RUN-TIME COMPILATION AND DYNAMIC MEMORY USE ANALYSIS FOR GPUs

by

DEREK WHITE

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2014

ACKNOWLEDGEMENTS

Foremost, I would like to thank my dissertation advisor Prof. Ishfaq Ahmad for his kind and extraordinary support from my arrival at the University of Texas at Arlington to the conclusion of my Ph.D. study. I would like to thank the rest of my dissertation committee: Prof. Lynn Peterson, Prof. Ramez Elmasri for his guidance and patience, as well as Dr. Bahram Khalili for his unwavering support and dedication to students. Dr. Khalili has been not only a wonderful advisor but friend during this process. I would also like to thank Dr. Nathaniel Nystrom and Dr. Christoph Csallner for their invaluable inspiration and direction during their time on my committee and afterwards.

My sincere thanks goes to Camille Costabile for always patiently working with me through the paperwork and process associated with the completion of Ph.D. milestones.

I thank Rocky Lombardo for my first programming course. Also I thank Dr. Marjan Trutschl, Dr. Urska Cvek, Dr. John Sigle and all of my friends at Louisiana State University Shreveport.

I would like to thank my family: my parents Jerry and Denise White, and my sister Devin for being my biggest supporters and always encouraging me to follow my interests, as well as my in-laws Robert and Kimberly Bell for believing in me.

Finally, I would like to thank my wife, Jennifer. She has always there through the easy and the difficult times with limitless love and patience. I can't imagine having done this without her by my side.

May 13, 2014

ABSTRACT

RUN-TIME COMPILATION AND DYNAMIC MEMORY USE ANALYSIS FOR GPUs

DEREK WHITE, Ph.D.

The University of Texas at Arlington, 2014

Supervising Professor: Ishfaq Ahmad

Powerful Graphics Processing Units (commonly called GPUs) are proliferating rapidly and are becoming a viable choice for a wide range of user applications. For performing computationally intensive tasks on these processors, it is highly desirable to have software tools that can facilitate writing effective programming code while taking advantage of the full potential offered by these processors. Multi-level memory hierarchy, extensive data transfer, and the utilization of a large number of processing cores are daunting challenges in writing code for data-parallel computing tasks. The programming experience becomes more cumbersome due to the significant restrictions imposed by the OpenCL specification including the inability to allocate memory dynamically. The contribution of this dissertation is developing a methodical framework that can assist the programmer in writing efficient code using a modern, expressive programming language combined with creative uses of static and dynamic program analysis that can alleviate some of the challenges mentioned above. Our methodology allows writing codes with object-oriented programming style that can take advantage of the GPU capabilities. Our framework addresses the problem of

dynamic memory allocation and proposes a memory usage analysis that gives programmers the advantage of dynamic memory allocation while operating within the constraints of OpenCL devices. We have also developed a sofware tool that performs static analysis on GPU kernels written in the Scala programming language using the Firepile GPU programming library. The tool computes an upper bound on memory usage and utilizes this bound to pre-allocate memory needed to successfully execute the kernel on the GPU.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

1.1 Thesis Statement

Restrictions of the OpenCL GPU programming model can be overcome with static and dynamic program analysis.

1.2 Motivation

Graphics processing units (GPUs) are increasingly being used to solve general-purpose computational problems. A single GPU can provide hundreds of processors at little cost. They are being used for scientific data analysis, financial applications, digital signal processing, cryptography and other applications Yet, programming these devices remains difficult. Languages such as OpenCL [1] and CUDA [2] allow data-parallel programming of GPUs at the C (or C++) level of abstraction. However, the programming models of these languages are restrictive, prohibiting a number of standard features of object-oriented languages, e.g., recursion, dynamic memory allocation, and virtual method dispatch. In addition, programmers must explicitly manage the movement of data between the host memory and the device and between layers of the memory hierarchy on the device. Achieving optimal performance often relies on subtle details of the programming model. For example, misalignment of memory accesses can degrade performance by an order of magnitude [3].

This chapter introduces Firepile [4], the contribution of a sophisticated library for GPU programming in Scala. Firepile hides details of GPU programming within the library, allowing programmers to focus on the problem they wish to solve. This

library and the research behind program analysis and architecture that supports it is the main topic and contribution of this dissertation. The library performs facilities for managing devices and memory. Collections classes support data-parallel, functional operations. Performance of Firepile-generated code is comparable to C/C++ code. The library uses a novel approach of constructing code trees from function values at runtime. From these code trees, Firepile generates OpenCL kernels to run on the GPU. Kernel functions may be written in Scala and may use objects, higher-order functions, and virtual methods. The run-time compiler uses method specialization to translate these problematic constructs into the subset of C accepted by the OpenCL specification. The code tree mechanism provides advantages over explicit staging, as found in Lisp and other programming languages [5, 6, 7, 8, 9], that otherwise might require annotations, special types, or other extensions of standard Scala.

1.3 Background of GPU programming

To help understand some of the challenges on GPU programming, we first sketch the architecture of a GPU and how one is programmed. We illustrate using NVIDIA's latest hardware architecture, Fermi [10] and the OpenCL programming model [1]; however, many of the same issues occur with other GPU architectures and other GPU programming models.

A Fermi GPU consists of up to 16 streaming multiprocessors (SM), each with 32 scalar processors, for a total of up to 512 cores. Threads are executed in groups of 32 called *warps*. All threads in a warp are executed using SIMD instructions on a single SM; that is, all threads execute the same instruction simultaneously. OpenCL abstracts threads and groups of threads into *work items* and *work groups*, respectively. The number of work items in a work group need not match the number of hardware threads in a warp.

A thread (work item) running on the GPU has access to multiple classes of memory. In addition to private, per-thread memory, all threads in a warp running on the same SM have access to shared, per-SM memory. Correspondingly, in OpenCL all work items in a work group have access to shared, *local* memory.[1] Finally, all threads running on the GPU have access to global GPU memory. Typical memory sizes are 64 KB for per-block memory and 256 MB up to 6 GB for global memory. In this architecture, communication between threads is limited. Threads running on the same SM can communicate through per-block memory. Threads running on different SMs could communicate through global memory, but global memory provides no synchronization guarantees: a write performed by a thread running on one SM will not be visible to a thread running at a different SM. OpenCL programmers must allocate data to work items and work groups with all of these restrictions in mind.

1.4 Example: reduce

To illustrate how the GPU architecture is utilized, consider a parallel reduction operation, a common building-block of many parallel algorithms. The `reduce` operation takes a non-empty input array with element type `T` and an associative function $\texttt{f} : (\texttt{T}, \texttt{T}) \to \texttt{T}$. The operation performs a reduction (or *fold*) on the array, computing a single value of type `T` for the entire array by applying `f` to pairs of elements. For instance, reducing an array of numbers with the `+` operation will sum the array, reducing with `max` will compute the maximum element of the array.

Because of the limited communication between threads on a GPU, a reduce operation cannot be written in the same way as it typically would be in a sequential

[1] CUDA and OpenCL use different names to refer to similar concepts. Per-thread memory is called *local* memory in CUDA, but *private* memory in OpenCL. Per-block memory is called *block* memory in CUDA, but *local* memory in OpenCL. We use the OpenCL terminology in this thesis.

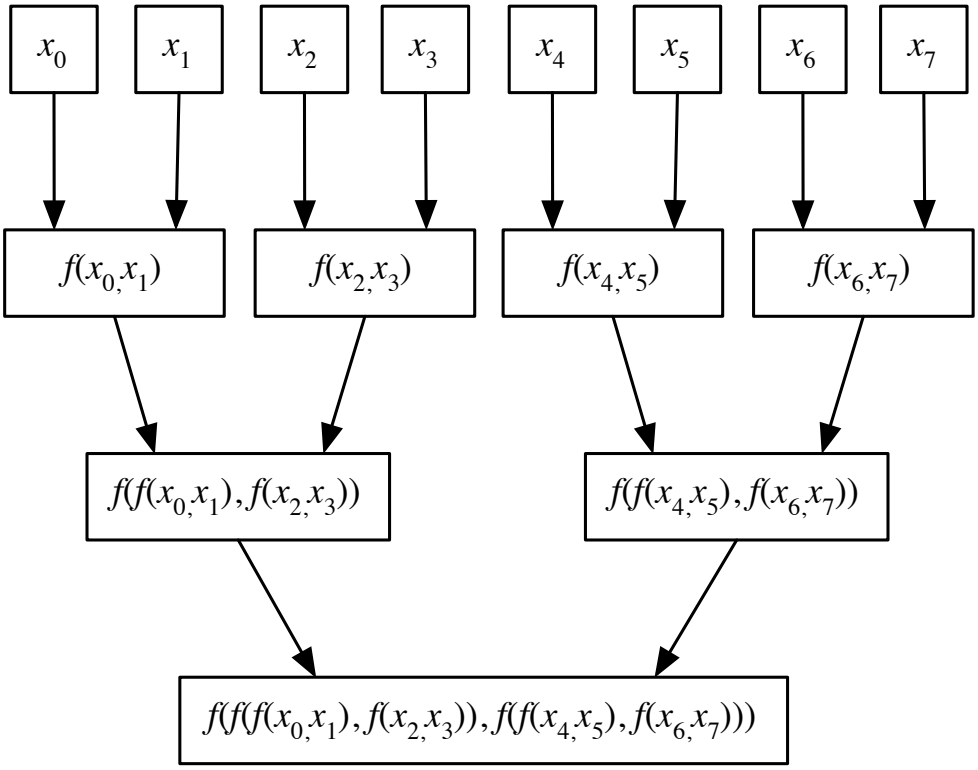


Figure 1.1. Performing a reduction in parallel. Operations in each row can be implemented independently. The output of the previous row is input to the next..

program or in a parallel program to run on multicore processor with shared memory. The developer must keep in mind memory access limitations and the hierarchy of work items and work groups. Global data is partitioned among work groups, and each group can be allocated an area of per-group local memory for its work items to perform their task.

The reduce operation is implemented as a *kernel*. Each work item executes the kernel in parallel on a segment of the input array. The kernel uses work group and work item identifiers to calculate indices into global and local memory arrays. Using this approach, reduction of an array of length n can be computed in parallel $\log_2 n$ rounds, illustrated in Figure 1.1. First, `f` is applied to pairs of elements, producing

$n/2$ intermediate results. Then `f` is applied to pairs of these elements, halving the number of elements again. This process is repeated until one element remains.

