Compiling a High-Level Language for GPUs

(via Language Support for Architectures and Compilers)

Christophe Dubach[1,2] Perry Cheng[2] Rodric Rabbah[2] David F. Bacon[2] Stephen J. Fink[2]

[1]University of Edinburgh [2]IBM Research

christophe.dubach@ed.ac.uk {perry,rabbah,dfb,sjfink}@us.ibm.com

Abstract

Languages such as OpenCL and CUDA offer a standard interface for general-purpose programming of GPUs. However, with these languages, programmers must explicitly manage numerous low-level details involving communication and synchronization. This burden makes programming GPUs difficult and error-prone, rendering these powerful devices inaccessible to most programmers.

We desire a higher-level programming model that makes GPUs more accessible while also effectively exploiting their computational power. This paper presents features of Lime, a new Java-compatible language targeting heterogeneous systems, that allow an optimizing compiler to generate high quality GPU code. The key insight is that the language type system enforces isolation and immutability invariants that allow the compiler to optimize for a GPU without heroic compiler analysis.

Our compiler attains GPU speedups between 75% and 140% of the performance of native OpenCL code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Performance

Keywords GPU, OpenCL, Java, Lime, Streaming, Map, Reduce

1. Introduction

In response to increasing challenges with frequency scaling, hardware designers have turned to architectures with increasing degrees of explicit parallelism. Today's hardware offerings range from general purpose chips with a few cores (*e.g.*, Intel Core i7), to specialized distributed-memory multiple-SIMD platforms (*e.g.*, IBM Cell), to graphics processors (GPUs) that support large-scale data parallel computations. Additionally, several efforts underway attempt to exploit reconfigurable hardware (FPGAs), with massively bit-parallel execution, for general-purpose computation.

OpenCL [9] and CUDA [15] have emerged as mainstream languages for programming GPUs and multicore systems. These popular languages provide APIs that expose low-level details of the device architecture. The programmer must manually tune low-level code for a specific device in order to fully exploit its processing resources. For example, a sub-optimal mapping of data to a GPU's non-uniform memory hierarchy may degrade performance by a factor of ten or more.

Experience shows that programming in a high-level language is more productive, portable, and less error-prone. Ideally, a programmer should express a program using high-level constructs that are architecture independent, and have the compiler automatically generate device-specific code that is competitive with low-level hand-written code. Indeed, programmers have enjoyed these benefits with general purpose programming languages on general purpose CPUs for several decades. In this paper, we address challenges in delivering similar benefits for programs running with GPUs.

This paper presents details of a GPU programming model in a new programming language called Lime. As presented earlier [2], Lime is a Java-compatible object-oriented language which targets heterogeneous systems with general purpose processors, FPGAs, and GPUs. The Lime methodology allows a programmer to gently refactor a suitable Java program into a pattern amenable for heterogeneous parallel devices. We present the design and evaluation of the Lime compiler and runtime subsystems specific to GPUs.

The Lime language exposes parallelism and computation explicitly with high level abstractions [2]. Notably, the type system for these abstractions enforces key invariants regarding isolation and immutability. The optimizing compiler leverages these invariants to generate efficient parallel code for multicores and GPUs, without relying on deep program analysis.

This paper shows how a Lime programmer can exploit a GPU without writing complex low-level code required with mainstream approaches (OpenCL or CUDA). The compiler and runtime system coordinate to automatically orchestrate communication and computation, map data to the GPU memory hierarchy, and tune the kernel code to deliver robust end-to-end performance. The contributions of this paper are:

- A design and implementation of an optimizing compiler to generate high quality GPU code from high-level language abstractions including isolated parallel tasks, communication operators, value types to express immutable data structures, and fine-grained map-and-reduce operations (Sections 3-4).
- A set of automatic optimizations for GPU architectures that may be applied without sophisticated alias analysis or data dependence analysis. These include memory optimizations that improve locality, reduce bank conflicts, and permit vectorization (Section 4.2).
- A detailed empirical performance evaluation comparing the generated code to hand-tuned OpenCL programs. The performance of the generated code lies between 75% to 140% of hand-written and tuned native OpenCL code (Section 5).

The performance results show that across a suite of 9 benchmarks, our compiler delivers a speedup ranging from $12x - 430x$ for the NVidia GeForce GTX580 architecture (Fermi) and $12x -$

PLDI'12, June 11–16, Beijing, China.

$416x$ for the AMD Radeon HD5970. Further, using the OpenCL multicore runtime, we report a performance gain between $4.8x - 32.5x$ for the Intel Core i7 architecture.

2. OpenCL Background

OpenCL and CUDA represent the *de facto* standards for general purpose programming on GPUs. OpenCL, in particular, was designed to provide an industry-standard API for systems with heterogeneous devices.

Although OpenCL provides a portable API, the API presents a low-level interface to the underlying hardware. The OpenCL programmer must explicitly manage many low-level details to map data to appropriate address spaces, enable vectorization, schedule data transfers between the host and device, and manage synchronization among queues connecting the host and devices.

In this section, we briefly review the structure of a small OpenCL N-Body application. This example motivates the need for high-level abstractions and introduces tuning issues germane to GPU performance.

Figure 1 illustrates an N-Body simulation expressed using a common OpenCL programming pattern. Lines 1-16 embody a data-parallel kernel that represents a n^2 force calculation. The kernel may run on a GPU or a multicore CPU.

Address Space Qualifiers The kernel declaration (line 1) includes address space qualifiers that map the respective data to the GPU memory hierarchy. OpenCL presents a non-uniform memory hierarchy with five types of memories: `private`, `local`, `global`, `constant` and `image`. The `private` memory represents a fast, small memory private to each computational thread. The `local` memory represents a shared memory, used to coherently share data between a small group of related threads called a *work group*. The `global` memory provides a shared address space for all threads on a device, with no implicit coherency guarantees across work groups. The `constant` and `image` memories represent specialized read-only storage. The former, typically a small space, usually holds constants referenced by a computational kernel. Graphics applications typically store texture objects for rendering a scene in the image memory. We inspected a number of OpenCL benchmarks and found that the kernels often use private, constant, and global qualifiers in the kernel signatures.

Vectorization The kernel code features the use of the OpenCL `float4` data type (lines 5, 6, 12). The code represents the forces as an array of tuple elements each consisting of four floating-point values, even though each force value has only three components. This decision allows the device to vectorize the memory accesses. OpenCL 1.0 only supports vectors of size 2, 4, 8, and 16.

