Firepile: Run-time Compilation for GPUs in Scala

Nathaniel Nystrom
Faculty of Informatics
University of Lugano
Via Giuseppe Buffi 13
6904 Lugano, Switzerland
nate.nystrom@usi.ch

Derek White
CSE Department
University of Texas at Arlington
P.O. Box 19015
Arlington TX 76019-0015, USA
dwhite@uta.edu

Kishen Das
CSE Department
University of Texas at Arlington
P.O. Box 19015
Arlington TX 76019-0015, USA
kishen.das@gmail.com

Abstract

Recent advances have enabled GPUs to be used as general-purpose parallel processors on commodity hardware for little cost. However, the ability to program these devices has not kept up with their performance. The programming model for GPUs has a number of restrictions that make it difficult to program. For example, software running on the GPU cannot perform dynamic memory allocation, requiring the programmer to pre-allocate all memory the GPU might use. To achieve good performance, GPU programmers must also be aware of how data is moved between host and GPU memory and between the different levels of the GPU memory hierarchy.

We describe Firepile, a library for GPU programming in Scala. The library enables a subset of Scala to be executed on the GPU. Code trees can be created from run-time function values, which can then be analyzed and transformed to generate GPU code. A key property of this mechanism is that it is modular: unlike with other meta-programming constructs, the use of code trees need not be exposed in the library interface. Code trees are general and can be used by library writers in other application domains. Our experiments show Firepile users can achieve performance comparable to C code targeted to the GPU with shorter, simpler, and easier-to-understand code.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages, Object-oriented languages; D.3.4 [*Processors*]: Code generation, Compilers

General Terms Languages

Keywords GPU, Scala, OpenCL, run-time code generation

1. Introduction

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 [22] and CUDA [23] 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 [24].

This paper describes Firepile, a 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. 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 [3, 21, 29–31], that otherwise might require annotations, special types, or other extensions of standard Scala.

1.1 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 [25] and the OpenCL programming model [22]; 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

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0689-8/11/10. . . $10.00

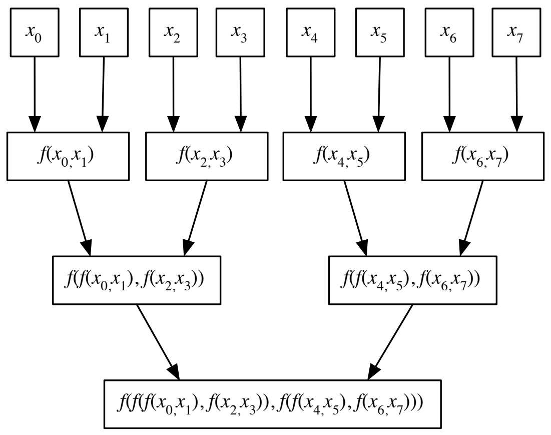
x_0
x_1
x_2
x_3
x_4
x_5
x_6
x_7
$f(x_0,x_1)$
$f(x_2,x_3)$
$f(x_4,x_5)$
$f(x_6,x_7)$
$f(f(x_0,x_1),f(x_2,x_3))$
$f(f(x_4,x_5),f(x_6,x_7))$
$f(f(f(x_0,x_1),f(x_2,x_3)),f(f(x_4,x_5),f(x_6,x_7)))$

Figure 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.

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 date to work items and work groups with all of these restrictions in mind.

1.2 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 `f` : $(\mathtt{T},\mathtt{T}) \rightarrow \mathtt{T}$. 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 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. 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.

1.3 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 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 [6], handles the details of data movement to and from the GPU, and compiles Scala code into OpenCL kernels, as described in Section 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

[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 paper.

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

to spawn. The compiler identifies array and object allocations within the kernel body and generates code to run *on 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.

1.4 Organization

The rest of the paper is organized as follows. Section 2 introduces the code tree mechanism used to translate Scala code into GPU kernels. Section 3 presents the Firepile library and compiler, including object and function translations and memory management. In Section 4, we describe our experiments and present our results. Related work is discussed in Section 5. Finally, in Section 6 we conclude and discuss of future work.