To execute on a GPU, the array is first copied to GPU global memory. Then, in the first round, pairs of elements of the array are reduced and stored in a per-group local array. The remaining rounds read from this local array. All SMs thus run in parallel on their own data. Since each round depends on writes to the local memory performed by the previous round, a memory barrier is required to ensure these writes are visible to other threads running in that work group. In the end, each work group produces a single value.

Because writes at one SM are not visible to other SMs, the reduction of these intermediate per-group results cannot be completed on the GPU. Hence, the next step is to write each of the per-group results back to global memory. From here, the results, a much shorter array, are copied back to the host memory, and the reduction is completed on the CPU.

Using OpenCL, before invoking the kernel, the programmer must explicitly allocate storage on the GPU for the input, output, and local arrays. The programmer must also explicitly copy the input to the GPU's global memory and copy back the output. In addition, because C does not support first-class functions and because OpenCL does not allow dynamic dispatch, a new version of reduce must be written for each function `f`. This code duplication can be avoided by using the C preprocessor, but this is potentially error-prone.

1.5 Reduce in Firepile

Using Firepile's `GPUArray` class, to compute the sum of an array `A` via parallel reduction, one simply calls:

```
A.reduce(0)(_+_)
```

Figure 1.2 shows the implementation of `reduce` in the `GPUArray` class. The code is based on an example from the NVIDIA OpenCL SDK.[2] The implementation of `reduce` handles partitioning of the problem space and specifying how the operation should be parallelized. This implementation is built on top of a lower-level device library [11], handles the details of data movement to and from the GPU, and compiles Scala code into OpenCL kernels, as described in Section 2.3.

The `reduce` method starts by partitioning the problem space by rounding the input array size up to the next power of two and dividing it into blocks of equal length. Each work group will work on a different block, and each work item is responsible for computing a single output or intermediate result. Each work group thus accesses one segment of the array, performing a reduction into a local array.

The call to `space.spawn` invokes the reduce kernel on the GPU. The `spawn` method takes a call-by-name parameter. The block of code containing the kernel implementation is passed as a closure into `spawn`. The `spawn` method then compiles this block at run time into an OpenCL kernel.

Bytecode of the block passed to spawn is loaded into Soot for further examination. Case classes for Soot units representing bytecode expressions are matched for the generation of code trees that will later be translated into OpenCL C code for execution on the device.

The compiler has access to the environment of the block, allowing it to specialize the code based on the run-time values `z` and `f`. For instance, in the call to `reduce` above, the compiler can generate a specialized version of `reduce` using the `+` operator.

Although the GPU itself does not support dynamic memory allocation, the programmer can allocate memory within code passed to `spawn`. The compiler identifies array and object allocations within the kernel body and generates code to run *on*

[2] `http://developer.nvidia.com/opencl-sdk-code-samples`.

the host to allocate sufficient storage in GPU memory before the kernel is invoked. When the kernel is invoked, the `input` array is copied to the GPU. When the kernel completes, the `output` array is copied from the GPU into host memory. Unlike with the C API for OpenCL, the programmer does not need to specify the kind of memory (global, local, etc.) to allocate. Instead, the Firepile compiler analyzes the code to determine where a given buffer should be allocated. Movement of data between the host and GPU device are also handled by the library.

Developers can use the `GPUArray` library to program at a high-level or can implement their own kernels using the lower-level mechanisms provided by Firepile. Even at this lower level, tedious details of data movement and memory allocation are handled by the library rather than by the kernel developer.

1.6 Memory Use Analysis

A novel static memory use is described to overcome a serious restriction of OpenCL by allowing programmers to dynamically allocate memory on a GPU. We implement our memory use analysis on Firepile [4], a library for GPU programming in Scala. Like other resource consumption analyses [12, 13, 14, 15], our analysis operates at the bytecode level. Information about allocation points, locations within bytecode where objects and arrays are created, is collected. Loops containing allocation points are reconstructed into an AST that allows them to be easily solved for a loop complexity expression, assisting in summing up the total number of allocations performed at an allocation point. A simple escape analysis is performed that aids in determining which level of memory should be used for heap space. Finally, runtime information is substituted for unknown values in our complexity expression, combined with types sizes and number of thread executions required for program execution to

automatically pre-allocate the suitable amount of memory as a heap buffer for each allocation point.

```
def reduce(z: A)(f: (A,A) => A)
          (implicit dev: Device): A = {

  val input = this.array

  val n = input.length

  // partition the problem by padding out to the next
  // power of 2, and dividing into equal-length blocks
  val space = dev.defaultPaddedBlockPartition(n)

  // spawn the computation on the GPU, returning
  // an array with one result per block
  val results = space.spawn {
    // allocate output in global storage
    val output = Array.ofDim[A](space.groups.size)

    // for each block, in parallel
    for (g <- space.groups) {
      // allocate temp in per-block storage
      val temp = Array.ofDim[A](g.items.size)

      // for each thread, in parallel
      for (item <- g.items) {
        val i = item.id
        val j = g.id * (g.items.size * 2) + i

        // do the first round of the reduction into
        // per-block storage
        temp(i) = if (j < n) input(j) else z

        if (j + g.items.size < n)
          temp(i) = f(temp(i), input(j + g.items.size))

        // make sure subsequent rounds can see
        // the writes to temp
        g.barrier

        // do the remaining log(n) rounds
        var k = g.items.size / 2

        while (k > 0) {
          if (i < k)
            temp(i) = f(temp(i), temp(i + k))
          g.barrier
          k /= 2
        }

        // copy the result back into global memory
        if (i == 0)
          output(g.id) = temp(0)
      }
    }

    output
  }

  // finish the reduction on the CPU
  results.reduceLeft(f)
}
```

Figure 1.2. Reduce in Firepile.

CHAPTER 2

Firepile Run Time System

2.1 Code Trees

In this section we describe the mechanism by which Firepile translates Scala code into OpenCL kernels to run on the GPU. The key idea is to create code trees from run-time function values. These trees can then be analyzed by the program and, in the case of Firepile, compiled into OpenCL C code.

The `spawn` method introduced in Section 1.5 serves as the entry point for the translation mechanism of Firepile. The method translates its argument, a function value, into an abstract syntax tree representing the function. These abstract syntax trees are implemented using tree classes based on the `scala.reflect.generic.Tree` subclasses in Scala standard library. The standard library classes were not used directly since using them requires implementing a large number of abstract classes and methods that were not needed.[1]

Code trees are constructed from function values by locating the JVM bytecode for the function, loading and parsing it, reconstructing Scala type and symbol information, and then finally creating the trees from the bytecode instructions.

The Scala compiler translates function values into anonymous Java objects with an `apply` method. This translation is illustrated for the function `(x: Int) => x + a` in Figure 2.1(a). Here, `a` is a local variable captured by the closure. Function invocation is translated into a call to the `apply` method. Variables captured by the function body are represented in bytecode as fields of the function object. Uses of these captured

[1] `http://github.com/dubochet/scala-reflection/wiki`.

```
public final class A$$anonfun$m$1
  extends scala.runtime.AbstractFunction1
{
  private final int a$1;
  public int apply(int x) { return x + this.a$1; }
  public A$$anonfun$m$1(int a) { this.a$1 = a; }
}
```

(a) Java translation of `(x:Int) => x+a` (simplified)

```
Function(List(LocalValue(_, x, scala.Int)),
         Apply(Select(Ident(LocalValue(_, x, _)),
                      Method(scala.Int.$plus, scala.Int)),
                      List(Select(Ident(ThisType(A$$anonfun$m$1), _),
                           Field(a$1, scala.Int)))))
```

(b) Code tree for `(x:Int) => x+a`

Figure 2.1. Translation to Java and code tree for the function `(x:Int) => x+a` where `a` has been captured from the environment of the function. The Scala compiler generates JVM bytecode similar to the code generated for the Java code in (a). We use Java syntax rather than bytecode syntax since it is more readable. The Scala compiler generates additional methods in the function object, which we elide, to allow it to be used with generics and for type specialization. The code tree in (b) has been simplified to remove temporary variables. _ is used to elide symbols. .

variables are translated into field accesses. In the example, the captured variable `a` is translated into the field `a$1`. The Scala compiler translates function creation into instantiation of the anonymous function object, passing the captured variables into the function object constructor, which initializes the fields.

The code tree constructed from the bytecode for `(x: Int) => x + a` is shown in Figure 2.1(b). The code tree nodes are instances of the case classes in Figure 2.2. By using Scala case classes, pattern matching can be done on code trees, making it easier to identify complex code sequences that need special handling in GPU code generation.

Tree class	Description
`Function(formals,body)`	Function tree containing list of formal parameter symbols and body tree.
`ValDef(symbol,rhs)`	Definition of a value represented by `symbol` with `rhs` tree for initialization.
`Assign(lhs,rhs)`	Assign `rhs` expression to `lhs` target.
`Select(qual,symbol)`	Selection of method or field `symbol` to be invoked on `qual`.
`Apply(fun,args)`	Apply a function `fun` to arguments list `args`.
`Method(name,type)`	Symbol for a method with `name` and return type `type`.
`LocalValue(owner,name,type)`	Symbol to represent a local value with `owner`, `name`, and `type`.
`If(cond,tcase,fcase)`	A conditional branch.
`Literal(value)`	A literal `value`.
`Target(symbol,body)`	A branch target with label `symbol` and `body`.
`Goto(target)`	Represents a jump to a target symbol `target`.
`Block(stmts,type)`	Represents a code block containing a tree of statements `stmts` with result type `type`.

Figure 2.2. Selected code tree classes.

2.2 Constructing Code Trees at Run-time

Next, we describe the steps the run-time compiler takes to construct code trees from bytecode produced by the Scala compiler. These code trees form an abstract syntax tree that will later be used to generate OpenCL kernel code.

2.2.1 Loading the Function Bytecode

The first step in constructing the code tree for a function is to locate the bytecode for the function. Given a function `f`, the `java.lang.Class` for `f` can be retrieved

with `f.getClass`. Using the class object, the bytecode is loaded from the classpath into the Soot bytecode analysis framework [16]. The code for the function object's `apply` method is then located.

