementation of the KD tree algorithm. Indeed, this was the author's first experience implementing a KD tree. Harlan made it easy to focus on implementing the algorithm rather than the artifactual details of encoding the data structures for the target machine. Some parts of the implementation, however, were more verbose than is ideal. Several one-off data structures were needed, such as `IntPair` and `FloatPair`, as well as comparison and merging functions for the various auxiliary structures. Harlan's expressiveness could be significantly improved with the addition of polymorphism, which would eliminate the need to manually specialize structures at each type.

7.4. Graph Algorithms

Graph algorithms are the subject of much research lately due to their applications in many domains such as social network analysis, product recommendation engines, web search, intelligence, circuit design, and many others. These algorithms are interesting not just because of their wide application, but because they involve irregular memory access patterns with relatively low computation compared to the volume of data. This means that many of the optimization techniques from more traditional dense linear algebra algorithms are not nearly as effective on graph algorithms. GPUs are an attractive target for graph algorithms in part because of their massive parallelism, but also because of their much higher memory bandwidth than traditional CPUs. Still, this bandwidth is tuned for regular, streaming access patterns which means irregular graph algorithms struggle to achieve peak memory bandwidth. Despite these challenges, a number of recent works have achieved good performance for GPU graph algorithms [**23**, **34**, **41**, **57**, **81**, **82**, **88**].

In this section, we will explore two graph algorithm implementations in Harlan. The first is BFS in Section 7.4.1 and the second is strongly connected components in Section 7.4.2.

In general a graph G can be thought of as a pair of two sets, (V, E). V is the vertex set while E is the edge set. E is a subset of $V \times V$, and $(v_1, v_2) \in E$ means that vertex v_1 is directly connected to v_2. Graphs can be directed, meaning that $(v_2, v_1) \in E$ whenever $(v_, v_2) \in E$, or undirected, which does not carry that same restriction. We will sometimes speak of the *transpose* of a graph, which is the graph obtained by reversing all of the edges in E.

7.4.1. Breadth First Search. BFS is a relatively simple graph algorithm that also serves as a basis for several other algorithms.

The goal of a breadth first search is to traverse all edges radiating from a starting node such that all children of a node are visited before any of its children. The result is often a tree of the edges and vertices traversed by the BFS. Consider the graph in Figure 7.12. Each node in this graph has been labeled with the length of the path from node A when the graph is traversed in a breadth-first fashion. The search in this example starts at node A, then nodes B, D and E can be reached by following one additional edge. The next iteration

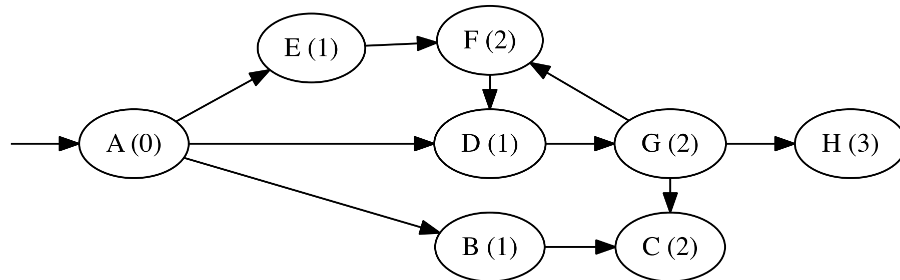


FIGURE 7.12. An example graph on which to perform a breadth first search.

would visit nodes C, G and F, because these are reachable by one edge from one of B, D or E. Finally, the last remaining node, H, is added in the third iteration because it is reachable in one step from node G.

The first design consideration is how to represent the graph. There are many common choices, such as edge lists, adjacency matrices or incidence matrices. These representations are often isomorphic to matrices, as many graph algorithms are compactly expressed in terms of linear algebra [**49**]. Figure 7.13 shows how the graph in Figure 7.12 looks in several representations. For our purposes, we will represent a graph as a vector of nodes, where each node is represented as a vector of node identifiers which have an incoming edge to the current node. Figure 7.14 shows what the example graph would look like in Harlan. Harlan's format is essentially the transpose of the adjacency matrix in compressed sparse row (CSR) format.

Incidentally, because of the way Harlan lays out data in regions, this vector-of-vectors representation looks very similar in memory to a CSR representation. CSR is a popular way of representing graphs because it helps lay out memory that needs to be accessed together nearby.

This layout also admits convenient processing with Harlan kernels. We can now consider a basic breadth first search kernel.

```
(kernel ((i (iota (length graph)))
```

$$\{(A,B),(A,D),(A,E),(B,C),(D,G),(E,F),(F,D),(G,C),(G,F),(G,H)\}$$

(A) Edge List

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(B) Adjacency matrix. Rows and columns are in order of A through H. The edge (i,j) being present is indicated by a 1 in entry (i,j).

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

(C) Incidence matrix. Each row represents an edge. Again, each column represents the vertex A through H, in order. A -1 in a column indicates that the edge in that row leaves that vertex and a 1 indicates that the edge ends at that vertex..