2. 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 1.3 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.[3]

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 3(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 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 3(b). The code tree nodes are instances of the case classes in Figure 4. 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.

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

```
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 2. Reduce in Firepile

2.1 Constructing code trees at run time

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 [33]. The code for the function object's apply method is then located.

```
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 3. 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.

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 4. Selected code tree classes

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.

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 4. 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 [26].

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)`:[4]

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

Figure 3(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.

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 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 are used to implement the compiler.

3.1 Writing a kernel

Consider again the `reduce` example from Figure 2 in Section 1.3. 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

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

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.

3.2 The run-time compiler

The Firepile compiler generates OpenCL kernels using code trees described in Section 2. The `spawn` function first constructs a code tree for its code block argument as outlined in Section 2. The trees are then compiled into C through a recursive translation function. The C code is, in turn, compiled by the JavaCL [6] 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 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.

```
struct Object {                union Object_sub {
 int __id;                      int __id;
};                              struct Object _Object;
                                struct B _B;
struct A {                      struct A _A;
 struct Object _Object;        }
 int x;
};                             union A_sub {
                                int __id;
struct B {                      struct B _B;
 struct A _A;                   struct A _A;
 int y;                        };
};
                               union B_sub {
                                int __id;
                                struct B _B;
                               };
```

Figure 5. Firepile class translation

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 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 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.

3.3 Class translation

Figure 5 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.

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

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 3.3.

```
val a: A = ...
a.m(10)
```

The call is translated into the following call to the dispatch function in Figure 6:

```
A_sub* a = ...;
dispatch_A_m(a, 10);
```

The types `A_sub` and `B_sub` used in the figure are defined in Figure 5.

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.

4. Experimental results

To evaluate the performance of Firepile, examples from the NVIDIA OpenCL SDK[5] 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 7. In addition to Reduce (similar to the code in Figure 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, 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 inter-

[5] `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 7. Summary of benchmarks run for Firepile and C++

ference 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 8. 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 benchmark 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 highest 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.

5. Related work

5.1 Meta programming

Lisp [29] was the first language to introduce meta-programming features. Lisp supports *quasiquoting* [3], 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 [15], Haskell [21], and C♯ [9]. MetaML [31] and MetaOCaml [30] support type-checking of quoted code. Scala has some experimental quasiquoting support in the class `scala.reflect.Code`. The Mnemonics [28] 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 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 [11] 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 [17] 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 [7] transforms uses of the standard Scala collections library into OpenCL kernels. The ScalaQL plugin [10] extends Scala with database queries.

5.2 GPU programming

The two most widely used programming models for GPUs are CUDA [23] an OpenCL [22]. 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 [16, 19, 27] and Java [6, 14, 34].

CLyther [8] 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 [7] translates Scala code into corresponding kernel code, employing JavaCL wrappers for execution. Originally, ScalaCL allowed ker-