Kernel Tuning The force calculation illustrates a typical data parallel pattern where multiple instances (*work items*) of the same kernel operate on disjoint data sets. Lines 2-4 show some manual calculations to determine the working set for a running work item. The working set depends on the number of concurrent kernel instances, a user-determined property that is typically determined through trial and error to suit the computational power of the target device. For N-Body, each working set represents the subset of particles for which the work item computes forces. The loop at lines 8-15 iterates through the kernel working set and computes the forces for the corresponding particles. Lines 9, 10, and 15 access local memory and perform synchronizations between kernel instances to ensure correct access to local memory.

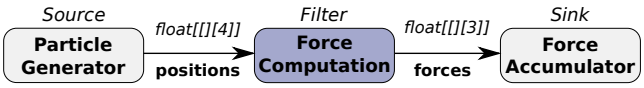
Orchestrating Execution The second half of Figure 1 shows a fraction of the host code necessary to orchestrate the execution of the kernel code. A typical execution pattern applies the following steps: (1) discover and initialize the device and compile the kernel code, (2) create a command queue, (3) create the kernel, (4) create read and write buffers, (5) enqueue commands to transfer the read buffer, invoke kernel, and transfer the write buffer. The programmer is responsible for scheduling the data transfers and overlapping these with kernel computation. This process uses at least a dozen OpenCL procedures based on our inspection of several hand-tuned OpenCL programs. We omitted an additional 182 lines of code which deal with step (1) alone.

3. Lime Programming Language

The previous example shows that OpenCL allows fine-grain control of the host and kernel code, but the low-level details impose a significant burden on the programmer. Similar points hold for CUDA. In contrast, Lime [2] provides a high-level object-oriented language offering task, data, and pipeline parallelism. It extends Java with several constructs designed for programming heterogeneous architectures with GPU and FPGA accelerators. Here, we briefly review the Lime constructs which the compiler relies on to efficiently offload computation to a GPU.

Figure 2 represents a Lime implementation of the N-Body example. Since the language is Java-compatible and interoperable, it provides a gentle migration path from Java. The figure illustrates this process with white and black squares: white squares prefix original Java code and black ones prefix new Lime code.

Line 3 sets up the main computation, as embodied in a *task graph* data structure. A task graph is a directed graph of computations, where values flow between tasks over edges in the graph. The `nbody` task graph from line 3 is illustrated in the figure below.



A particle generator task emits an initial set of particles, stored as an array of 4-element tuples: three elements for the position and one for the mass. A force computation task computes the force acting on each particle using a simple n^2 algorithm. This task produces an array of 3-element tuples representing the forces acting on each particle. The force accumulator task consumes these forces, and computes a new position for each particle.

The task graph just described represents a single simulation step. Typically the algorithm runs for a large number of simulation steps.

Two noteworthy Lime operators appear on line 3. First is the `task` operator, which creates a computational unit equivalent to an OpenCL kernel. Second is the `=>` (connect) operator, which represents the flow of data between tasks. The `finish` on line 4 initiates the computation and forces completion. This is necessary since Lime decouples the creation of task graphs from their execution.

3.1 Task and Connect

The `task` operator is used to create tasks (line 3). A task repeatedly applies a *worker* method as long as input data is presented to the task via an input port, and enqueues its output (the result of the method application) to an output stream. The operator binds the method specified after the dot to the task worker method. The worker method may be static (`NBody.computeForces`) or an instance method (*e.g.*, `NBody().particleGen`). In the latter, the task operator creates an instance of the type `NBody` and binds its instance method to the worker. The distinction between the two cases is that static worker methods are essentially pure functions and the instance methods may be stateful. Methods in lime are task-agnostic, meaning they may be invoked as conventional static or in-

```
1  kernel void calcForceKernel(global float4* positions, global float4* forces, local float4* localPos) {
2    int gi = get_global_id(0);     int ti = get_local_id(0);
3    int gs = get_global_size(0);   int ls = get_local_size(0);
4    int blks = gs / ls;
5    float4 pi = positions[gi];
6    float4 force = {0,0,0,0};
7    int baseIndex = 0;
8    for (int b=0; b<blks; b++) {
9      localPos[ti] = positions[baseIndex+ti];
10     barrier(CLK_LOCAL_MEM_FENCE);
11     for (int j=0; j<ls; j++) {
12       float4 pj = localPos[j];
13       ... update force based on pi and pj ... }
14     baseIndex += ls;
15     barrier(CLK_LOCAL_MEM_FENCE); }
16   forces[gi] = acc; }
17
18 void runKernel(int N /*num particles*/, cl_float4* positions /*initial state*/) {
19   cl_float4* forces = calloc(N, sizeof(cl_float4);
20   // create the input and output buffers using helper functions
21   cl_mem input  = clCreateBuffer(context, CL_MEM_READ_ONLY,  sizeof(cl_float4)*N, NULL, NULL);
22   cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float4)*N, NULL, NULL);
23   // lock-step iteration
24   while(true) {
25     // send the input data to the device (non-blocking)
26     clEnqueueWriteBuffer(queue, input, CL_FALSE, 0, sizeof(cl_float4) * N, positions, 0, NULL, NULL);
27     // pass the arguments to the kernel
28     clSetKernelArg(calcForceKernel, 0, sizeof(cl_mem), &input);
29     clSetKernelArg(calcForceKernel, 1, sizeof(cl_mem), &output);
30     clSetKernelArg(calcForceKernel, 2, maxWorkGroupSize * sizeof(cl_float4), NULL);
31     // start the one-dimensional kernel with N threads (force computation)
32     clEnqueueNDRangeKernel(queue, calcForceKernel, 1, NULL, &N, NULL, 0, NULL, NULL);
33     // read back the data from the device (blocking)
34     clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float4) * N, forces, 0, NULL, NULL );
35     // update state based on the acceleration computed (force accumulator)
36     update_position(positions, forces); } }
```

Figure 1. Parts of the OpenCL kernel ■ and host code ■ for N-Body

stance methods and only become worker methods by applying the task operator.

The language runtime repeatedly invokes the worker method as long as data items are available on the input port. A special case is the source task (lines 6-8) that emits data until interrupted by an `Underflow Exception` that can be thrown by any task to notify that the computation is finished.

Tasks in Lime are either *isolated* or *non-isolated*. An isolated task, also known as a *filter*, has its own address space and may not access mutable global state (*e.g.*, non-final statics fields in Java). Lime achieves isolation using a combination of *local* methods, and *value* types.

The worker method of an isolated task must be declared `local` (lines 10 and 13). A local method may only call other local methods, and may not access global mutable fields. The worker methods input immutable (value types) arguments (if any) and must return values (if any). This ensures that data exchanged between tasks does not mutate in flight, and provides the compiler and runtime greater opportunities for optimizing communication between tasks without imposing undue burden on the compiler to infer invariants involving aliasing.