FIGURE 7.13. Several representations of the graph in Figure 7.12.

```
(vector
  (vector)     ;; A
  (vector A)   ;; B
  (vector B G) ;; C
  (vector A F) ;; D
  (vector A)   ;; E
  (vector E G) ;; F
  (vector D)   ;; G
  (vector G))  ;; H
```

FIGURE 7.14. Harlan's representation of the graph in Figure 7.12. Assume variables are in scope that match vertex names to identifiers. For example, `(let ((A 0) (B 1) ...) ...)`.

```
        (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)))))
```

This kernel works by creating a color vector, which indicates the processing state at each node. At the beginning, all nodes are white except for the starting node which is gray. Then, each white node checks if the color of any node with an incoming edge is gray. If so, that node changes from white to gray. Gray nodes simply become black, indicating that no more processing is required. This performs one level of the BFS, so some surrounding code is needed to iterate this until a fixed point is reached.

The kernel as written now merely performs the traversal. Often we will want some product of the traversal, such as a parent tree. This kernel can be modified to perform some extra book keeping in order to produce a parent tree at the end.

7.4.1.1. *Challenges and Possible Improvements.* Many BFS algorithms implemented in CUDA make heavy use of mutation, making it difficult to duplicate these algorithms in Harlan. BFS algorithms can largely be broken down into top-down and bottom up traversals. Some algorithms even combine both directions in order to improve performance [**3**]. The algorithm presented here in Harlan is an example of a bottom up algorithm, which works well on searches where the frontier—that is, the current set of gray nodes—covers a sizable subset of the graph. On the other hand, many algorithms are top-down, which are very difficult to implement without mutation.

One possible extension to Harlan is a `kernel-update!` form, which would allow kernels to do limited mutation to vectors without sacrificing determinism. Figure 7.15 shows an example of how `kernel-update!` could work, including both the code for a program and the output from running the program. The `kernel-update!` expression

```
(module
  (define (main)
    (let ((a* (iota 10)))
      (println a*)
      (kernel-update!
          ((a a*)
           <- (i (iota 10)))
        (* a i))
      (println a*)
    0))
```

(A) Code

```
[ 0 1 2 3 4 5 6 7 8 9 ]
[ 0 1 4 9 16 25 36 49 64 81]
```

(B) Output

FIGURE 7.15. An example of how `kernel-update!` might work.

indicates that it iterates over `a*` and `(iota 10)`, binding `a` to an element in `a*` and `i` to the corresponding element in `(iota 10)`. The difference is that while a normal `kernel` form returns a new vector, `kernel-update!` overwrites `a*` with the new values. In this way, it is similar to the following expression:

```
(set! a* (kernel ((a a*) (i (iota 10))) (* a i)))
```

Doing this safely would require careful analysis to ensure that this mutation is not unexpectedly observed. Alternatively, the compiler might be able to automatically replace `a*` with the updated version if it can detect that the old version is not read again, without requiring any changes to the language.

The `kernel-update!` form as described here would not be powerful enough to switch to a top-down BFS algorithm, but it might be able to avoid some memory transfer cost by reusing the storage space for each intermediate result.

7.4.2. Strongly Connected Components. In a directed graph, a strongly connected component (SCC) is a set of nodes such that each node in the component has a path to every other node in the component. Algorithms for finding the strongly connected components in a graph have a variety of applications. As we saw in Chapter 5, the Harlan compiler itself currently uses Tarjan's Algorithm [**79**] to find sets of functions that may recursively call each other in order to remove the recursion.

There are several algorithms for finding strongly connected components in parallel [**30**, **43**, **56**, **73**, **74**]. Many of these are based on breadth-first searches, as this is easier to parallelize than the depth-first search used by Tarjan's algorithm. Graph problems like breadth-first search already have algorithms for the GPU which could be adapted to find strongly connected components [**7**, **35**, **42**].

The algorithm presented here is based on Orzan's coloring algorithm [**61**]. This algorithm uses two traversals. The first assigns colors to each node, while the second uses the coloring from the previous traversal to assign components. Not all components will be discovered during one iteration of these two steps, so they are repeated until all vertices have been assigned a components.

To assign colors, each vertex is initialized with its own vertex identifier as its color. These then propagate forward through the graph, and each vertex takes the largest of its incoming colors.

The second traversal starts by determining the roots of each component, which are the vertices whose color matches its identifier. Then any component that is reachable backwards from the root and has the same color as the root is part of that component.

We are now ready to develop the code to compute SCCs in Harlan. The Harlan code will utilize two data structures, `graph` and `reverse-graph`, which are the graph and the graph with all edges reversed. These will be used to perform the backward traversals and forward traversals. Surprisingly, the reverse graph is used for forward traversals while the normal graph is used for backward traversals. This is because Harlan processes vertices in a bottom-up fashion.