2.2.2 Type Reconstruction

One challenge with constructing code trees is that type information is lost in translation from Scala to JVM bytecode. Before building trees, an analysis is performed to reconstruct Scala types from the bytecode. Many Scala types are easily inferred due to the Scala compiler directly mapping these to appropriate JVM equivalents. For instance, primitive types such as `scala.Int` and `scala.Float` are compiled into their corresponding JVM types. `scala.Array[T]` is likewise mapped to a `T[]` array. However, many Scala types, e.g., structural types and type parameters of generic types, do not have a corresponding JVM equivalent. In addition, the types of local variables and operand stack temporaries are not represented directly in the bytecode.

The Scala compiler encodes signature information for each method and field as attributes in the class file. These give the Scala types of formal parameters and return types. Using the formal parameter types and the types of any fields and methods a method `m` accesses as a starting point, a simple forward flow analysis can be used to reconstruct the types of `m`'s local variables and temporaries.

2.2.3 Tree Construction

Code trees are generated from the Soot representation of the method bytecode. The translation is mostly straightforward. Scala extractor objects are used to conveniently match various elements of Soot's bytecode representation. The function itself is represented by a `Function` object, shown in Figure 2.2. Most Scala expression or

type forms have a corresponding `Tree` class. Branches are represented using `Goto` nodes; loops are not reconstructed.

Formal parameters and local variables are mapped to instances of the `LocalValue` class. `LocalValue(owner,name,type)` trees contain an owner `Symbol`, a parameter name `String`, and a `Type`. `Symbol` is an abstract class representing a class, type, method, variable, or similar declaration. The `Type` classes correspond to the various types constructors defined in the Scala specification [17].

In Scala, primitive operations such as `+` are actually method calls on values of the primitive classes (e.g., `scala.Int`), unary and binary bytecode instructions are therefore translated into method calls, represented by the `Apply` class. For example, the `iadd` bytecode instruction, which adds two integers `a` and `b` is translated into the node for `a.$plus(b)`:[2]

```
Apply(Select(Ident(LocalValue(_, a, _)),
             Method(scala.Int.$plus, scala.Int)),
      List(Ident(LocalValue(_, b, _))))
```

Figure 2.1(b) shows an example of the tree generated from the function `(x: Int) => x + a`, where `a` is a value captured from the environment of the function. The `Function` tree is generated containing a list of `LocalValues` for the parameters to the function. A method of type `scala.Int` representing `+` is then applied to the field `a` of class `ThisType(A$$anonfun$m$1)`, which is the type of the function itself.

2.3 The Firepile Compiler

The Firepile library provides collections classes to perform data-parallel operations on a GPU. We expect most users to write programs that use these classes. However, to implement these collections classes, and to provide more control over

[2]For conciseness, we elide some sub-expressions by writing _.

GPU resources, the library also provides lower-level classes and methods for device and memory management, as well as a compiler for translating Scala functions into kernels to run on the GPU. In this section, we focus on the Firepile compiler and in particular how the code trees described in Section 2.1 are used to implement the compiler.

2.3.1 Writing a Kernel

Consider again the `reduce` example from Figure 1.2 in Section 1.5. An input array of element type `A` is passed into the `reduce` method along with a reduction operation `f` and an initial value `z`. The method is written in an explicitly parallel style. The problem space is mapped onto a one-dimensional grid of work items, one for each element of the input array, padded out to the next power of two (line 8). Each work item corresponds to a single thread on the GPU. The grid of work items is then partitioned into a set of work groups, each of which corresponds to a set of threads that execute on the same streaming multiprocessor (SM) on the GPU. All work items can access global memory. Each work group has its own local memory that can be used to share data between work items in the group.

The method then spawns a computation to run on the GPU (lines 14–56). On the GPU, an `output` array is allocated (line 16) into which the result will be written. In the generated code, the call to allocate memory on the GPU is actually performed on the CPU. Each work group will write to one element of `output`.

Next, each work group executes code in parallel with the other work groups (lines 19–53). Each group creates an array of the group size (line 21) in the local memory for that group. Again, in the generated code, the actual allocation runs on the CPU. This array will be used to share partial results between work items in the group. Next, each work item performs the same computation in parallel on different

segments of the `input` array (lines 24–52). First, two elements of `input` are reduced into one element of the `temp` array. Then, $\log_2 |group|$ additional reduction operations are performed on the `temp` array. After each reduction operation, a barrier is executed to ensure that other work items in the group observe the writes to the array. Barriers on local memory are the only synchronization operation on the GPU supported by OpenCL. These operations put the reduced value for the work group's segment of the `input` array into `temp(0)`. The first work item in the group copies this partial result to the output array (lines 50–51), which is then returned to the CPU (line 55). Finally, the partial reduction results are reduced sequentially on the CPU (line 59).

The `spawn` function invokes the run-time compiler on the code block that is passed into it. It has the following signature:

```
def spawn(block: => Unit): Unit
```

The formal parameter type `=> Unit` indicates that `block` is a call-by-name argument of type `Unit`. Passing by name allows `spawn` to compile and run `block` as an OpenCL kernel rather than having `block` execute on the CPU. The block passed to `spawn` is responsible for assigning computation to the appropriate work items and work groups to be run on the GPU. In general, the block consists of one or more nested loops over work groups and work items. Any data declared in the scope of the group loop (line 19) but outside of the items loop (line 24) is treated as local memory to the work group and can be shared among the work items. Only array or variable declarations are expected in this code block since statements can only be executed by work items. The run-time compiler verifies that the code follows the expected pattern. The `spawn` method compiles the body of the work-item loop into a kernel. The kernel is specialized on function values captured by the block. The results of the compilation is memoized so that the next invocation of `spawn` on the same code block, with a compatible environment, does not recompile the code.

2.3.2 The Run-time Compiler

The Firepile compiler generates OpenCL kernels using code trees described in Section 2.1. The `spawn` function first constructs a code tree for its code block argument as outlined in Section 2.1. The trees are then compiled into C through a recursive translation function. The C code is, in turn, compiled by the JavaCL [11] library into a binary to be executed on the GPU.

Variables captured by the block passed to `spawn`—for instance the `input` array in Figure 1.2—are compiled into function parameters in the generated code. When the kernel is invoked, these parameters are initialized by copying data from the CPU into the GPU's global memory.

The compiler assumes a closed world: any classes that will be used in the generated code are assumed to be available during compilation. Since the compiler is often invoked immediately before the kernel is run, this assumption usually holds. Any methods invoked by the kernel function being compiled are also compiled into native code. Any types referenced by the code block are also translated.

Built-in Scala data types are translated into C primitive types (e.g., `scala.Float` is translated into `float`). Arrays are translated into a two-word struct containing a field for the array length and a pointer to a buffer containing the array elements.

Because virtual dispatch is not supported by OpenCL, dispatch tables are not generated for each class; instead, objects are translated into a tagged union: that is, an object of class `C` is compiled into a struct containing a one-word type tag and a union of all possible subclasses of `C`. These tags will later be used to simulate virtual dispatch as described in Section 2.3.4. Methods are translated into C functions that take an explicit `this` parameter as their first argument.

Rather than treating first-class functions like regular Scala objects, they are instead handled specially. Since the compiler has access to the run-time environment

```
struct Object {
 int __id;
};

struct A {
 struct Object _Object;
 int x;
};

struct B {
 struct A _A;
 int y;
};
```

```
union Object_sub {
 int __id;
 struct Object _Object;
 struct B _B;
 struct A _A;
}

union A_sub {
 int __id;
 struct B _B;
 struct A _A;
};

union B_sub {
 int __id;
 struct B _B;
};
```

Figure 2.3. Firepile class translation.

of the code it is compiling, it can often identify the actual function values passed into a method and will then generate a specialized version of the method for that value. For instance, given the call `A.foldLeft(0)(_+_)` the compiler will generate a specialized version of `foldLeft` that returns the sum of the elements of `A`. If the function value has captured variables, the function's environment is translated into a C struct containing fields for each of the captured variables. This structure is passed into methods that take the function as an argument.

2.3.3 Class Translation

Figure 2.3 illustrates how the following Scala classes are translated.

```
class A(val x: Int) { ... }

class B extends A(val y: Int) { ... }
```

Each class is translated into a struct and a union. The struct for class `C` represents an object of exactly that class. Each struct begins with the struct for its immediate

superclass. The built-in `Object` class is translated into a struct with a type tag used to implement method dispatch, as described in the next section. The union generated for `C` is the union of the structs for all subclasses of `C`, plus the type tag. The union is used whenever a value that could be any subclass of `C` is needed. A source-language variable with type `C` is translated into a variable of the union type.

2.3.4 Expression Translation

When generating C code, the compiler performs pattern matching on the code trees. It first identifies certain known expressions that need to be handled specially on the GPU. These include accesses to the work group and work item identifiers, which are compiled into library calls in the generated kernel. Calls representing primitive operations like `+` are translated into the appropriate operation. The Firepile library also provides utility classes such as unsigned integers (useful when porting C code to Scala) and math operations analogous to those provided by the OpenCL math library. These are also handled specially by the compiler.

Since the GPU does not support virtual calls, these must be translated into nonvirtual calls. First, the compiler enumerates the possible receivers of the method to determine if it is monomorphic. This is done by first checking the modifiers of the class and the method for a `final` declaration. If the class and method are not `final`, the known subclasses are searched to determine if they override the method. If the method call is found to be monomorphic then the called method is recursively translated and a nonvirtual invocation is generated. If the call is not monomorphic, a call is generated to a dispatch method, which performs a switch on the object's type tag to invoke the appropriate method nonvirtually.

Consider a call to a non-monomorphic method `m` that could be implemented in either of classes `A` or `B` from Section 2.3.3.

```
int A_m(A_sub* _this, int _arg0) { ... }
int B_m(B_sub* _this, int _arg0) { ... }

int dispatch_A_m(A_sub* _this, int _arg0) {
 switch (_this->__id) {
   case A_ID: {
     return A_m((A_sub*) _this, _arg0);
   }
   case B_ID: {
     return B_m((B_sub*) _this, _arg0);
   }
 }
}
```

Figure 2.4. Dispatch method for a method `m` with two possible implementing classes.

```
val a: A = ...

a.m(10)
```

The call is translated into the following call to the dispatch function in Figure 2.4:

```
A_sub* a = ...;

dispatch_A_m(a, 10);
```

The types `A_sub` and `B_sub` used in the figure are defined in Figure 2.3.