A value type represents a deeply immutable object type (*e.g.*, data structure or array) declared using the *value* modifier on a type. A value array is indicated using double brackets, so for example `float[[][4]]` is a two dimensional array of floats, where the outer dimension is unbounded, the inner dimension is bounded to size four, and the entire array is immutable. Value arrays must be initialized at construction time (lines 8 and 17).

The `task` operator encapsulates computation whereas the `=>` (connect) operator encapsulates communication between tasks (line 3). This operator is used to connect two tasks when the output type of the upstream task (left of the operator) matches the input type of the downstream task (right of the operator). Lime exposes the communication between tasks explicitly using the connect operator so that the compiler and runtime can optimize the I/O and synchronization between tasks automatically and without programmer intervention, in contrast to OpenCL (see Figure 1 lines 20-35).

3.2 Map and Reduce

Lime also offers a *map* and *reduce* model for fine-grained data parallelism. This model suits the thread-level parallelism available in GPUs, and also short-vector SIMD instructions available in many general purpose ISAs.

A map operation applies a (logical) function to each element of some aggregate data structure, producing another aggregate data structure. The reduce operation combines values from an aggregate data structure using a combinator function. These abstractions are well-known in traditional functional languages.

The map operator is represented by the `@` token; see line 11 of Figure 2. It applies the function `computeParticleForces` to each element of the `particles` array, and returns the resultant array. Each application of the map function computes the force interactions between a particle `p` and all other `particles` (lines 13-17). The example omits the core force computation since it is similar in all implementations (OpenCL, Java and Lime).

```
1  □ public class NBody {
2  □   public static void simulate() {
3  ■     Task nbody = task NBody().particleGen => task NBody.computeForces => task NBody().forceAccumulator;
4  □     nbody.finish(); }
5
6  ■   private float[[][4]] particleGen() {
7  □     float[][4] theParticles = ...;  // initial state of the particles
8  ■     return new float[[][4]](theParticles); }
9
10 ■   private static local float[[][3]] computeForces(float[[][4]] particles) {
11 ■     return NBody @ computeParticleForces(particles, particles); }
12
13 ■   private static local float[[3]] computeParticleForces(float[[4]] p, float[[][4]] particles) {
14 □     float fx = 0, fy = 0, fz = 0;
15 □     for (int j=0; j<particles.length; j++) {
16 □       ... } // scalar computation to update fx, fy, and fz based on p and particles[j]
17 ■     return new float[[]] { fx, fy, fz }; }
18
19 ■   private void forceAccumulator(float[[][3]] particles) {
20 □     ... } // update the state of the particles for next iteration
21 □ }
```

Figure 2. Lime version of force calculation of NBody. The code marked with □ is original Java Source.

A reduction in Lime is expressed using an operator or method followed by `!` to indicate the operator or method should be treated as a combinator. The language permits instance or static methods as well as certain arithmetic operators to serve as reduction operators as long as they apply to two arguments of the same type and produce a result of that type.

4. Compilation Methodology

The explicit separation of computation and communication in Lime via the task and connect operators relieves the programmer from the burden of orchestrating the execution of tasks between host and device. This responsibility now falls onto the Lime compiler and runtime. Similarly, because Lime programs do not force the programmer to make an explicit distinction between the kernel and host codes, the compiler must determine a partitioning of the program between host (CPU) and device (GPU).

Our compilation methodology is illustrated in Figure 3. The compiler partitions the source code into host and device code, and compiles each partition to native code. The Lime compiler generates a mix of Java bytecodes and OpenCL. The bytecodes run in an unmodified Java virtual machine. The generated OpenCL code encompasses both the compiler-tuned kernels and the coordination and scheduling code for managing the buffers, scheduling data transfers and executing the kernels.

4.1 Kernel Identification

In the N-Body example, the main computation workload lies in the n^2 force calculation. Thus, a natural partition of the N-Body example runs the `particleGen` and `forceAccumulator` tasks on the host and the `computeForces` task on the GPU.

To allow offloading, the compiler requires the `computeForces` task to be an *isolated* task, also known as a filter. The language semantics for a filter guarantee that it does not perform globally side-effecting operations because it is isolated (*i.e.*, local) and its arguments are immutable (*i.e.*, values). As a result, our compiler recognizes filter task creations, and treats each filter as the unit of computation to offload. Note that the system may freely move filters between device and host without concern for data-races and non-determinism.

Within each filter, the compiler scans for map and reduce operations to identify opportunities for kernel-level data-parallelism.

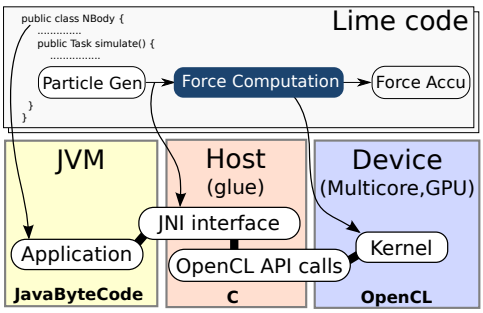


Figure 3. Starting from a Lime program, our compiler produces the application code that runs in the JVM, the C code that handle data exchange and the calls to the OpenCL API and finally the OpenCL kernel code.

The compiler detects data-parallel maps and generates corresponding parallel implementations without deep dependence analysis. Instead, it checks for the following invariants on map expressions: (a) the map function is static and local, and (b) the map function arguments are value types (includes primitive types). The type system guarantees that a static local method does not access globally mutable data, and because the value arguments are immutable, the map function is pure and side-effect free. In a similar way, the compiler may infer a parallel reduction.

Following kernel identification, the compiler performs kernel optimization (Section 4.2) with an emphasis on the GPU memory hierarchy. Lastly, the compiler generates appropriate glue code to orchestrate both the data transfers and the kernel invocations (Section 4.3). The glue code is implied by the connect operator which identifies the communication required between host and device.

Table 1 summarizes the differences between programming in OpenCL and compiling Lime code for GPUs. The salient observation is that Lime provides a much higher level of abstraction and does not expose low-level architectural details.

Table 1. GPU programming in OpenCL vs. Lime.

	OpenCL	Lime
offload unit	kernel	filter
communication	API	=> operator
data parallelism	manual	map & reduce
memory qualifiers	manual	compiler
synchronization	manual	compiler
scheduling	manual	compiler

4.2 Kernel Optimizations

The mechanics of optimizing and generating an OpenCL kernel from a Lime filter center on the exploitation of map and reduce operations for thread and SIMD parallelism, and applying a set of locality enhancing optimizations that take advantage of the OpenCL memory organization when applicable (*e.g.*, a GPU).