The code to assign vertex colors is shown in Figure 7.16. This function also uses a `components` vector which is used to avoid processing vertices that already have a component assigned. The `components` vector is made up of the following data type:

```
(define-datatype Component
  (None)
  (Color int))
```

```
(define (color-vertices reverse-graph colors components)
  (let ((next-colors
         (kernel ((color colors)
                  (edges reverse-graph)
                  (comp components))
           (match comp
             ((None)
              (if (> (length edges) 0)
                  (max color (inner-reduce max
                              (kernel ((e edges))
                                (vector-ref colors e))))
                  color))
             ((Color _) -1)))))
    (if (= colors next-colors)
        colors
        (color-vertices reverse-graph next-colors components))))
```

FIGURE 7.16. Harlan SCC coloring code.

Obviously, `(None)` indicates that a component is not yet assigned, while `(Color i)` indicates that a vertex has component `i`. The code in Figure 7.16 is very similar to the breadth-first search code we saw earlier.

The function that assigns components from the coloring is given in Figure 7.17. This has a similar structure, as it is BFS traversal of the graph. This time, it uses the regular `graph` structure in order to effect a backwards traversal. Much of the length of this function is the `lambda` expression passed to the reduction, which merges the incoming component assignments along with the current vertex' coloring.

Finally, the code to iterate the two previous functions is shown in Figure 7.18. This simply does the coloring and component assignment steps, iterating them until a fixed point is reached. The `components-eq` function is needed instead of just = because Harlan currently does not synthesize equality operators for most ADTs.

Knowing how to perform breadth first search and starting with an appropriate SCC detection algorithm, we have seen that it is not too hard to implement SCCs in Harlan.

```
(define (assign-components graph colors components)
  (let ((next-components
         (kernel ((id (iota (length graph)))
                  (color colors)
                  (comp components)
                  (edges graph))
           (match comp
             ((Color _) comp)
             ((None)
              (if (= id (vector-ref colors id))
                  (Color id)
                  (if (> (length edges) 0)
                      (inner-reduce (lambda (a b)
                                      (match a
                                        ((None)
                                         (match b
                                           ((None) (None))
                                           ((Color i)
                                            (if (= i color)
                                                (Color i)
                                                (None)))))
                                        ((Color i)
                                         (if (= i color)
                                             (Color i)
                                             (match b
                                               ((None) (None))
                                               ((Color i)
                                                (if (= i color)
                                                    (Color i)
                                                    (None)))))))))
                                    (kernel ((e edges))
                                      (vector-ref components e)))
                      (None)))))))))
    (if (components-eq components next-components)
        components
        (assign-components graph colors next-components))))
```

FIGURE 7.17. Harlan SCC component assignment code.

7.5. Integrating with external applications

Languages generally cannot live in a vacuum. Many practitioners already have large applications and it would be unreasonable to ask them to rewrite the entire application in a

```
(define (update-components graph reverse-graph components)
  (let ((colors
         (color-vertices reverse-graph
                         (iota (length reverse-graph))
                         components)))
    (let ((new-components
           (assign-components graph colors components)))
      (if (components-eq components new-components)
          components
          (update-components graph
                             reverse-graph
                             new-components)))))
```

FIGURE 7.18. Harlan SCC driver code.

new language. Instead, Harlan can both call into external code and provides mechanisms to make its code available to other applications.

The fact that Harlan compiles to C++ with OpenCL simplifies many aspects of the foreign function interface (FFI). Many existing languages include C interfaces, so the existing C interface can be used to interact with code written in Harlan as well. Additionally, Harlan inherits most of the calling convention of the host system's C compiler, further simplifying integration with external code.

Still, Harlan differs from C++ in several ways, which leads to several challenges for an FFI. These challenges include:

- Name mangling.
- Mapping Harlan and C++ types.
- Mutation.

We now address these challenges in turn.

7.5.1. Name mangling. Harlan, due to its Scheme heritage, allows several characters in identifiers that are illegal in C++. For example, dashes are frequently used as word separators within identifiers, while in C++ underscores or camel case are used. There are several options to address this:

(1) Use only legal C++ names in Harlan functions that may be called externally.
(2) Restrict the characters allowed in Harlan identifiers. This is less than ideal.

(3) Provide a preprocessor to translate Harlan names in C++ files into their mangled equivalents.
(4) Require the programmer to type the mangled name when calling a Harlan function from C++.

Options 1 and 4 are available without any support from the compiler, and these are currently the allowed options. A preprocessor would easily be written, as it requires little more than a find and replace in a source code file.

7.5.2. Mapping Harlan and C++ types. Type systems vary significantly between languages. This is especially true with Harlan, due to its region-based memory system. Some Harlan's base types, such as `int`, `float` or `char` correspond directly to a type in C++. Even Harlan's more complex types, like ADTs or procedures, are compiled into C++ structs, which facilitates interoperability between Harlan and C++.

There are two variants to consider. First, how can Harlan interact with data in a native C++ format? Second, how can C++ manipulate data generated by Harlan?

To address the first question, Harlan has some support for native C++ types. In the case of pointers, Harlan includes functions like `unsafe-deref-int` or `unsafe-set!-int` (and variants for `float` and `char` data). These naturally include all of the safety of direct pointer manipulation in C++, but to allow Harlan to read and write simple data from C++ code.