A potential performance issue that can arise from simulating polymorphic calls is *warp divergence* when performing a method call over a collection of mixed types. Depending on the dynamic type of an item in a collection, a given method call may be dispatched to different functions in different threads executing within the same warp on an SM. Because of the SIMD execution model, the different execution paths will be serialized, resulting a slowdown. A possible optimization is to split the collection into separate arrays based on their run-time type, thus making the calls monomorphic and avoiding the warp divergence. Rearranging the collection to clustering elements of the same type would have a similar effect. We plan to support these optimizations in later versions of Firepile.

The compiler also handles array and object allocation specially. Since dynamic allocation cannot be performed on the GPU, all memory used by a kernel on the GPU must be pre-allocated before the kernel is invoked. Memory is allocated in one of the levels of the GPU memory hierarchy: global, local, and private. It must also be determined whether a given global array is to be used for input, output, or both. Most object allocation is simply rejected by the compiler. The kernel can only access objects passed into the compiler. The compiler identifies array allocations and based on their scope determines in which class of memory to allocate the array before the kernel is invoked. If the array does not escape the scope of the kernel's work-item loop, it is allocated in per-thread private memory; if it does not escape the scope of the kernel's work-group loop, it is allocated in per-group local memory. Otherwise, it is allocated in global memory. Since all memory used by the GPU must be pre-allocated before the kernel is invoked, allocation within loops that run on the GPU are prohibited. An allocation site in a given scope must dominate the exit of that scope. In addition, if the size of the array depends on a value computed by the kernel itself, the allocation is rejected. We plan to implement a more thorough memory analysis in the future, computing the memory requirements of the kernel as functions of the kernel's formal parameters.

Other expression types such as basic control constructs are translated in a straightforward manner. Exception throws are rejected by the compiler; exception handlers are simply elided from the generated code.

2.4 Related Work

Work related to this chapter intersect areas of meta programming as well as alternative GPU programming approaches. These works are discussed in the following subsections.

2.4.1 Meta Programming

Lisp [5] was the first language to introduce meta-programming features. Lisp supports *quasiquoting* [6], which allows programmers to construct code templates with "holes" that can be filled in with concrete values. Various forms of quasiquoting are supported in many languages nowadays, including Scheme [18], Haskell [7], and C♯ [19]. MetaML [8] and MetaOCaml [9] support type-checking of quoted code. Scala has some experimental quasiquoting support in the class `scala.reflect.Code`. The Mnemonics [20] library uses this feature to generate bytecode from function values—the inverse of our code trees.

A key difference between our approach and quasiquoting is that our code trees are constructed at run time from function values. With quasiquoting, code trees are constructed statically by the programmer. By constructing trees statically, library writers who want access to the code of functions passed into the library must require that code trees, rather than functions, be passed into the library. Consider the `reduce` function from Figure 1.2. Rather than passing in a function of type `(A,A) => A`, the programmer would instead pass in a code tree, e.g., of type `Code[(A,A) => A]`. This exposes details of the implementation of the library to the caller. If the call to a function like `reduce` is hidden under a stack of other functions, then the use of code trees is exposed further.

Java [21] and other JVM languages also support code introspection or *reflection* features. These allow the programmer to inspect objects at run time and to access their members without having static knowledge. Firepile's code tree construction extends this feature by allowing introspection on the implementation of an object or function.

Bytecode instrumentation is an another approach we considered taking with Firepile. Java 6 supports *agents* in the `java.lang.instrument` package. Agents

are loaded into the VM to intercept class loading and can rewrite classes as they are loaded. They can be used to implement new language features. For example, Deuce [22] adds software transactional memory support to Java through bytecode instrumentation. Our code trees differ from Java agents in that the program representation is at a higher level. Code trees are closer to the original Scala code than to Java bytecode. The Scala program itself, rather than an external entity, has control over when and how the trees are constructed.

Compiler plugins provide another way to extend the base programming language with new functionality. Compiler plugins extend the base compiler with new semantics by adding new compiler passes. For instance, the ScalaCL compiler plugin [23] transforms uses of the standard Scala collections library into OpenCL kernels. The ScalaQL plugin [24] extends Scala with database queries.

2.4.2 GPU Programming

The two most widely used programming models for GPUs are CUDA [2] an OpenCL [1]. Both provide similar abstractions and require explicit management of data movement to and from the GPU as well as between the different classes of memory on the GPU. Firepile generates OpenCL kernels. OpenCL kernels are written in an "extended subset" of C with no support for dynamic memory allocation, function pointers, or recursion. OpenCL C supports additional vector types not found in standard C. Firepile does not support these vector types. Wrappers for both CUDA and OpenCL exist for several languages, including Python [25, 26, 27] and Java [11, 28, 29].

CLyther [30] is an extension of Python with OpenCL support. CLyther provides access to OpenCL APIs like PyOpenCL, allows device memory management, and supports an emulation mode for OpenCL code. Similarly to Firepile, CLyther

performs dynamic compilation of a subset of the base language (viz. Python) into OpenCL kernels.

An alternative for runtime code generation using Scala is to use the Scala compiler's plugin mechanism. The ScalaCL plugin [23] translates Scala code into corresponding kernel code, employing JavaCL wrappers for execution. Originally, ScalaCL allowed kernels to be written using an embedded DSL for specifying parallel computation on GPUs. The compiler plugin now identifies Scala loops that can be parallelized and transforms these to run on the GPU. The ScalaCL feature that performs translation is restricted to be used only with selected operations of its parallel collection library, whereas Firepile attempts to translate as much of Scala as possible and allows users to write root level kernel functions using Scala.

A hybrid compile time/runtime approach is taken by the commercial product from TidePowerd called GPU.NET [31]. GPU.NET enables GPU acceleration of .NET languages including C♯, F♯, and VB.NET. Methods must be annotated as kernels and kernel code generation from .NET bytecode (CIL) is performed behind the scenes and embedded inside assemblies. Memory transfers and scheduling are all performed in the background and programmers need not have any knowledge of the GPU architecture. Runtime plugins determine how the final assembly will be executed on available hardware, or executed on the CPU as a fallback option.

Accelerator [32] is a library for use with multicore CPUs and DX9 GPUs in order to increase performance of parallel code (array processing) execution. Intended for use with .NET, Accelerator programs are typically written in F♯ or C♯ 4.0, although using unmanaged C++ remains an option. Like GPU.NET, Accelerator supports multiple .NET languages.

Aparapi [33] is an API for AMD GPUs that allows the expression of data-parallel workloads in Java. Aparapi translates a subset of Java into OpenCL code.

The subset is restricted to allow only primitive data types and one-dimensional arrays. In addition, primitive scalar fields are read only, static field support is limited, arrays cannot be passed as method arguments, nor can their lengths be accessed. Static methods, method overloading, recursion, and object allocation are all unsupported.

Chafi et al. [34] introduce Delite, a framework for parallelization of DSLs that can use Scala ASTs as their base. Delite performs parallel optimizations and data chunking, and allows for translation to C++ for execution on target systems. Delite includes classes that can be used to specify parallel execution patterns such as Map, Reduce, ZipWith, and Scan. Kernels can be generated from these classes and an optimized execution graph that is executed by the Delite runtime.

Functional languages are a natural choice for data-parallel programming on (or off) GPUs. Nikola [35] is a first-order language for array computations that is embedded in Haskell and is compiled to CUDA. Low-level details such as data marshaling, size inference of buffers, management of memory, and loop parallelization are handled automatically. The quasiquoting feature of GHC [7] is used in translation to CUDA code and allows CUDA code to be written in as a Haskell program. Nikola supports both compile-time and run-time code generation.

Lee et al. [36] demonstrate GPU kernels embedded in Haskell as data-parallel array computations, mixing CPU and GPU computations while taking advantage of the type system to avoid some of the constraints associated with GPU architectures. The domain specific language used to write kernels is restricted to what can be compiled to CUDA. A run-time compiler `GPU.gen` translates the DSL into CUDA code and dynamically links it to the Haskell program using the Haskell plugins library.

There are several dedicated languages for GPUs and other accelerators. CUDA [2] supports GPU programming through an extension of C++. The Brook language [37] extends the C with data-parallel constructs for stream programming on GPUs. Ker-

nels are mapped to Cg shaders by the source-to-source compiler and the Brook runtime handles kernel execution. The Liquid Metal system [38, 39] introduces the Lime programming language and runtime for acceleration designed to be executed across many architectures, including CPUs and FPGAs. Unlike Firepile, Liquid Metal requires the use of a special purpose language for programming accelerators in order to be more adaptable to data-parallel programming (functional, stream computing, bit-level processing, etc).

CHAPTER 3

Memory Use Analysis

3.1 Introduction

This chapter describes a static memory use analysis that overcomes one of the most serious restrictions of OpenCL by allowing programmers to dynamically allocate memory on a GPU. We implement our memory use analysis on Firepile [4], a library for GPU programming in Scala. Like other resource consumption analyses [12, 13, 14, 15], our analysis operates at the bytecode level. Information about allocation points, locations within bytecode where objects and arrays are created, is collected. Loops containing allocation points are reconstructed into an AST that allows them to be easily solved for a loop complexity expression, assisting in summing up the total number of allocations performed at an allocation point. A simple escape analysis is performed that aids in determining which level of memory should be used for heap space. Finally, runtime information is substituted for unknown values in our complexity expression, combined with types sizes and number of thread executions required for program execution to automatically pre-allocate the suitable amount of memory as a heap buffer for each allocation point.

Our approach of using static analysis to perform pre-allocation rather than building a memory allocator for execution on the GPU itself has the advantage of avoiding significant management and synchronization overhead. While advances in performance have been made over the standard memory allocator included in recent versions of CUDA [40], keeping sections of code intended for execution on a GPU device as simple as possible is necessary for achieving the maximum performance

benefit that such an accelerator can provide while keeping the chance of introducing difficult to catch bugs at a minimum.

3.1.1 GPU Challenges

We will begin by sketching the architecture of a GPU and how one is programmed in order to give an overview of the challenges faced when programming for these devices. For this discussion we are using NVIDIAs latest hardware architecture, Fermi [10] and the OpenCL programming model [1]; however, many of the same issues occur with other GPU architectures and other GPU programming models.