Figure 4 shows the code generated by the Lime compiler for the Lime code snippet shown earlier. The compiler takes advantages of the semantic information available at the source level to determine that the map operation (Figure 2 line 11) is data-parallel, and it exploits the immutable and bounded-size nature of individual particles to perform memory optimizations and vectorization. The generated kernel code will adapt to any number of threads started by the Lime runtime or as requested by the user. In Figure 4 line 9, the kernel loop iterates over the array of particles with each thread assigned an element `i` at line 10. This generated code is more robust than the hand-written OpenCL kernels we inspected because it executes correctly independent of the number of threads.

In addition to the obvious kernel input and output arguments, the compiler also generates a structure to contain runtime bookkeeping information and data values needed by the kernel code (see Figure 4(b)). Examples of the latter include array lengths that are used explicitly at line 15 and implicitly at line 11. This record is passed to the kernel as a parameter on line 2.

4.2.1 Memory Optimizations

Once the kernel is identified, a key optimization maps non-scalar (*e.g.*, array) data to the different memory structures in the device. This section describes how the compiler optimizes memory accesses for the OpenCL memory hierarchy (common to GPUs). The compiler permits for any of the optimizations to be enabled and disabled so that it is possible to perform an automated exploration of the memory mapping and layout.

The compiler drives memory optimizations using a relatively simple *pattern matching* algorithm. It scans the intermediate representation for common memory access idioms and applies the corresponding transformation when a pattern is encountered. In contrast to much previous work, our memory optimizer does not require sophisticated alias analysis or data dependence analysis. Instead, our compiler exploits the strong type system in Lime to infer necessary invariants without deep analysis. For example, immutable value types in the source language provide the key invariants that memory locations are read only and cannot be reassigned.

We claim that enabling a relatively simple memory optimizer is a strength of our approach, as compared to more unconstrained input languages that necessitate heroic program analysis.

Figure 5 illustrates some of the idioms recognized by the Lime compiler. In the figure and the text that follows, a *parallel loop* corresponds to a data-parallel map operator that the compiler has already inferred.

Global Memory Mapping data to the global memory is the default behavior of the optimizer when no other mapping is possible.

```
1  kernel void computeForces(
2    constant runtime_info_t* ri,
3    global float4* particles,
4    global float* result)
5  {
6      int gi = get_global_id(0);
7      int gs = get_global_size(0);
8      int len = ri->particles_length;
9      for (int it=0; it<len; it+=gs) {
10         int i=it+gs;
11         if (i<len) {
12             float fx=0;
13             float fy=0;
14             float fz=0;
15             float4 pi = particles[gi];
16             for (int j=0; j<len; j++) {
17                 float4 pj = particles[j];
18                 ... computation to update fx, fy
19                 ... and fz based on pi and pj
20             }
21             result[gi*3+0] = fx;
22             result[gi*3+1] = fy;
23             result[gi*3+2] = fz;
24 }   }   }
```

(a) Generated OpenCL code

```
1  typedef struct {
2      int particles_length;
3  } runtime_info_t;
```

(b) Generated structure for runtime information

Figure 4. Lime to OpenCL code generation

Private Memory The compiler attempts to map all the arrays that are not shared across threads to the fast private memory. Due to the extremely small capacity of this type of memory, the compiler only considers arrays whose size can be determined statically and does not exceed a certain threshold value. An array is not shared when it is allocated within the inner most parallel loop since each thread will execute its own instance of the loop body. Figure 5(a-b) show a simple example where an array variable `arr` is mapped to the private memory in OpenCL.

Local Memory In most OpenCL application, the local memory is used as a type of scratch pad for shared data. This memory is the second fastest type of memory. It is typically used when the data is reused among several threads running on the same core. A typical example where this happens is in the case of a double nested loop as shown in Figures 5(c-d). The parallel inner most loop reuses the value `v`. Since this inner loop is parallelized into many threads, it is possible to store parts of the `arg` array in the local memory and thus increase data reuse. The compiler performs a code transformation similar to loop tiling. When the size of the local memory cannot be determined statically our system dynamically allocates memory at runtime depending on the number of parallel threads.

The local memory is typically organized in different banks, with consecutive words assigned to different banks. Once the optimizer has decided to map an array into the local memory it determines whether padding is necessary in order to avoid bank conflicts. This is a common optimization although often done manually [17] in OpenCL or CUDA kernels. In Lime, it is relatively easier to automatically apply this optimization using the type information available and the fact that the code is virtually free from pointers as opposed to lower-level pointer-rich programs. The Lime compiler

```
float[] doWork(...) {
  Parallel loop {
    T[] arr = new T[10];
  } ...
```

(a) Private memory candidate (allocation within parallel region)

```
kernel doWork(...) {
  ...
  private int arr[10];
  ...
```

(b) OpenCL using private array allocation

```
float[] doWork(float[] arg) {
  ...
  Parallel loop (i) {
    float v = arg[i];
    Parallel loop {
      v;
    } } ...
```

(c) Local memory candidate (nested loop with data reuse)

```
kernel doWork(global float* arg,
              local float* local_mem) {
  ...
  if (local_id(1) == 0)
    local_mem[local_id(0)] = arg[global_id(0)];
  float v = local_mem[local_id(0)];
  ...
```

(d) OpenCL using local memory (synchronization omitted)

```
float[] doWork(float[][4] arg) {
  ...
  Parallel loop {
    float[4] v = arg[x];
    v[2];
  } ...
```

(e) Image memory candidate (accesses to v static)

```
kernel doWork(read_only image_2d_f arg) {
  ...
  ...
  float4 v = read_imagef(arg, {x,0});
  v.s2;
  ...
```

(f) OpenCL using image memory

```
float[] doWork(float[] arg) {  ...
   Parallel loop {
    arg[x];
  } ...
```

(g) Constant memory candidate (x invariant in the loop)

```
kernel doWork(constant float* arg) {
  ...
  arg[x];
  ...
```

(h) OpenCL using constant memory

Figure 5. Example of code patterns that our optimizer is looking for and the corresponding generated OpenCL in pseudo-code. Readers familiar with writing OpenCL code may recognize here some typical code pattern often encountered in OpenCL kernels.

detects the size of the array elements and adds padding accordingly. This ensures that each consecutive thread reads data from a different bank, thus increasing memory throughput.