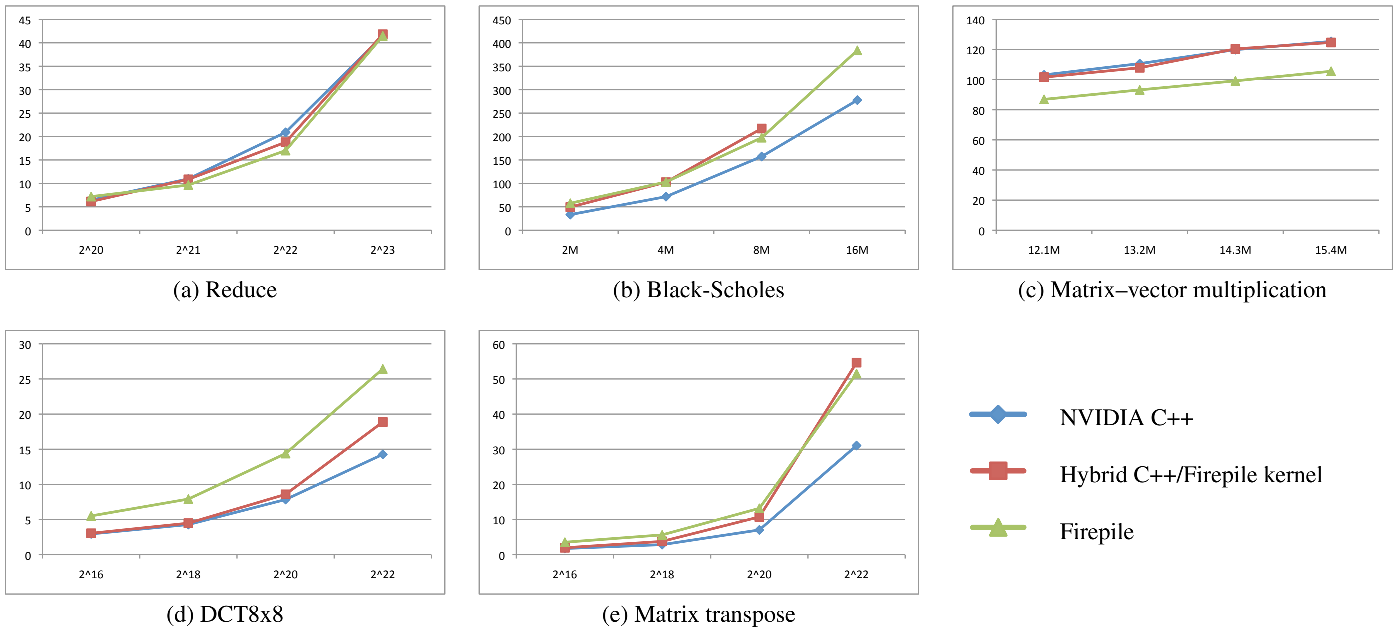

Figure 8. Total execution times in ms of each benchmark for different problem sizes.

nels 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 [12]. 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 [1] 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. [5] 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 [20] 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 [21] 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. [18] 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 [23] supports GPU programming through an extension of C++. The Brook language [4] extends the C with data-parallel constructs for stream programming on GPUs. Kernels are mapped to Cg shaders by the source-to-source compiler and the Brook runtime handles kernel execution. The Liquid Metal system [2, 13] 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).

6. Conclusions

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. Firepile works by translating function values at run time into code trees, from which GPU kernels can be generated. Performance of Firepile kernels is comparable to the performance of native C code.

Future plans for Firepile are to support more features of Scala—object allocation in particular—and to explore GPU-specific optimizations in the context of OO languages. Firepile currently supports only simple kernel where inputs are copied to the GPU, a kernel is run, and then outputs are copied back to the host. 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.

Acknowledgments

The authors thank Gutemberg Guerra-Filho for generously allowing us the use of his lab and computers for running experiments.

References

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

[2] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric 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)*, pages 89–108, 2010.

[3] Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers (SIGGRAPH '04)*, pages 777–786, 2004.

[5] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Onward! '10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, October 2010.

[6] Olivier Chafik. JavaCL: Java wrappers for OpenCL. http://code.google.com/p/javacl, 2011.

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

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

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

[10] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with database query capability. *Journal of Object Technology*, July 2010.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005. ISBN 0321246780.

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

[13] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103, 2008.

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

[15] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, October 1998.

[16] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA: GPU run-time code generation for high-performance computing. http://arxiv.org/abs/0911.3456, 2009. In submission.

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

[18] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM)*, 2009.

[19] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded gpu language by combining translation and generation. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*, pages 1610–1614, 2006.

[20] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM symposium on Haskell (Haskell '10)*, pages 67–78, 2010.

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

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

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

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

[25] 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.

[26] Martin Odersky et al. The Scala language specification, 2006–2011.

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

[28] Johannes Rudolph and Peter Thiemann. Mnemonics: type-safe bytecode generation at run time. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM)*, pages 15–24, 2010.

[29] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of Lisp. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 231–270, New York, NY, USA, 1993. ACM.

[30] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[31] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations (PEPM)*, pages 203–217, 1997.

[32] David Tarditi, Sidd Puri, and Jose 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)*, October 2006.

[33] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay 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.

[34] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*, pages 887–899, 2009.