At the moment, Harlan does not support C++ data such as structs and classes. Much of the compiler does support these features, since they are used to implement Harlan features such as ADTs, meaning if needed, facilities for manipulating structs or classes in C++ could be added to Harlan's surface language without too much trouble.

For the second question, the Harlan compiler could be extended to generate C++ accessors for Harlan data. This includes tools to interact with the region system (although the programmer must manage region data manually and explicitly) as well as wrappers to work with more complex objects as though they were native C++ objects. This is similar to

tools such as SWIG [**4**], but more tailored to the specifics of Harlan's data types and region system.

7.5.3. Mutation. Data in Harlan is immutable, but this is not the case in many languages that Harlan will interact with. The Harlan FFI currently does not specify the behavior in the presence of data mutated by outside sources. In many cases, mutation will probably be safe, but Harlan optimizes code under the assumption that data does not change. Were this an invalid assumption, Harlan programs could behave in unpredictable ways. More precise, permissive and safe rules could be the subject of future work.

7.5.4. Raw Data Import. Although Harlan does not natively operate on machine-level pointers, it does provide an unsafe API to read and write from machine pointers. These procedures cannot be used from kernels, because heap data in Harlan cannot move to the GPU unless it is stored in a region. Instead, these procedures provide a crude way for Harlan programs to copy data out of a buffer provided by an external application. Once the data has been copied, the Harlan code can compute with it as normal and then use the same unsafe API to copy the results back to the calling program.

This API also allows Harlan programs to directly call some C library functions.

The API is considered unsafe because it does not perform any array bounds checking or have any way of determining whether pointers actually point to the type of data they claim to. Thus, it is recommended that one of the higher level foreign function interfaces be used when possible.

7.6. GPU Performance Characterization

Harlan has a number of implementation decisions that depend on characteristics of the target architecture. In this section, we will explore several microbenchmarks that attempt to measure some of these characteristics.

7.6.1. GPU Memory Performance. We argued in Section 4.2.3 that the per-transfer overhead is small relative to the time spent doing the actual transfer. Figure 7.19 shows

FIGURE 7.19. Transfer times between the CPU and GPU memory for buffers of various sizes.

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 7.20 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 8MB 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.

7.6.2. Roofline Model. The roofline model [**85**] is a simple way to visualize the impact of memory bandwidth and computational throughput for processors. It compares

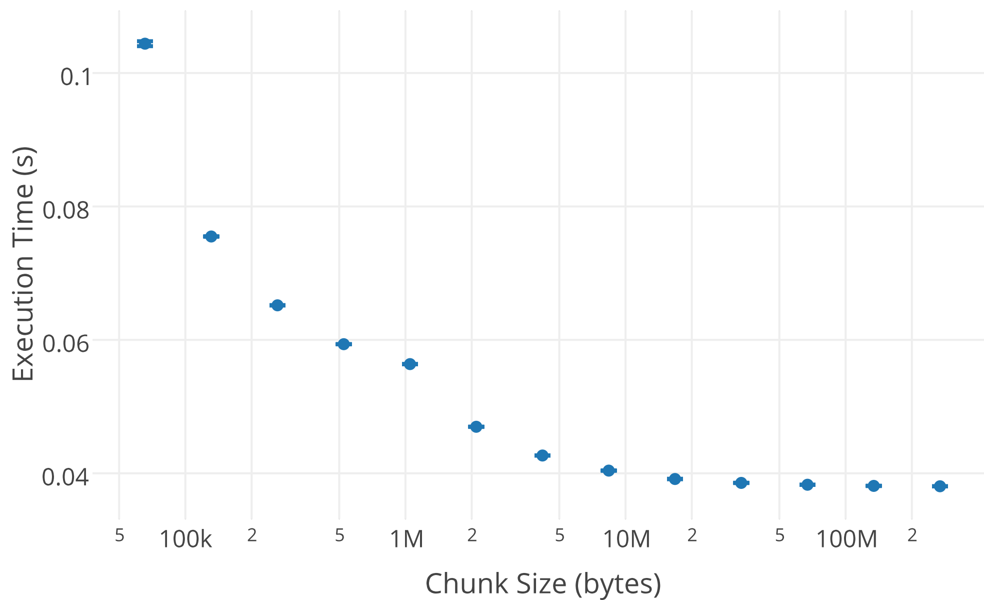


FIGURE 7.20. Time to transfer 256MB of data from the CPU to the GPU, dividing the total data into chunks. The per-transfer overhead is minimal when the chunk size is 8MB or greater.

the achieved floating point operations per second (flops) with the arithmetic intensity, or number of floating point operations per byte of memory traffic, of a computation. Up to a certain threshold, the flops increase linearly as the memory system is not able to provide data fast enough to keep the compute units busy. After that threshold, the curve is basically flat, as there is enough work to keep all of the compute units busy.