Image Memory The compiler tries to map read-only arrays into the image memory. Since OpenCL 1.0 only supports access to the image memory by groups of 4 words, the compiler limits the scope to arrays whose last dimension is either 2 or 4. The compiler adopts a packed representation in the case where the last array dimension is of length 2. In addition, it prevents the optimizer from assigning to image memory arrays whose last dimensions elements are not accessed contemporaneously, in order to ensure good performance. Figure 5(e-f) shows a typical candidate (`arg`) for this optimization. The Lime array access expressions are converted into the appropriate OpenCL image access functions. Since the OpenCL does not support 1D images, the compiler maps the index `x` to the 2D coordinate `(x,0)` [1].

Constant Memory The constant memory is reserved for values that are *broadcast* to all the threads. That is, all the threads read the same address. In this case, the compiler identifies array accesses within a parallel loop that are accessed using a loop-invariant index as shown in Figure 5(g-h).

[1] The compiler implementation is more complex since the compiler may perform modulo operations when the index is greater than the maximum width supported by the image format.

4.2.2 Vectorization

Following the memory optimizations, the compiler vectorizes memory accesses for multidimensional arrays. An innermost array dimension is a vectorization candidate if it is of length 2, 4, 8 or 16. This optimization is only applied for arrays that are read-only and whose access to the last dimension is known statically. Vectorizing the memory accesses usually reduces the total number of memory accesses and thus improves bandwidth utilization and performance. This optimization is applied for data mapped to the global, local or constant memory (the image memory optimization is intrinsically already vectorized). Once again, the benefit of using a high-level language that allows for pointer-free programming makes this type of analysis simple.

4.3 Orchestrating Communication

Although the data and code isolation of Lime tasks makes computation offloading possible, the runtime system must still efficiently transfer data to and from the main system memory to the device. Because Lime targets devices that include GPUs and FPGAs, the runtime implementation adopts a universal "wire" format that relies only on sending a byte stream as shown in Figure 6.

The communication steps between the host JVM and the native device entail (1) serializing a Lime value to a byte array, (2) crossing the JNI boundary, and (3) converting this byte array into a C-style value. The particular representation of a value for use in OpenCL is specific to our code generator; the C deserializer does not necessarily convert to a standard C format. The return path is a mirror image in which we convert the OpenCL data structure to a

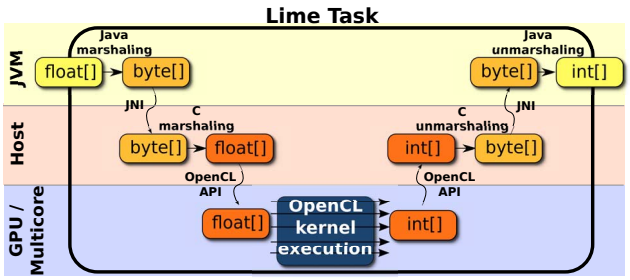


Figure 6. Data transfer between Java and the OpenCL device. This example shows a task that takes a float array as input and returns an int array.

byte array, return from the JNI call, and then deserialize from the byte array back into a heap-resident Lime value.

If not optimized, high communication costs can cancel out any performance gains from exploiting a GPU. Our initial implementation was simple and used Lime's internal runtime type information to serialize and deserialize. Unfortunately, the performance was so poor that more than 90% of the time was spent marshaling data to or from a byte array. Performance was greatly improved by writing custom serialization routines for the most common types–primitives and (nested) arrays of primitives. During the initialization process of migrating a task to a native device, the runtime will find a custom serializer based on the data type. Because the default marshaller is written recursively, we modified it to use a specialized marshaller recursively when available. For instance, if the data type is a tuple of integer arrays, then although there is no specialization at the tuple level, the lowest level integer arrays (where most of the data actually is) will still be optimized. Finally, because Lime arrays can express bounds (e.g., sub-rectangular arrays are possible), the runtime system can sometimes determine the exact size of the target byte array up-front.

Marshaling on the C side is similar but more specialized. Because our OpenCL backend only handles rectangular arrays of primitives, the data is generally densely packed. The layout must take alignment and vectorization into consideration, making the marshaller more specialized though less comprehensive. Because the serialization is primarily memory-bound, we simply use malloc/free rather than implement our own memory manager to lower costs.

Our current communication implementation could be further optimized since it entails repeated serialization in the same address space. However, our current design affords a common format as a starting point for a communication subsystem that supports heterogeneous devices. One might further optimize the protocol by creating specific communication channels so that the sender and receiver are aware of the data format the other party desires. Going even further, one might be able to avoid a low-level memory copy by pinning memory pages and managing memory explicitly. However, these changes come at the cost of OS and JVM portability.

5. Evaluation

We present an empirical evaluation of the Lime system to answer three questions:

1. **End-to-end Speedup.** Can the Lime programmer effectively exploit a GPU to improve performance? That is, can the system deliver high performance, including all communication costs and runtime overhead?
2. **Comparison to hand-tuned OpenCL.** What is the quality of the OpenCL code generated by the Lime compiler as compared to hand-tuned native OpenCL implementations?
3. **Computation vs. Communication.** How much overhead does the system introduce to communicate between the host and device?

Table 3 reports the set of benchmarks used in this study. The set includes three benchmarks from Parboil [1] and two benchmarks from JavaGrande (JG) [13]; we selected the benchmarks that were easiest to port to Lime. We expect other benchmarks which can be expressed using task graphs and map and reduce to benefit in the same way. In addition we include two benchmarks, N-Body and Mosaic, that we wrote from scratch. Previous sections reviewed N-Body in detail. Mosaic features a map-and-reduce algorithm to compare tiles from a reference image to tiles from an image library to find the best-matched tiles using a scoring function.

Some of the benchmarks predominantly exercise floating-point arithmetic. ALUs in modern processors perform floating-point arithmetic in at least double-precision, whereas GPU ALU building blocks are single-precision. For GPUs, single-precision operations run faster than double-precision ones. Because this paper focuses on compilation rather than numerical stability issues, we present results for both single- and double-precision variants in cases where precision strongly affects performance.

Table 2 lists the hardware platforms evaluated. We measure performance on four platforms. In each case, Lime tasks are compiled to OpenCL and run natively using the OpenCL runtime, while the remaining application code runs in bytecode. The Intel Core i7 system runs 64-bit Ubuntu Linux 10.10 with the 2.6.35-28 kernel. The NVidia cards represent two generations of GPU architectures: a recent GeForce GTX580 (Fermi) and a 2006 GeForce GTX8800. The latter is used to compare the Parboil benchmarks since they are specifically hand-optimized for this card [17]. The GeForce architecture evolved substantially between these generations. Notably, the Fermi architecture adds caches in addition to the local memory. The NVidia GPUs use CUDA 4.0.13 with device driver version 270.40. The AMD GPU uses driver version 11.9 and AMD OpenCL SDK 2.5-RC2.