A Fermi GPU may contain up to 512 cores divided over 16 streaming multiprocessors (SM), each with 32 scalar processors. Threads are executed in groups of 32 called warps. All threads in a warp are executed using SIMD instructions on a single SM; that is, all threads execute the same instruction simultaneously. OpenCL abstracts threads and groups of threads into work items and work groups, respectively. The number of work items in a work group need not match the number of hardware threads in a warp, although a mismatch can introduce a significant performance penalty through underutilized resources.

Work items (threads) running on the GPU have access to multiple types of memory. In addition to private, per-thread memory, all threads in a warp running on the same SM have access to shared, per-SM memory. Correspondingly, in OpenCL all work items in a work group have access to shared, local memory. Finally, all work items running on the GPU have access to global GPU memory. Typical memory sizes are 64 KB for local memory and 256 MB up to 6 GB for global memory. Communication between threads is limited under this architecture. Threads running on the same SM can communicate through per-block local memory memory. Threads running on different SMs could communicate through global memory, however, global memory

```
1  def foo(k: Int) = {
2    for (group <- space.groups) {
3      val n = k * k
4      val A = new Array[Float](n)
5
6      for (item <- group.items) {
7        for (i <- 0 to n) {
8          val B = new Array[Float](n + i) // k*k + k*k
9
10         for (j <- 0 to n + i)
11           B(j) = math.sqrt(item.globalId).toFloat
12
13         A(i) = B(i)
14       }
15     }
16   }
17 }
```

Figure 3.1. Example of two array allocations, one within a loop..

provides no synchronization guarantees: a write performed by a thread running on one SM will not be visible to a thread running at a different SM. OpenCL programmers must allocate data to work items and work groups prior to execution beginning on the GPU with all of these restrictions in mind.

3.1.2 Motivating Example

Figure 3.1 shows a Scala method containing a work item code block that allocates data dynamically. Two arrays of floating point numbers are allocated, one of which is contained within a loop body.

The goal of this work is to locate memory allocations and the size of memory needed for those allocations within code that is intended for translation and execution on the GPU device. To do this the analysis must calculate the number of allocations for object and arrays, even if those allocations take place in loops or if ar-

rays are created using values that are not known until runtime. It is also necessary to make decisions about where memory can be reused and at what level of the memory hierarchy is best to provide heap space for allocations.

To allocate array A we must track the origin of the array size expression n. The analysis will determine that the value of n comes from k*k. Following the origination of k it is found to be an unknown value at compile time. A symbolic expression representing the bound of the array will be generated and runtime values for k substituted to evaluate the expression k*k for a concrete array bound. An array of float type using the concrete array bound as its size will be supplied to the kernel by Firepile at runtime. Firepile finds that array A is declared inside of the work group loop but outside of the work item loop, it will be allocated once per work group as an array in local memory space.

Array B provides a more interesting example of array allocation since the allocation takes places inside of a loop body within the work item. As with array A, the array size expression is extracted and origin of variables is tracked. The value for n comes from k*k but the value for i comes from the upper bound of the loop since it is the loop iteration variable. Solving for the amount of memory required by array B will involve finding the complexity of the loop from 0 to n (or 0 to k*k) as well as the upper bound on n+i (or k*k + k*k). Since the array allocation of size k*k + k*k is nested within the loop from 0 to k*k, the number of allocations would typically be considered to be (k*k)*(k*k+k*k). However, since B does not escape the loop, memory will be reused over loop iterations resulting in heap space being allocated for only one array of size k*k+k*k.

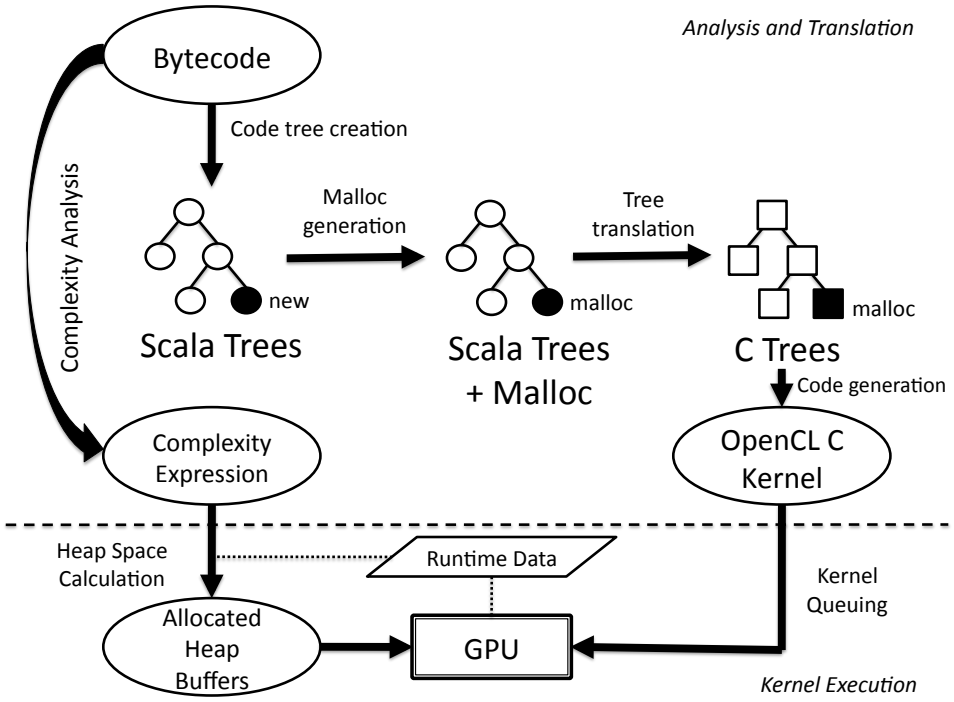


Figure 3.2. Phases of memory analysis and translation..

3.2 Approach

An overview of our approach can be seen in Figure 3.2. Memory analysis and translation is shown above the dotted line and later phases that occur just before kernel execution are shown below. Code tree generation and translation are performed primarily by the Firepile library with the addition of the memory allocation (malloc) generation phase.

Code tree generation recovers a tree representation (Scala code trees) of original Scala work item code from bytecode. Instructions involving dynamic data allocation (allocation points) are replaced with calls to a generated memory allocation function producing a tree representation that includes new calls to the generated function. A tree translation phase takes the Scala code trees with newly added memory allocation

function calls and produces a C-like AST suitable for generation of kernel code in the C99 language subset known as OpenCL C. Kernel code is then passed to the device driver of a GPU device supporting OpenCL. The kernel is compiled by the driver and execution can be queued using the OpenCL API.

The goal of complexity analysis in our approach is to collect loops and array allocations from bytecode and generate a symbolic complexity expression that can later be evaluated for the number of allocations performed at an allocation point. We use ABC [41] to compute symbolic expressions for loop bounds and synthetic loops for array allocations. Synthetic loops are nested inside of the representation of an outer loop when array allocations are contained within a loop. This allows the resulting symbolic complexity expression to easily capture the number of data elements required to store multiple allocations of an array. Symbolic complexity expressions are a parametric representation of the upper bound on the number of executions for given loop or nested loops. Unknown variables in the symbolic complexity expression are left to be substituted for runtime values. The following is an example of a symbolic expression after evaluating the loop containing allocation of array B in Figure [**?**]:

```
(k*k) * (k*k + k*k)
```

Note that this example symbolic expression does not take into consideration that space used for array allocation can be reused over loop iterations. Our algorithm would recognize this fact in the code example shown and eliminate the term for the outer loop leaving the array size expression `(k*k + k*k)`.

ABC was chosen as the library to compute symbolic expressions due to its ability to handle the kind of loops that regularly appear in GPU code. ABC also provides a convenient means of interfacing with its solver through the use of Loop Language Trees (LLTs). Loop Language Trees will be described in Section 3.3.1.

At the time of kernel execution, runtime values are provided for variables in the symbolic complexity expression. The expression is evaluated and combined with data sizes for the type of variable being allocated. Since all work items (threads required to complete the data-parallel task) execute the same kernel, the number of work items is included to arrive at the final heap buffer size. Heap buffer allocation requests are performed on the host to make the needed memory available to the kernel during execution.

3.3 Implementation

Our memory use analysis is implemented as an extension to the Firepile library for Scala programming of GPUs. Using a forward flow analysis, information about points of memory allocations and loops contained within the bytecode of a work item are collected into a Loop Language Tree (LLT) representation during the complexity analysis phase. Loop Language Trees are described in Section 3.3.1. LLTs can be solved resulting in *complexity expressions*. Complexity expressions are later used for calculating total memory use. Firepile is used to produce memory allocation functions and runtime information is retrieved in order to evaluate complexity expressions for the size of device buffers to be used as heap space during execution.

3.3.1 Loop Language Trees

Loop Language Trees are an AST representation used to describe loops extracted from bytecode. LLTs provide a means for describing loops containing loop variable initialization, loop condition, loop variable increment, as well as a loop body section. The loop body section is used to include other loops that are nested within the outer loop. An example of the LLT representation for the inner loop found on line 10 of Figure 3.1 is shown:

```
1 ForLoop(Assign(i, 0),
2        LessThan(i, n),
3        Assign(i, i + 1),
4        ForLoop(Assign(j, 0),
5               LessThan(j, n + i),
6               Assign(j, j + 1)))
```

Variables `i`, `j`, and `n` are `IntVar` nodes that represent integer loop variables. `IntVar` and `IntVal` (used for integer constants) are nodes of type `AExpr`, used to represent arithmetic expressions. A mathematical operation such as addition shown above is converted into an `AExpr` node such as `Sum`. Boolean expressions like `LessThan` are nodes of type `BExpr`. Loop language tree nodes are described in Figure **??**

Loop language trees constructed from bytecode are passed to the solver to find the loop complexity expression for the given loop.