Figure 7.21 shows an empirically determined roofline model for the NVIDIA Tesla K40c GPU in Tesla. This was produced by a benchmark that computes a Taylor Series expansion for a vector of many randomly determined points. The Taylor Series expansion makes it easy to vary the arithmetic intensity of the computation by controlling how many iterations of the expansion are computed. The best fit curve shows a peak computation rate of 1.61 Tflops and memory transfer rate of 308 GB/sec. These match the published specifications, although the memory bandwidth is higher than expected **[18]**. One interesting phenomenon is that the performance drops off after a time. This is because the GPU can only run at full power for a limited time and will slow down in order to keep the average power consumption within its design limits. The curve in Figure 7.21 labeled

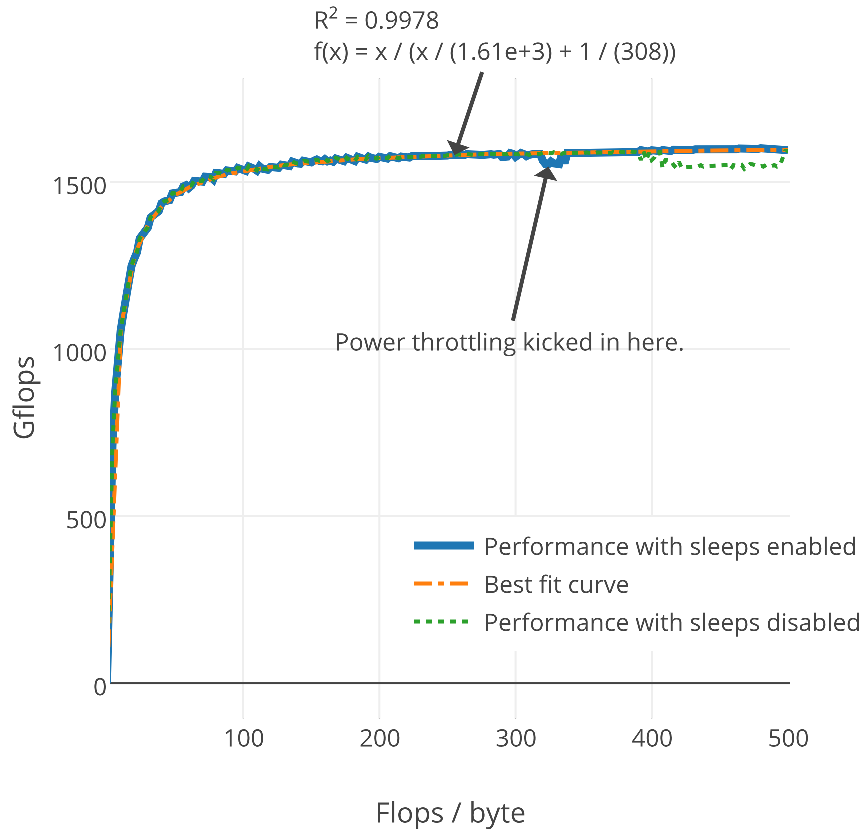


FIGURE 7.21. Roofline model for NVIDIA Tesla K40c GPU.

"Performance with sleeps enabled" shows what happens when the benchmark code inserts a delay between each iteration after a certain point. This reduces the average power consumption so that each iteration runs at full speed.

Models such as this could be useful to Harlan in the future, since Harlan reserves broad latitude in scheduling kernels. Although currently kernels are always scheduled onto a single OpenCL device (typically a GPU), future versions of Harlan could support multiple devices at once. The Harlan compiler could analyze kernels in order to estimate its arithmetic intensity and then decide whether it is worth sending the data across the PCI bus to compute on the GPU or if it would be better to simply compute on the host CPU.

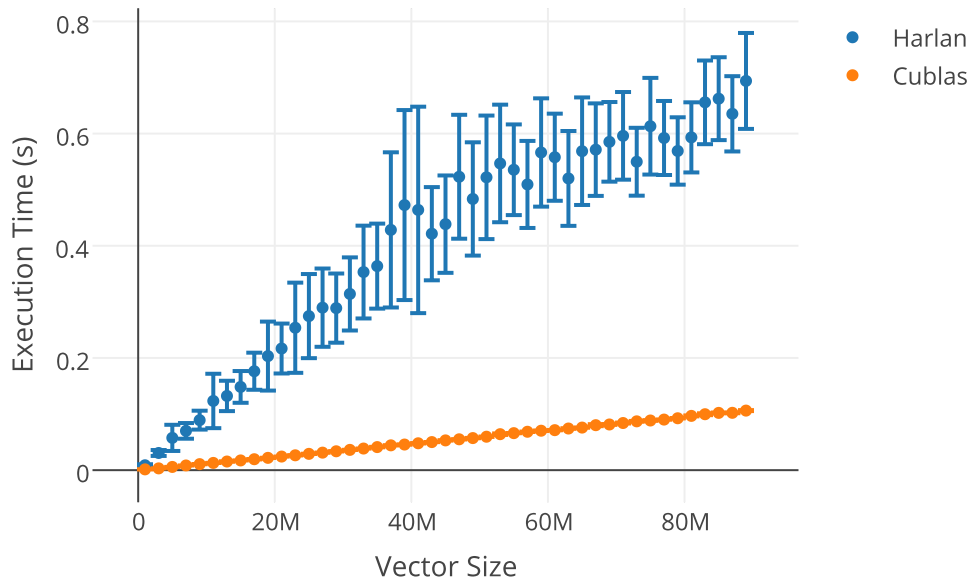


FIGURE 7.22. Vector addition in Harlan compared with vector addition in CUBLAS.