5.1 End-to-end Speedup

Figure 7 shows the bottom-line, end-to-end performance results including all system overheads. The figure represents performance results of the Lime code compiled to OpenCL, running partially in bytecode and partially in the native OpenCL runtime for the CPU (top) and the GPU (bottom). The figure reports speedup based on wall-clock execution times, measured after a preliminary warmup phase to ensure JIT optimizations occur. The figure normalizes speed as compared to Lime code running entirely in bytecode.

The baseline Lime bytecode performance achieves 95-98% of the performance of the original pure Java implementations for N-Body, Mosaic and JG-Series. In the worst case, the performance of JG-Crypt is half as fast when running Lime compiled to bytecode, as compared to the original Java. This slowdown is an artifact of our methodology in porting from Java, since we only ported the dominant computational kernels to Lime. As a result, the Java to Lime interoperability introduces some overhead with respect to array conversion, and further the cost of byte-array accesses in Lime are more expensive than in Java. However, note that the acceleration gained by compiling the kernels to OpenCL more than compensates for the slowdown due to this interoperation.

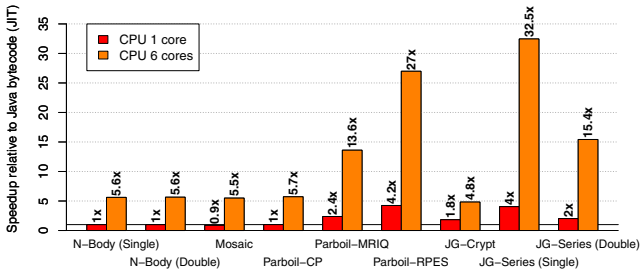
Since the end-to-end measurements include both computation and communication costs, we only show the results for the Core i7, the faster NVidia GTX580, and the AMD HD5970. This is because the overheads are proportionately larger with greater acceleration.

Table 2. Evaluation platforms. Note, the number of GPU cores represents the number of Streaming Multiprocessors.

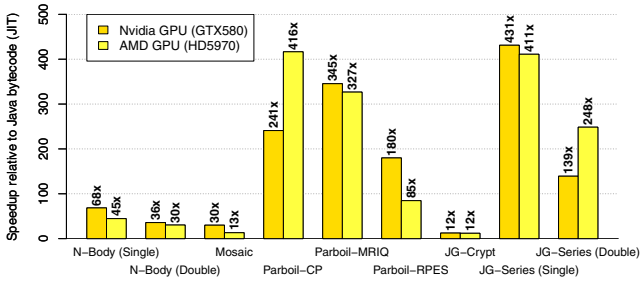
Type	Model	Cores	FP units per core	Const. mem	Local mem	L1 cache	L2 cache	L3 cache
CPU	Intel Core i7-990X	6	4 single (4 double)			6x64KB	6x256KB	12MB
GPU	NVidia GeForce GTX 8800	16	8 single	64KB	16x16KB			
GPU	NVidia GeForce GTX 580	16	32 single (16 double)	64KB	16x48KB	16x16KB	768KB	
GPU	AMD Radeon HD 5970	20	80 single	64KB	20x32KB			

Table 3. Benchmarks used in the evaluation.

Name	Description	Input size	Output size	Data Type
N-Body	N-Body simulation	64KB / 128KB	48KB / 128KB	Float / Double
Mosaic	Mosaic image application	600KB	5MB	Integer
Parboil-CP	Coulombic Potential	62KB	1MB	Float
Parboil-MRIQ	Magnetic Resonance Imaging	432KB	256KB	Float
Parboil-RPES	Rys Polynomial Equation Solver	13MB	4MB	Float
JG-Crypt	IDEA encryption	3MB	3MB	Byte
JG-Series	Fourier coefficient analysis	780KB / 1560KB	780KB / 1560KB	Float / Double



(a) CPU (Core i7)



(b) GPU

Figure 7. End-to-end speedup (includes overhead).

Figure 7(a) shows the speedups (higher is better) when running on 1 or 6 cores. Since the CPU supports hyperthreading, running on a single core runs two threads, one each for the JVM and OpenCL kernel. The 1-core performance is generally the same as the baseline, with a 10% degradation in the worst case because of high marshaling costs (see Figure 9). The gains for Parboil-MRIQ, Parboil-RPES, and JG-Series result from a faster implementation of the transcendental functions in OpenCL compared to Java.

The performance scales as the number of cores is increased, with five benchmarks showing a speedup of $4.8 - 5.7x$. The four remaining benchmarks show super-linear speedups of $13.6 - 32.5x$ using 6-cores. This is attributed to hyper-threading (permitting two OpenCL threads to run per core) and cache effects.

Figure 7(b) shows the speedups resulting from co-execution between the JVM and the GPU. The speedups vary significantly depending on the benchmark and platform, ranging from 12x to 431x. The benchmarks which do not use floating-point (JG-Crypt and Mosaic), or which use simple floating-point operations (N-Body) see the lowest end-to-end speedups. These benchmarks also have high communication to computation ratios, as shown in later results. The largest performance gains manifest for applications using transcendental functions.

The results also show that double-precision computation on the GTX580 is approximately $2 - 3x$ slower than single-precision, and $\approx 1.5x$ slower on the HD5970.

Overall, the results demonstrate that the Lime system delivers substantial end-to-end speedups as compared to the original Java programs, for all the benchmarks and platforms considered.

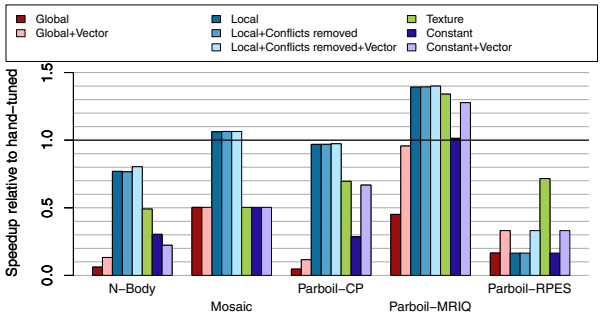
5.2 Comparison to hand-tuned OpenCL

Next, we evaluate the code quality of the generated OpenCL code, as well as the different memory optimizations described earlier. For this purpose, we wrote and hand-tuned OpenCL versions of N-Body and Mosaic, and converted three existing Parboil benchmarks originally written in CUDA to OpenCL. We made our best effort to optimize N-Body and Mosaic for the GTX580 GPU. We also include results for the GTX8800 GPU because the Parboil benchmarks were optimized specifically for this card by another research group [17].