3.3.2 Complexity Analysis Algorithm

`allocations: allocation_point --> ABC_complexity`

`heapBuffer: allocation_point --> (memory_region, heap_size)`

The algorithm, shown in Figure **??** starts by examining nodes that contain an instruction to allocate dynamic memory, known as *allocation points*. The first set of allocation points collected are the simple case where allocations do not occur inside of a loop. `collectNonLoopAPs` takes the body of the work item code and returns a list of these allocation points. If the allocation point is found to be creating a new array through an array creation instruction, a loop representation is generated to calculate an array bound. The loop representation is created by using a generated loop variable, a conditional expression based on the size expression extracted from the

Tree node	Description
`ForLoop(init,cond,inc,body)`	Loop representation containing loop variable initialization, loop condition, variable increment, and loop body.
`IntVar(name)`	Definition of an integer value represented by `name`.
`IntVal(const)`	Constant integer value.
`Sum(e1,e2)`	Addition operation on arithmetic expression nodes `e1` and `e2`.
`Sub(e1,e2)`	Subtraction operation on arithmetic expression nodes `e1` and `e2`.
`Mult(e1,e2)`	Multiplication operation on arithmetic expression nodes `e1` and `e2`.
`Div(e1,e2)`	Division operation on arithmetic expression nodes `e1` and `e2`.
`IntDiv(e1,e2)`	Integer division operation on arithmetic expression nodes `e1` and `e2`.
`LessEqual(e1,e2)`	Less than or equal to comparison of nodes `e1` and `e2`.
`LessThan(e1,e2)`	Less than comparison of nodes `e1` and `e2`.
`GreaterEqual(e1,e2)`	Greater than or equal to comparison of nodes `e1` and `e2`.
`GreaterThan(e1,e2)`	Greather than comparison of nodes `e1` and `e2`.
`Equals(e1,e2)`	Equivalence of nodes `e1` and `e2`.

Figure 3.3. Loop language tree nodes.

array allocation, and a loop increment of 1 on the generated variable. Once the loop representation for the single fake loop has been generated, it is passed to the solver and a symbolic complexity expression for the loop is returned. If the allocation point is not creating an array but instead a single class instance, the number of instances at this time is considered to be simply 1 and does not require the generation of a

loop. A map of allocations is maintained that uses the allocation point as a key and the complexity expression as a value.

Next, the algorithm iterates over a set of outer loops that has been collected from the body of work item code. Loops are recognized using a method similar to that described in [42]. For each of these loops the loop variable initialization, loop condition, increment expression, and loop body are extracted. All allocation points are also collected from the loop body. A new LLT is constructed for each allocation point. The scope of a variable (Loop, Method, Returned) holding the allocated space is retrieved from a map that has been populated by a simple escape analysis prior to algorithm execution. If it is found that the allocation point assigns to a variable with Loop scope, an LLT that will result in only a single loop iteration is created since the new variable is overwritten for each loop execution. Otherwise, the new LLT is populated with the information extracted from the loop and the body of the loop is further checked for nested loops by calling `collectNestedLoops`. If the current allocation point involves creation of an array within the loop, an LLT is generated for the array as described previously and appended as the innermost loop. Once again, the LLT is passed to the solver which returns a complexity expression and is then stored by allocation point key into the allocations map.

3.3.3 Calculating Memory Use

To calculate memory use, the size of each memory region is computed by summing up the sizes at each allocation point as shown in Figure 3.5. Heap size is calculated by finding the product of an evaluated complexity expression, the total number of global work items, and the size of storage required for a data type. The complexity expression is evaluated by substituting captured runtime values of variables. It is assumed that all memory regions will be global unless it is found that an

```
allocations <- {}       // Map ( Allocation Point [Soot Unit] -> Complexity Expression )
loops <- collectLoops(body)
nonLoopAPs <- collectNonLoopAPs(body)

function collectNestedLoops(body)
  loops <- collectLoops(body)
  LLTs <- Nil
  if loops.length > 0
    foreach L in loops
      (loopVarInit, condition, incrementExpr, body) <- L
        LLT <- new LoopLanguageTree
      LLT(init, cond, inc, body) <- (loopVarInit, incrementExpr, incrementExpr, collectNestedLoops(body))
      LLTs += LLT
      return LLTs
    else
      return Nil

foreach AP in nonLoopAPs
  if AP contains NewArrayExpr
    generatedLoopVar <- 0
    arraySizeExpr <- LessThan(0,sizeExpression(AP))
    generatedInc <- Sum(generatedLoopVar, 1)
    LLT <- new LoopLanguageTree
    LLT(init, cont, inc, body) <- (generatedLoopVar, arraySizeExpr, generatedInc, Nil)
    allocations += (AP -> solveABC(LLT))
  else
    allocations += (AP -> 1)

foreach L in loops
  (loopVarInit, condition, incrementExpr, body) <- L
  allocationPoints <- collectAllocPoints(body)
  foreach AP in allocationPoints
    LLT <- new LoopLanguageTree

    if scopes(AP) == Loop
      LLT(init, cond, inc, body) <- (loopVarInit, LessThan(init,Sum(init,1)), inc, collectNestedLoops(body))
    else
      LLT(init, cond, inc, body) <- (loopVarInit, condition, incrementExpr, collectNestedLoops(body))

    if AP contains NewArrayExpr
      generatedLoopVar <- 0
      arraySizeExpr <- LessThan(generatedLoopVar,sizeExpression(AP))
      generatedInc <- Sum(generatedLoopVar, 1)
      innerLoop <- new LoopLanguageTree
      innerLoop(init, cont, inc, body) <- (generatedLoopVar, arraySizeExpr, generatedInc, Nil)
      LLT(body) <- LLT(body) += innerLoop
    allocations += (AP -> solveABC(LLT))
```

Figure 3.4. Complexity Analysis Algorithm.

allocation point has been marked with Method scope and that heap space required for the allocation fits within the hardwares amount of local memory. Local variables or arrays that are declared inside of the work item but are declared outside of a loop are marked as MethodScope and become candidates for storage in faster local memory. If a variable is declared inside of a loop, it is marked as LoopScope and

```
scopes <- escapeAnalysis(body) // Map ( Allocation Point -> (Scope [Loop,Method,Returned]) )
heapBuffers <- {}       // Map ( Allocation Point -> (Memory Region, Size) )

foreach (AP, complexityExpr) in allocations
  heapSize <- solveSubsRuntimeVals(complexityExpr) * device.globalWorkItems * typeSize(AP)
  memRegion <- Global

  if scopes(AP) == Method && heapSize < device.localMemSize
    memRegion <- Local

  heapBuffers += (AP -> (memRegion, heapSize))
```

Figure 3.5. Memory Use Algorithm.

can be assigned to the fastest memory region available and says that the memory will be reused across loop iterations. Tracking whether or not a variable belongs to LoopScope is performed through examination of the context in which it was created. Allocation points are associated with tuples containing the memory region and heap size for a buffer through a `heapBuffer` map. This map is later used for marking the memory regions for kernel parameters and allocation of space on the device.

3.3.4 Malloc Generation

A memory allocation function is generated for each allocation instruction encountered during the code tree creation phase. Allocation functions are generated from two code tree templates: one for objects and another for arrays. Functions are generated for each allocation point and are named based on the allocation point identifier, `malloc_<ID>`. The template for creation of code trees for the required functions has placeholders to substitute information for generation of the function name, data type, and array length in the case of arrays. Malloc functions take a pointer to the heap space associated with the allocation point as a parameter and return a pointer to the section of heap space reserved for that call by the current work item.

Unique global IDs assigned by OpenCL for work items during execution are retrieved and used to index into the heap space array passed to the malloc function.

Use of global IDs avoids the need to maintain and synchronize an offset into the heap space for each call by each thread. The function call performs a simple calculation for the offset into a heap by multiplying work item ID by the size of data elements, setting a pointer to the beginning of a section of the heap be returned by the function call.

Each allocation instruction in the code tree is replaced with a call to its corresponding malloc function. A pointer to the heap space held by the kernel environment is passed to malloc since the pointer to memory must be passed down from the top level kernel function. Any other information required by malloc such as the size of data needed by the allocation is contained in the generated function for the associated allocation point.

3.3.5 Heap Buffers and Runtime

Heap buffer space allocated on the device is passed as pointers marked with the required memory region similar to standard kernel array parameters. This pointer to heap memory is added to the kernel environment structure by the kernel function so the caller of malloc can pass it at the allocation point. Buffers on the GPU device must be allocated prior to kernel execution. Once the symbolic expression for the number of instances created at an allocation point and region has been found, runtime values of any remaining variables that exist in the symbolic expression are substituted and the expression is evaluated. Firepiles marshalling system is used to determine the size in bytes of memory required to hold the representation for an instance of the type being allocated. Final buffer sizes are the product of the evaluated symbolic expression for number of instances, type size, and number of global work items. The host then makes a request for device buffer allocations and sets the arguments to the kernel prior to queuing the kernel for execution.

3.4 Evaluation

Our algorithm is evaluated by walking through a calculation of the expected memory that would be required by the example listing in Figure 3.1 as well as an abbreviated ray tracing example kernel shown in Figure 3.6.

We begin with the motivating example assuming a runtime value of `10` for parameter `k` and for simplicity a total number of `128` work items. A constant `WORK_ITEMS` will be used to denote the number of work items for this example execution.

The value for `n` becomes 100:

$$n = k * k \rightarrow n = 10 * 10 \rightarrow n = 100$$

Next, on line `4` a `Array[Float]` is created of size `100` and assigned to `A`. The amount of memory expected for the array `A` is:

$$n * WORK\_ITEMS * sizeof(Float) \rightarrow 100 * 128 * 4 \rightarrow 51200$$

On line `8` `B` is created of type `Array[Float]` contained in a loop on line `7` with an upper bound of `n` since it starts at `0`. Note that the variable `B` is rewritten every loop iteration so it is not neccessary to include the outer loop in the calculation other than using it to find an upper bound on `i`. We have already determined the value of `n`, we will include it in the calculation now by substituting it for i:

$$\begin{aligned} & (n + i) * WORK\_ITEMS * sizeof(Float) \\ \rightarrow & (n + n) * WORK\_ITEMS * sizeof(Float) \\ \rightarrow & (100 + 100) * 128 * 4 \\ \rightarrow & 102400 \end{aligned}$$

```
1  sealed case class Vector(x: Float, y: Float, z: Float)
2  sealed case class Ray(start: Vector, dir: Vector)
3
4  def rayTrace(scene: Scene, rgb: Array[Int], screenHeight: Int,
       screenWidth: Int)(implicit dev: Device) {
5    val space = dev.defaultPaddedPartition(screenHeight)
6
7    val camera = scene.camera
8
9    space.spawn {
10     for (group <- space.groups) {
11       for (item <- group.items) {
12         val y = item.globalId
13         val stride = y * screenWidth
14
15         for (x <- 0 until screenWidth) {
16           rgb[x + stride] = trace(new Ray(camera.pos, getPoint(x, y,
                 camera)), scene, 0)
17         }
18       }
19     }
20   }
21
22 }
```

Figure 3.6. Abbreviated ray tracing kernel showing new Ray objects being allocated inside a loop.