7.6.3. Vector Addition. Figure 7.22 shows the results of a vector addition benchmark in Harlan compared with a similar call to CUBLAS. The timings include the time to transfer memory as well as the actual kernel execution time. Harlan significantly under performs 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.

7.6.4. Dot Product. Harlan performs much better on the dot product benchmark, as shown in Figure 7.23. Here, Harlan's performance is around an order of magnitude slower than the CuBLAS dot product.

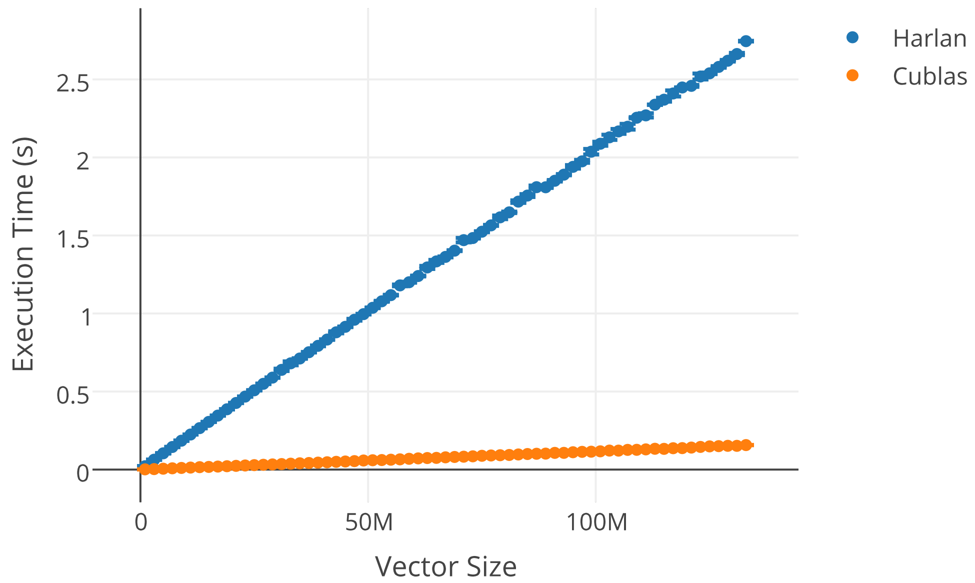


FIGURE 7.23. Dot product in Harlan compared with dot product in CUBLAS.

CHAPTER 8

Conclusion

While GPUs are powerful data parallel processors, programming them is challenging. This difficulty is in part because of the architectural differences between CPUs and GPUs, but one particular challenge is handling the movement of data between the two processors. Programming languages can help.

We have seen in the preceding chapters that region-based memory management is an effective technique for enabling high level features on GPUs and similar devices. In Chapter 4, we were introduced to Harlan, a high level data parallel programming language. Harlan enables GPU computing with its lightweight `kernel` syntax, yet includes a full complement of functional programming features such as hygienic macros, algebraic data types and first class procedures. Harlan's high level features are enabled through its use of RBMM. We saw this in more detail in Chapter 5, when we saw how the Harlan language and its region system are implemented, and how the region system enables rich structures like trees and first class procedures to move between memories.

In Chapter 6, we explored the semantic properties of a model of Harlan's region system. The model language, Core Harlan, includes enough features to illustrate the challenges in moving regions between distinct memories. This is especially true for closures, where it is normally not obvious what data might be captured in a closure. The type safety proof shows that well-typed Core Harlan programs behave well and informs the full Harlan implementation in what invariants its type checker must enforce.

Finally, in Chapter 7 we saw several programs written in Harlan. These show that the language is expressive and simplifies the development of challenging GPU codes. Programs in Harlan can perform as well as programs written in lower level languages, though there is room for more optimization.

The work described in this dissertation shows that region based memory is a useful tool for language implementers to increase language expressiveness even in the presence of multiple memories. Throughout this dissertation, we have touched on several avenues for future work, some of which we will summarize here. While we have focused on the use of regions in CPU-GPU systems, work in other distributed systems is worth pursuing. One promising step in this direction would be to generalize the semantics to support more than two compute devices. Adding the ability to subdivide regions would enable Harlan to more easily handle larger data sets and divide and conquer algorithms. In addition, the design of Harlan leaves the compiler room to apply aggressive transformations and optimizations. Furthermore, different choices in data layout and representation could lead to significant performance improvements. Harlan already performs some of these high level optimizations, but additional work is needed in this area.

Region-based memory management is an effective way of enabling high level features in languages targeting machines with multiple disjoint memories.

Bibliography

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams, IV, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^4 report on the algorithmic language scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, July 1991.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[3] Scott Beamer, Asanović, Krste, and David Patterson. Direction-Optimizing Breadth-First Search. *Scientific Programming*, 21(3-4):137–148, 2013.