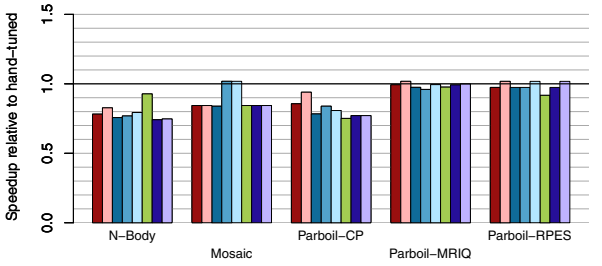
OpenCL requires the programmer to select the number of threads to run and how these threads map to cores. These tuning parameters can have a strong impact on performance. To control for these variables, we conducted an exhaustive systematic offline exploration of the tuning parameters and use the best settings for each experiment. For example, the hand-tuned versions of the Parboil benchmarks are optimized for the GTX8800, but those settings are not competitive on the GTX580. A system could perform this auto-tuning automatically ahead of time or at runtime, but such tuning falls outside the scope of this paper.

To evaluate code quality, we measure only time spent in computational kernels on the GPU, and exclude time spent on the host and time spent in explicit communication between host and GPU. Figure 8 shows the relative performance of computational kernels for compiled Lime code as compared to hand-tuned OpenCL. A speedup greater than one indicates performance better than hand-tuned code. A speedup less than one indicates a slow down and an opportunity for further improvements in the Lime compiler.

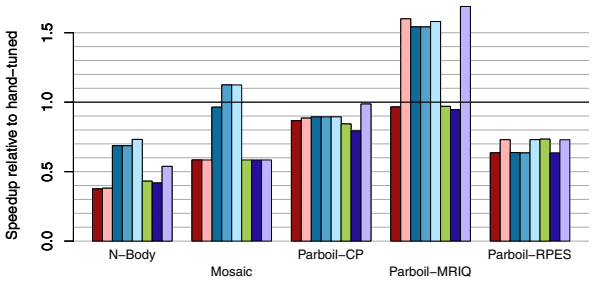
For each benchmark, the Figure shows results using the various memory optimizations applied by the Lime compiler including



(a) NVidia GTX8800



(b) NVidia GTX580 (Fermi)



(c) AMD Radeon HD5970

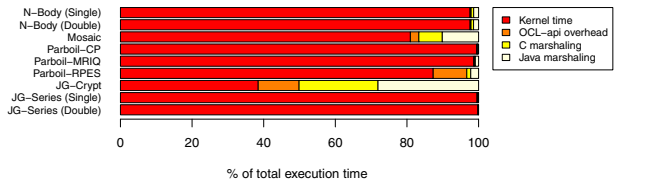
Figure 8. Lime vs. hand-tuned OpenCL kernel-times and effects of optimizations.

vectorization. The Figure shows 8 bars per benchmark, representing each of the memory optimizations covered in Sections 4.2.

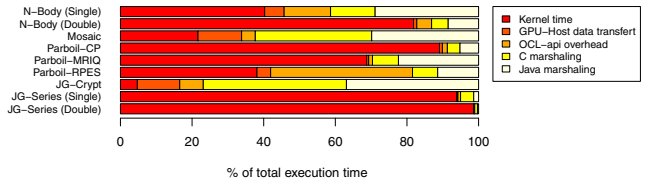
Overall, the results show that with the best optimization choices, the Lime compiler delivers competitive performance, attaining between 75% and 140% of the hand-coded performance. Exceeding hand-coded performance indicates cases where the human programmer was imperfect – we discuss specific issues below.

The results show that using the global memory generally yields the worst performance, even when using vectorization. In the worse cases, the slowdown is up to $10x$ compared to hand-tuned for the GTX8800, up to 60% for the HD5970, and 20% for the GTX580.

On the other hand, the compiler can often use the local memories effectively. Note in particular that the compiled code surprisingly outperforms the hand-tuned versions for the Mosaic benchmark. After further investigation, we discovered that the compiled code is more effective at reducing memory bank conflicts. The compiler-generated code for Parboil-MRIQ also slightly outperforms the hand-tuned kernel, when using constant memory. The Parboil-RPES benchmark benefits significantly from the use of texture memory on the GTX8800 because it is equipped with a hardware cache, and this benchmark exhibits good spatial locality.



(a) CPU (Core i7)



(b) GPU (GTX580)

Figure 9. Computation and communication costs.

The GTX580 architecture differs from the other GPUs by placing a cache between the device memory and the cores. As a direct consequence, the performance is less sensitive to memory optimizations, as shown in Figure 8(b). Using global memory delivers performance relatively close to the hand-tuned version, however, optimizations are necessary to recover the last 10-20% of performance in some benchmarks.

We conclude that it is possible for a Lime compiler to achieve performance competitive with typical hand-tuned code for the platforms and benchmarks considered. The results also demonstrate the sensitivity of performance to the GPU memory architecture. We claim these results further demonstrate the need to lift the level of programming abstraction away from low-level GPU details. Clearly, writing a portable, high-performance OpenCL code for multiple devices imposes a substantial burden on a human programmer.

5.3 Communication vs. Computation

The end-to-end speedups shown earlier include all runtime overhead, including communication between host and device. In our system, offloading the computation involves moving the data from Java to C and also from C back to Java (refer to Figure 6). In addition, there are costs attributed to the OpenCL API, and PCIe transfer costs to move data from the host to the device.

Figure 9 shows the breakdown of computation (kernel time) and communication costs (everything else). When running on a multicore, shared memory obviates the need for memory transfers; as a rule, computation dominates the execution time (Figure 9(a)). JG-Crypt provides an exception to the rule, since its computation ratio per byte is particularly low.

In contrast, the communication costs on a GPU are relatively higher due to its greater computational power. We show the results for the GTX580 in Figure 9(b); the computation to communication ratios are comparable for the other GPUs.

Most of the overhead (30%) comes from data marshaling (both Java and C). Marshaling objects in Java suffers from significant overheads due to arrays bounds checking and object allocation. Setting up OpenCL data structures is relatively fast (typically 5%), except for JG-RPES (40%). We are investigating the cause of this anomaly.

The raw data transfer from host memory to device memory does not play a major role in communication costs. We expect this trend to continue with PCIe3.0 and tighter integration of GPU and CPU.

Overall, the combined overhead due to all communication averages 40%. Although this overhead is high, the tremendous computational power of the GPU still allows impressive end-to-end speedups.

We conclude that communication costs, while not yet crippling, leave much room for improvement. In future work, we plan to pursue various strategies to reduce communication overhead. To avoid extraneous copying, the Java marshaling code should marshal directly to a format as required for device memory. This would approximately halve the marshaling overhead. More generally, the communication costs can be hidden by well-known pipelining techniques that overlap communication and computation; these techniques lie beyond the scope of this paper.