Next we will examine the ray tracing example in Figure 3.6.

3.5 Related Work

Work related to Firepile's dynamic memory allocation features can be split into two categories: GPU memory allocation and memory use analysis. These categories are discussed in the following subsections.

3.5.1 GPU Memory Allocation

The first known memory allocator for use inside of kernel functions targeting GPU devices comes from [43]. Their work introduces a scalable, lock-free memory allocator designed for use with the NVIDIA CUDA architecture. The goal of XMalloc is high-throughput access to a shared memory pool by a highly multithreaded program implemented on data-parallel architectures such as GPUs. As with most efficient GPU programming, an emphasis is made on making the best use of memory traffic and avoiding contention. Memory block metadata is stored in block headers that are atomically modified to avoid the need for locking. Free lists are maintained as lock-free FIFOs using minimal memory operations to enqueue and dequeue items. A hierarchical memory pool is used to aggregate parallel memory requests into larger requests in order to increase throughput to the allocator, these larger memory chunks are then redistributed to the requesters. Our memory allocator avoids the need for such a strategy since heap space is determined prior to kernel execution and heaps are divided into sections accessed by associated threads. The programmer calls an initialization function that sets up a section of global GPU memory to act as heap space. Requests to allocate memory are made through a malloc function and memory can be deallocated using free.

NVIDIA has added support for dynamic memory allocation inside GPU kernels [2] since the introduction of CUDA 3.2. Later work [40] by the XMalloc authors focuses on the use of their technique to outperform CUDAs memory allocator through allocation coalescing and efficient queues. Firepile is implemented using the portable OpenCL standard supported by all major graphics device manufacturers including NVIDIA, AMD, and Intel allowing it target a wider range of devices. OpenCL does not currently support dynamic memory allocation.

Hong [44] implements a lightweight memory allocator to support the intermediate results created by their MapReduce framework for GPUs. A large block of global device memory is used for heap space. The maximum size of heap space allocated can be determined fairly easily due to the fact that memory usage patterns are fixed in MapReduce applications. A free space pointer into the global buffer is maintained and incremented by CUDAs atomicAdd() operation. Each warp, a group of 32 threads scheduled together, has a buffer block. Organizing memory into buffer blocks assigned to warps allows for more coalesced memory accesses. Information required for maintaining buffer blocks is kept in faster local memory to reduce overhead of allocation.

Another Map Reduce framework designed for GPUs described by [45] takes a less sophisticated approach by allowing programmers to annotate CUDA code with comments specifying dynamic allocations for arrays and what runtime variables are used to determine sizes.

Ramamurthy [46] instruments benchmarks to obtain the maximum memory needed to create linked list and hash tree data structures dynamically on a GPU. Data structures are stored in pre-allocated arrays on a per-type basis. Each array has a counter variable that is incremented atomically for each dynamic memory allocation request to ensure every thread received a unique position in the arrays. Counters are reset each iteration allowing heap space to be reused. Firepiles dynamic memory allocation avoids the need for maintaining counters by using unique global thread IDs to index into heap space arrays. Similar to [46], we reuse memory by determining when a variable is being reallocated by consecutive loop iterations.

3.5.2 Memory Use Analysis

In general, heap space analysis has the goal of finding bounds on heap space consumption of programs. [12] describes a static heap space analysis that operates at the byte code level, the same level that Firepile analysis and translation is performed, and generates heap space cost relations producing heap consumption from a program as a function of data size. Escape analysis is used to take into account space that is de-allocated by the garbage collector, producing improved cost relations. Control flow graphs (CFGs) are built from analyzing the structure of Java bytecode programs. An intermediate representation of the CFGs is constructed and information about changes in size of data structures between blocks is inferred. Cost relations are generated from the intermediate representation and information collected about size changes, resulting in a set of heap space cost equations and size relations corresponding to method entries or blocks in the control flow graph. Finally, cost equations are simplified to obtain an upper bound in closed form for the cost relations.

Another low-level approach that operates on assembly-level programs is proposed by [47]. A Presburger solver captures symbolic memory requirements and fixpoint analysis serves the analysis of loops and recursion. Also operating at a bytecode or assembly-level, [15] discusses a general resource usage analysis to infer upper bounds on the cost of user-defined resources through annotations to Java code. User-defined resources can include things such as the number of SMS packets sent by a mobile phone application or cost can be defined in terms of memory consumption, which is of interest to our work.

Static memory use analysis is used in verification of memory consumption policies for devices with limited resources such as smart cards, mobile phones, and other embedded devices [14, 13, 48]. [13] depends on code annotations to verify memory consumption of bytecode but includes an annotation assistant to aid in the creation

of annotations. [14] addresses the problem of providing an analyzer that can execute on a Java Card. [48] discusses the application of static analysis to compute an upper bound on dynamically allocated memory by synthesizing non-linear formulas and estimating memory use as a function of parameters to methods. This work is not specific to Java based languages but instead focuses on object-oriented programming in general and use of object-oriented languages with automatic dynamic memory management in real time and embedded systems.

CHAPTER 4

EXPERIMENTAL EVALUATION AND CONCLUSION

This chapter discusses experimental evaluation of Firepile and concludes the discussion with directions of future research that build on our work.

4.1 Experimental results

To evaluate the performance of Firepile, examples from the NVIDIA OpenCL SDK[1] were ported to Scala to use the library. The original examples are implemented in C++ with kernel code in the OpenCL subset of C with no optimizations beyond what is demonstrated in the example code added. The chosen examples are summarized in Figure 4.1. In addition to Reduce (similar to the code in Figure 1.2), Black-Scholes, matrix–vector multiplication (MVM), the discrete cosine transform (DCT8x8), and matrix transpose were ported. All benchmarks were run with four of the same problem sizes for each benchmark. Command-line options for problems sizes were added to the C++ versions when needed. For all benchmarks except MVM, problem sizes increase exponentially; the problem size for MVM increases linearly. The benchmarks were chosen to not exceed the capabilities of the current Firepile implementation. For instance, benchmarks with OpenCL vector operations were not ported since Firepile does not yet support these operations.

We compared Firepile against two C++ versions of each benchmark. In addition to the NVIDIA implementation of the benchmark, we constructed a hybrid version by replacing the NVIDIA kernel code with the Firepile-generated kernel. All other code,

[1] `http://developer.nvidia.com/opencl-sdk-code-samples`.

Benchmark	Sizes
Reduce	2^{20}, 2^{21}, 2^{22}, 2^{23}
Black-Scholes	2M, 4M, 8M, 16M
Matrix–vector multiplication (MVM)	12.1M, 13.2M, 14.3M, 15.4M
Discrete cosine transform (DCT8x8)	2^{18}, 2^{19}, 2^{20}, 2^{21}
Matrix-transpose	2^{16}, 2^{18}, 2^{20}, 2^{22}

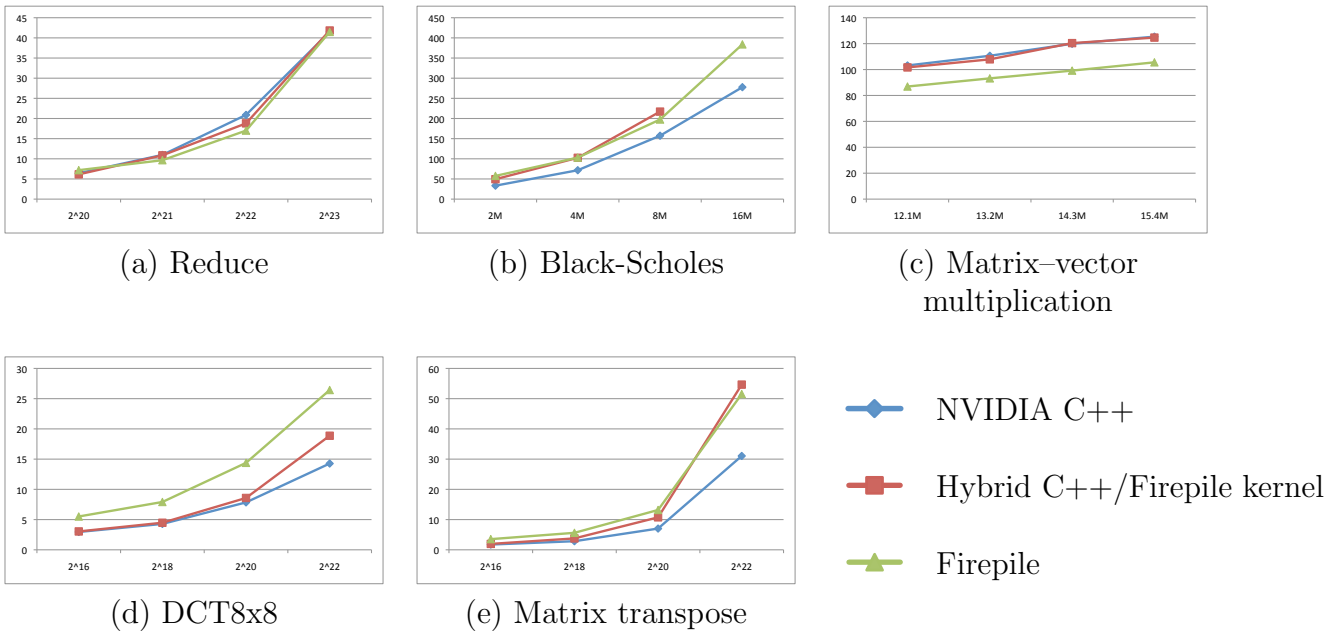
Figure 4.1. Summary of benchmarks run for Firepile and C++.

including the code to copy data to and from the GPU is identical to the original C++ version. This hybrid version is used to determine if performance differences between Firepile and the NVIDIA code can be attributed to the kernel translation or to differences in device initialization or data movement.

The three versions of each benchmark were run with the same data values and range of problem sizes. Experiments were performed on a system with a 3.0GHz Intel Core 2 Quad Q9650 CPU, 8GB RAM, and an NVIDIA GeForce 9800GT graphics card with 512MB of video memory, running Windows 7 Professional 64-bit. Firepile was compiled and run with Scala 2.9.0 and Java 1.6.0_24b07 using the HotSpot VM.