[4] David M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[5] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy tree splitting. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 93–104, New York, NY, USA, 2010. ACM.

[6] Lars Bergstrom and John Reppy. Nested data-parallelism on the gpu. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 247–258, New York, NY, USA, 2012. ACM.

[7] M. Bernaschi, M. Bisson, E. Mastrostefano, and D. Rossetti. Breadth first search on APEnet+. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 248–253, 2012.

[8] Guy E Blelloch. *Vector models for data-parallel computing*, volume 75. MIT press Cambridge, 1990.

[9] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[10] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[11] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[13] Bryan C. Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 47–56. ACM, 2011.

[14] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[15] Arthur Charguraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[16] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA: A functional language for data parallelism. Technical Report NVR-2013-001, NVIDIA, July 2013.

[17] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Elsevier Science, October 2003.

[18] NVIDIA Corporation. Tesla k40 and k80 gpu accelerators for servers. `http://web.archive.org/web/20151104151402/http://www.nvidia.com/object/tesla-servers.html`, 2015. Accessed: November 4, 2015.

[19] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 301–312, New York, NY, USA, 1998. ACM.

[20] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. Gpu programming in a high level language: Compiling x10 to cuda. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 8:1–8:10, New York, NY, USA, 2011. ACM.

[21] J. Anish Dev. Bitcoin mining acceleration and performance quantification. In *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*, pages 1–6, May 2014.

[22] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.

[23] Massimo Bernaschi Enrico Mastrostefano. Efficient breadth first search on multi-GPU systems. *Journal of Parallel and Distributed Computing*, 73(9):1292–1305, 2013.

[24] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(01):1–31, 1999.

[25] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[26] Matthew Fluet and Greg Morrisett. Monadic regions. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 103–114, New York, NY, USA, 2004. ACM.

[27] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[28] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[29] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008. This is a revision of the original paper that corrects a few typos.

[30] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 16–25, New York, NY, USA, 1994. ACM.

[31] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[32] Axel Habermaier. The model of computation of CUDA and its formal semantics. Technical Report 2011-14, Informatik, 2011.

[33] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category theory and computer science*, pages 140–157. Springer, 1987.

[34] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[35] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] Mark Harris. CUDA 7 release candidate feature overview: C++11, new libraries, and more. `http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/`, January 2015. Accessed: November 3, 2015.

[37] Chris Hathhorn, Michela Becchi, William L Harrison, and Adam Procter. Formal semantics of heterogeneous CUDA-C: A modular approach with applications. *arXiv preprint arXiv:1211.6193*, 2012.

[38] Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. Declarative parallel programming for GPUs. In *Proceedings of the International Conference on Parallel Computing (ParCo)*, September 2011.

[39] Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. Region-based memory management for GPU programming languages: Enabling rich data structures on a spartan host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 141–155, New York, NY, USA, 2014. ACM.

[40] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. 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.

[41] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[42] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[43] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. Technical report, Stanford University, mar 2013.

[44] Matt Humphreys and Greg Pharr, editors. *Physically Based Rendering*. Morgan Kaufmann, second edition edition, 2010.

[45] Intel. Threading building blocks. `https://www.threadingbuildingblocks.org/`, 2013.

[46] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[47] Feng Ji, Heshan Lin, and Xiaosong Ma. Rsvm: A region-based software virtual memory for gpu. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 269–278, Piscataway, NJ, USA, 2013. IEEE Press.

[48] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.

[49] Jeremy Kepner and John Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[50] Khronos OpenCL Working Group. *The OpenCL Specification*. Khronos OpenCL Working Group, November 2012. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[51] Oleg Kiselyov and Chung-chieh Shan. Lightweight monadic regions. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 1–12, New York, NY, USA, 2008. ACM.

[52] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 55–55, New York, NY, USA, 2001. ACM.

[53] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[54] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[55] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.

[56] William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.*, 65(8):901–910, August 2005.

[57] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128. ACM, 2012.

[58] Mozilla. The Rust programming language. `http://www.rust-lang.org/`, 2013.

[59] NVIDIA. *CUDA C Programming Guide*. NVIDIA, October 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[60] K. J. O'Dwyer and D. Malone. Bitcoin mining and its energy footprint. In *Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET*, pages 280–285, June 2014.

[61] Simona Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.

[62] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[63] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA, 2010. ACM.

[64] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[65] Matt Pharr and Greg Humphreys. Chapter four - primitives and intersection acceleration. In Matt Humphreys and Greg Pharr, editors, *Physically Based Rendering*, pages 182 – 258. Morgan Kaufmann, Boston, second edition edition, 2010.

[66] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[67] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: accelerating flow analysis with GPUs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.

[68] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[69] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[70] Grigore Roşu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006.

[71] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 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*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[72] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, ICFP '04, pages 201–212, New York, NY, USA, 2004. ACM.

[73] Warren Schudy. Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 146–151, New York, NY, USA, 2008. ACM.

[74] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[75] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 309–318, New York, NY, USA, 2008. ACM.