6. Related Work

Recent years have seen many projects targeting general purpose languages to exploit multicores and GPUs. The closest related work is Sponge [7], a compiler which generates CUDA code from the StreamIt [19] programming language. Udupa et al. [20] also target StreamIt for GPUs. This work focuses on the problem of scheduling the tasks to the GPUs. Similar to our work, Sponge schedules different tasks onto GPUs and optimizes the mapping to the different type of memory. In contrast to Sponge, our system generates OpenCL code which can target multicore platforms as well as GPUs. Sponge supports only coarse-grained parallelism, whereas Lime includes constructs that support fine-grained data parallel as well. Data parallel operations make it easier to exploit thousands of threads on a GPU. Further, the StreamIt programming model is much simpler compared to Lime as it does not permit object allocation or unbounded arrays, requires the task graph to be fully resolved at compile time, and does not support object-oriented programming features. Lime on the other hand does not have any of these limitations. Further, because Lime is Java-compatible, it permits a gentle migration of existing Java code.

6.1 GPU Programming and Optimization

There are many other task-oriented languages that are suitable for GPU programming. Cg [12] was among the first languages to be developed to program GPUs. Then Brook [3] was introduced featuring the use of a streaming programming model. Accelerator [18] was later developed as a C# API library that uses a data-parallel model based on parallel array to program GPUs. Today, general purpose computations on GPUs is dominated by OpenCL [9] and CUDA [15]. Some researchers are investigating automatically translating OpenMP source code to CUDA [10] while others apply directives to sequential C code to convert them into CUDA programs with the hiCUDA [6] framework. Earlier parts of this paper address the primary differences between Lime and OpenCL. These advantages are the same compared to CUDA.

Another related new language is the IBM X10 language. It provides abstractions for programming distributed memory parallel computers (*e.g.*, clusters) with a globally shared, partitioned address space. X10 considers the GPU as a shared-memory parallel computer (X10 "place"), where threads ("activities") communicate and synchronize through shared memory. Their work [4] to incorporate CUDA abstractions does not describe compiler optimizations to map data structures to the GPU memory hierarchy. Instead it provides language constructs for the programmer to manage this mapping explicitly.

Yang et al. [23] contribute a compiler framework to optimize GPU code. This compiler takes a simple unoptimized kernel as input and applies optimizations as discussed in [17]. Similarly, CUDA-Lite [21] coalesces memory accesses of existing kernels. A compiler for a high level language (HLL) can apply similar optimizations after translating to a low level representation. However, a HLL compiler can perform more aggressive transformations by exploiting higher-level semantic information embodied in the source code. We demonstrated several techniques whereby the Lime compiler exploits high-level information exposed by the type system to realize aggressive parallelization, prove isolation, and optimize the mapping to the memory hierarchy. It is unclear whether a low-level approach can in general recover this level of semantic information, due to difficulties inherent to sound whole program static analysis of object-oriented languages.

6.2 Multicores Programming

Gordon et al. [5] developed a compiler that maps the StreamIt language to multicore architectures. Intel's array building block (ArBB) [14] consists of a virtual machine and a C++ API that defines new parallel types such as collections. These collections are treated like values and the JIT optimizes these and extract thread and vector (SIMD) parallelism. Our work differs in that we start directly with a streaming computational model, making it easier to decompose programs for heterogeneous platforms. In contrast to ArBB, where the programmer has to deal specifically with data transformation between the C data types and the parallel collections (using the bind function), the programmer simply uses the standard array types provided by Lime. Finally, our system works with GPUs as well as multicores without changes to the program.

6.3 Runtime for Heterogeneous Platforms

SoCC [16] is an extension to C that allows the programmer to manage distributed memory, express pipeline parallelism and map the different tasks to resources. EXOCHI [22] is an effort from Intel that focuses on providing a runtime for integrating accelerators with general purpose processor. It provides shared memory and dynamic mapping of tasks to accelerators. The Quilin [11] system is composed of a C API that is used to write parallel programs and an adaptable runtime that dynamically maps computations to processing elements in a CPU+GPU system. Jablin et al. [8] proposed a new runtime management system that frees the programmer from explicitly managing data movement between the CPU and GPU on the host side. It determines which data are required by a GPU kernel and also copies the data to the GPU memory. The Lime model intrinsically provides this functionality via the task graph, and our compilation methodology leverages the language semantics and type system to automatically partition the code between host and device, generate the corresponding code, and coordinate the overall execution without programmer intervention.

7. Conclusion

This paper reviewed how a compiler for Lime, a high-level Java-compatible language, can exploit computational resources on a GPU. Exploiting invariants enforced by the type system, the compiler and runtime system implement transformations to exploit massively parallel GPU devices with non-uniform memory hierarchies. Benefiting from language and compiler co-design, the system achieves these goals without ambitious program analysis. Experimental results show that for the cases considered, the system delivers impressive speedups as compared to a Java implementation, and generates code quality in the same ballpark as hand-tuned code.

Although this paper has focused on GPUs, Lime supports a variety of architectures including specialized multicores and FPGAs. The results from this paper indicate that the Lime approach remains promising for GPUs, as part of a larger vision for programming

heterogeneous system. We remain encouraged that this language-based approach may help bring the computational power of heterogeneous architectures to mainstream programmers.

Acknowledgments

We thank Joshua Auerbach and the members of the Liquid Metal team who contributed infrastructure and a stimulating working environment. Christophe Dubach was partially supported by the Royal Academy of Engineering and EPSRC.

References

[1] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php, 2011.

[2] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*, 2010.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[4] D. Cunningham, R. Bordewekar, and V. Saraswat. GPU programming in a high level language: Compiling X10 to CUDA. In *X10 Worksop*, 2011.

[5] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[6] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. Parallel Distrib. Syst.*, 22, Jan 2011.

[7] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *ASPLOS*, 2011.

[8] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[9] Khronos OpenCL Working Group. *The OpenCL Specification*.

[10] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, 2009.

[11] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.

[12] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH*, 2003.

[13] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pp. 72–80, New York, NY, USA, 1999. ACM.

[14] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *CGO*, 2011.

[15] NVIDIA Corporation. *The CUDA Specification*.

[16] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin. SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In *CASES*, 2008.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[18] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS*, 2006.

[19] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *CC*, 2002.

[20] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *CGO*, 2009.

[21] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. Languages and compilers for parallel computing. In *LCPC*, 2008.

[22] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI*, 2007.

[23] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, 2010.