For each problem size and configuration tested, the benchmark was executed 30 times and the results averaged. To reduce interference from the JIT compiler, each kernel execution consisted of a warm-up run followed by 16 repetitions. Warm-up runs were not included in the reported times. Results are shown in Figure 4.2. The hybrid configuration for Black-Scholes for a data size of 16M crashed the graphics card driver on our test system and times were not collected.

Firepile performance compares favorably to the NVIDIA implementation. For most benchmarks, run times for all three configurations were within 15%. The execution times for the hybrid configuration were between the times for the NVIDIA and Firepile configurations, as expected. With most benchmarks the NVIDIA C++ code outperformed Firepile, again as expected. The Firepile version of the Reduce bench-



(a) Reduce (b) Black-Scholes (c) Matrix–vector multiplication

(d) DCT8x8 (e) Matrix transpose

Figure 4.2. Total execution times in ms of each benchmark for different problem sizes.
.

mark performed as well as the NVIDIA and hybrid versions. The Firepile version of MVM was faster than the NVIDIA and hybrid versions. Firepile did not perform as well on DCT8x8, with execution times nearly double the NVIDIA version.

Execution time differences can be attributed to a variety of factors. First, there are differences between the generated kernel and the hand-written kernel. While the algorithms used were the same, Firepile generated kernels introduce additional temporary variables and use goto statements rather than structured control flow statements. In addition, placement and alignment of data differs from the C++ versions. In particular, arrays in the C++ version do not have a length field. The C++ versions of the code can also make use of pointer arithmetic, while Firepile generated kernels do not. The NVIDIA version of Matrix-transpose performed consistently better across all problem sizes. We attribute this difference to the heavy use of pointer arithmetic for data access into global arrays. The Black-Scholes benchmark generated the high-

est number of temporary variables of the benchmarks tested and performed slower than the NVIDIA example as expected. Additional temporaries can increase the memory footprint of a kernel by requiring it to use more private memory. A larger footprint can then reduce parallelism since the GPU can support only a fixed amount of private memory per SM.

There are also differences in data movement between the configurations. Because Firepile arrays have a length field, this additional data is copied to the GPU when the kernel is invoked. Array lengths are copied to the GPU in the hybrid configuration as well. For MVM, Firepile was 15% faster than both the NVIDIA and hybrid configurations, indicating that data copying times were faster. In contrast, the Firepile version of DCT8x8 was consistently slower than the other configurations.

Additional optimizations we have planned for later versions of the library should help to close the performance gap with the native versions of the kernels, while maintaining Firepile's ease of use.

4.2 Conclusion

General-purpose computing on GPUs remains a difficult task. GPUs have a restricted programming model, disallowing features such as dynamic memory management and virtual methods. The Firepile library supports a richer programming model, allowing kernels to be written in Scala and perform dynamic memory allocation. Performance of Firepile kernels is comparable to the performance of native C code.

4.3 Future work

Future plans for Firepile are to support more features of Scala and to explore GPU-specific optimizations in the context of OO languages. We wish to support more complex scenarios where multiple kernels are executed with minimal data movement between the host and GPU. We also plan to explore the use of run-time generated code trees to implement other domain-specific extensions of Scala. Furthermore, it is our intention to leverage forms of automatic parallelization through runtime identification of code portions that are suitable candidates for conversion into kernels and can benefit from execution on an accelerator device. The memory use analysis framework can be extended to support alternate, less conservative and possibly more efficient estimation schemes. We also intend to explore the extension of these techniques to other accelerator devices beyond GPUs as well as other JVM-based languages.

REFERENCES

[1] A. Munshi and Khronos OpenCL Working Group, "The OpenCL specification," 2009.

[2] NVIDIA, "Compute unified device architecture programming guide," `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`, 2008.

[3] ——, "NVIDIA OpenCL best practices guide, version 1.0," `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`, 2009.

[4] N. Nystrom, D. White, and K. Das, "Firepile: run-time compilation for gpus in scala," in *GPCE*, 2011, pp. 107–116.

[5] G. L. Steele, Jr. and R. P. Gabriel, "The evolution of Lisp," in *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*. New York, NY, USA: ACM, 1993, pp. 231–270.

[6] A. Bawden, "Quasiquotation in Lisp," in *Partial Evaluation and Semantic-Based Program Manipulation*, 1999, pp. 4–12.

[7] G. B. Mainland, "Why it's nice to be quoted: Quasiquoting for Haskell," in *Proceedings of the 2007 ACM symposium on Haskell (Haskell '07)*, 2007.

[8] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," in *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations (PEPM)*, 1997, pp. 203–217.

[9] W. Taha, "A gentle introduction to multi-stage programming," in *Domain-Specific Program Generation*, 2003, pp. 30–50.

[10] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Fermi," `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2010.

[11] O. Chafik, "JavaCL: Java wrappers for OpenCL," `http://code.google.com/p/javacl`, 2011.

[12] E. Albert, S. Genaim, and M. Gómez-Zamalloa, "Heap space analysis for Java bytecode," in *Proc. 6th International Symposium on Memory Management (ISMM)*. ACM, Oct. 2007, pp. 105–116.

[13] G. Barthe, M. Pavlova, and G. Schneider, "Precise analysis of memory consumption using program logics," in *Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, Sept. 2005, pp. 86–95.

[14] P. Giambiagi and G. Schneider, "Memory consumption analysis of java smart cards," 2005.

[15] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo, "User-definable resource usage bounds analysis for java bytecode," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 5, pp. 65–82, 2009.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Vijay Sundaresan, "Soot: A Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, 1999.

[17] M. Odersky *et al.*, "The Scala language specification," 2006–2011. [Online]. Available: www.scala-lang.org/docu/files/ScalaReference.pdf

[18] R. Kelsey, W. Clinger, and J. R. (editors), "Revised^5 report on the algorithmic language Scheme," *ACM SIGPLAN Notices*, vol. 33, no. 9, pp. 26–76, Oct. 1998. [Online]. Available: http://www.schemers.org/Documents/Standards/R5RS

[19] ECMA, "Standard ECMA-334: C♯ language specification (4th edition)," http://www.ecma-international.org/publications/standards/Ecma-334.htm, June 2006.

[20] J. Rudolph and P. Thiemann, "Mnemonics: type-safe bytecode generation at run time," in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM)*, 2010, pp. 15–24.

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison Wesley, 2005, iSBN 0321246780.

[22] G. Korland, N. Shavit, and P. Felber, "Noninvasive concurrency with Java STM," in *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-3)*, Jan. 2010.

[23] O. Chafik, "ScalaCL: Faster Scala: optimizing compiler plugin + GPU-based collections (OpenCL)," http://code.google.com/p/scalacl, 2011.

[24] M. Garcia, A. Izmaylova, and S. Schupp, "Extending Scala with database query capability," *Journal of Object Technology*, July 2010.

[25] "PyOpenCL: Python programming environment for OpenCL," http://mathema.tician.de/software/pyopencl, 2011.

[26] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA: GPU run-time code generation for high-performance computing," http://arxiv.org/abs/0911.3456, 2009, in submission.

[27] C. Lejdfors and L. Ohlsson, "Implementing an embedded gpu language by combining translation and generation," in *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*, 2006, pp. 1610–1614.

[28] "JOCL: Java bindings for OpenCL," http://www.jocl.org, 2011.

[29] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA," in *Proceedings of the 15th Inter-*

national Euro-Par Conference on Parallel Processing (Euro-Par '09), 2009, pp. 887–899.

[30] "Clyther: Python language extension for OpenCL," `http://clyther.sourceforge.net`, 2011.

[31] "GPU.NET: Library for developing GPU-accelerated applications with .NET," `http://www.tidepowerd.com/product`, 2011.

[32] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program GPUs for general-purpose uses," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006.

[33] "Aparapi: Java API for expressing GPU bound data parallel algorithms," `http://developer.amd.com/zones/java/aparapi/Pages/default.aspx`, 2011.

[34] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing," in *Onward! '10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Oct. 2010.

[35] G. Mainland and G. Morrisett, "Nikola: embedding compiled GPU functions in Haskell," in *Proceedings of the third ACM symposium on Haskell (Haskell '10)*, 2010, pp. 67–78.

[36] S. Lee, V. Grover, M. M. T. Chakravarty, and G. Keller, "GPU kernels as data-parallel array computations in Haskell," in *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM)*, 2009.

[37] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers (SIGGRAPH '04)*, 2004, pp. 777–786.

[38] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, ser. Lecture Notes in Computer Science, vol. 5142, 2008, pp. 76–103.

[39] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the 25th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2010)*, 2010, pp. 89–108.

[40] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, "Scalable SIMD-parallel memory allocation for many-core machines," *The Journal of Supercomputing*, pp. 1–13, 2011.

[41] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács, "Abc: Algebraic bound computation for loops," in *LPAR (Dakar)*, 2010, pp. 103–118.

[42] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[43] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. mei W. Hwu, "XMalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *Proc. 10th IEEE International Conference on Computer and Information Technology (CIT)*. IEEE, July 2010, pp. 1134–1139.

[44] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: Writing parallel program portable between CPU and GPU," in *Proc. 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, Sept. 2010, pp. 217–226.

[45] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *Proc. 3rd Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[46] A. Ramamurthy, "Towards scalar synchronization in SIMT architectures," Master's thesis, The University Of British Columbia, Sept. 2011.

[47] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin, "Analysing memory resource bounds for low-level programs," in *Proc. 7th International Symposium on Memory Management (ISMM)*. ACM, June 2008, pp. 151–160.

[48] V. A. Braberman, D. Garbervetsky, and S. Yovine, "A static analysis for synthesizing parametric specifications of dynamic memory consumption," *Journal of Object Technology*, vol. 5, no. 5, pp. 31–58, June 2006.

BIOGRAPHICAL STATEMENT

Derek White is currently a Ph.D. student in Computer Science and Engineering at the University of Texas at Arlington working with Dr. Ishfaq Ahmad. He is one of the first recipients of a Graduate Assistance in Areas of National Need (GAANN) Fellowship at the University of Texas at Arlington. He received a B.S. in Computer Science from Louisiana State University in Shreveport in 2007 and has been a graduate student at the University of Texas at Arlington since August 2008 researching software engineering, program analysis, high-performance computing, and green computing.