[76] Guy L. Steele, Jr. and W. Daniel Hillis. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 279–297, New York, NY, USA, 1986. ACM.

[77] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[78] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

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

[80] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[81] K. Ueno and T. Suzumura. Parallel distributed breadth first search on GPU. In *2013 20th International Conference on High Performance Computing (HiPC)*, pages 314–323, December 2013.

[82] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 149–160. ACM, 2012.

[83] F. Vasquez, E. M. Garzon, J. A. Martinez, and J.J.Fernandez. The sparse matrix vector product on GPUs. Technical report, University of Almeria, jun 2009.

[84] Larisse Voufo, Marcin Zalewski, and Andrew Lumsdaine. Scoping rules on a platter: A framework for understanding and specifying name binding. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 59–70, New York, NY, USA, 2014. ACM.

[85] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[86] Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 362–374, New York, NY, USA, 2015. ACM.

[87] Ke Yang, Bingsheng He, Qiong Luo, Pedro V. Sander, and Jiaoying Shi. Stack-based parallel recursion on graphics processors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 299–300, New York, NY, USA, 2009. ACM.

[88] Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Scott McMillan, Jonathan Chu, and Andrew Lumsdaine. Dynamic parallelism for simple and efficient GPU graph algorithms. In *5th Workshop on Irregular Applications: Architectures and Algorithms (IA^3)*, November 2015. To Appear.

Eric Holk

Research Interests

Programming language design and implementation, semantics, type systems, compilers, GPU programming, parallel architectures, operating systems.

Education

Indiana University
Ph.D., Computer Science, June 2016.

Indiana University
M.S., Computer Science, May 2013.

Rose-Hulman Institute of Technology
B.S., Computer Science and Mathematics, May 2006.

Florida College
A.A., May 2003.

Employment Experience

Google, Software Engineer, January 2016 – Present.

Indiana University, Research Assistant, August 2009 – December 2015.

University of Utah, Research Associate, May 2013 – August 2013.

Mozilla Corporation, Research Engineering Intern, May 2012 – August 2012.

Mozilla Corporation, Research Engineering Intern, May 2011 – August 2011.

Microsoft Corporation, Software Design Engineer, August 2006 – August 2009.

Sandia National Laboratories, Technical Intern, June 2005 – August 2005.

Rose-Hulman Institute of Technology, Learning Center, Peer Tutor, May 2004 – May 2006.

Publications

[1] B. J. Svensson, M. Vollmer, **Eric Holk**, T. L. McDonell, and R. R. Newton, "Converting data-parallelism to task-parallelism by rewrites: Purely functional programs across multiple GPUs," in *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC 2015)*, Sep. 2015.

[2] M. Vollmer, B. J. Svensson, **Eric Holk**, and R. R. Newton, "Meta-programming and auto-tuning in the search for high performance GPU code," in *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC 2015)*, Sep. 2015.

[3] P. Zhang, **Eric Holk**, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine, "Dynamic parallelism for simple and efficient GPU graph algorithms," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3 2015)*, Nov. 2015.

[4] **Eric Holk**, R. Newton, J. Siek, and A. Lumsdaine, "Region-based memory management for GPU programming languages: Enabling rich data structures on a spartan host," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*, New York, NY, USA: ACM, Oct. 2014.

[5] J. A. Cottom, **Eric Holk**, W. Byrd, A. Chauhan, and A. Lumsdaine, "High level coordination specification," in *Workshop on Leveraging Abstractions and Semantics in High-performance Computing (LASH-C 2014)*, Feb. 2013.

[6] **Eric Holk**, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "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.

[7] J. Hemann and **Eric Holk**, "Visualizing the Turing tarpit," in *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*, Boston, Massachusetts, USA: ACM, Sep. 2013.

[8] W. E. Byrd, **Eric Holk**, and D. P. Friedman, "Minikanren, live and untagged," in *Workshop on Scheme and Functional Programming*, Sep. 2012.

[9] **Eric Holk**, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine, "Kanor – a declarative language for explicit communication," Thirteenth International Symposium on Practical Aspects of Declarative Languages (PADL'11), Jan. 2011.

[10] **Eric Holk**, W. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine, "Declarative parallel programming for GPUs," in *Proceedings of the International Conference on Parallel Computing (ParCo)*, Sep. 2011.

Patents

- **Eric Holk**, Rajneesh Mahajan, Frank D. Yerrace. "Redirection of Multiple Remote Devices." U.S. Patent Number 8,645,559. February 4, 2014.

Awards and Honors

Artifact Exceeds All Expectations Award Awarded by the OOPSLA 2014 artifact evaluation committee for submitting an artifact accompanying [4] that was consistent with the paper, complete, well documented and easy to reuse.

Funding

Mozilla Corporation $49,998. June 2013.

- Research gift awarded to support the development of high level GPU programming languages with a focus on features that would fit well with the Rust programming language.

Professional Activities

- Program Committee Member for *Scheme Workshop 2015*
- Artifact Evaluation Committee Member for *OOPSLA 2015*
- Artifact Evaluation Committee Member from *PLDI 2014*