LIQUID METAL
OBJECT-ORIENTED PROGRAMMING
OF HETEROGENEOUS MACHINES

2011 FCCM Workshop on High-Level Synthesis and Parallel Computation Models

Joshua Auerbach
Perry Cheng
Rodric Rabbah
Christophe Dubach

David F. Bacon
Steven Fink
Sunil Shukla
Yu Zhang
The Heterogeneous Era

GPU

FPGA

Tilera 64

Cell BE

IBM PowerEN
The FORTRAN Automatic Coding System

J. W. BACKUS†, R. J. BEEBER†, S. BEST‡, R. GOLDBERG†, L. M. HAIJT†,
H. L. HERRICK†, R. A. NELSON†, D. SAYRE†, P. B. SHERIDAN†,
H. STERN†, I. ZILLER†, R. A. HUGHES§, AND R. NUTT∥

Introduction

THE FORTRAN project was begun in the summer of 1954. Its purpose was to reduce by a large factor the task of preparing scientific problems for IBM’s next large computer, the 704. If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. For it was known that about two-thirds of the cost of solving most scientific and engineering problems on large computers was that of problem preparation. Furthermore, more than 90 per cent of the elapsed time for a problem was usually devoted to planning, writing, and debugging the program. In many cases the de-
HETEROGENEOUS PROGRAMMING TODAY

- Java
- C++
- Python

- Cuda
- OpenCL

- C++ Intrinsics

- VHDL
- Verilog
- SystemC

- CPU Compiler
- GPU Compiler
- Node Compiler
- Synthesis

- binary
- binary
- binary
- bitfile

- CPU
- GPU
- PowerEN
- FPGA
THE LIQUID METAL PROGRAMMING LANGUAGE

Lime

Lime Compiler

CPU Backend
- bytecode
  - CPU

GPU Backend
- binary
  - GPU

Node Backend
- binary
  - PowerEN

Verilog Backend
- bitfile
  - FPGA
THE Artifact Store & Exclusion

Lime

Lime Compiler

bytecode

binary

binary

bitfile

Artifact Store

CPU

GPU

PowerEN

FPGA
HETEROGENEOUS EXECUTION OF LIME

Heterogeneous execution of Lime involves the following components:

1. **Lime**
   - lime compiler
   - artifact store
   - lime virtual machine (LVM)

2. **Lime Compiler**
   - Bytecode input
   - Binary output
   - Bitfile output

3. **Artifact Store**
   - Stores the compiled artifacts

4. **Lime Virtual Machine (LVM)**
   - Supports CPU, GPU, PowerEN, and FPGA
   - "Bus" for interconnection

The diagram shows the flow of execution from Lime to the LVM, facilitated by the artifact store.
EXECUTION, COMMUNICATION, AND REPLACEMENT
THE LIME LANGUAGE
LIME: JAVA IS (ALMOST) A SUBSET

% javac MyClass.java
% java MyClass

% mv MyClass.java MyClass.lime
% limec MyClass.lime
% java MyClass

INCREMENTALLY USE LIME FEATURES
LIME LANGUAGE OVERVIEW

Core Features
Programmable Primitives
Map & Reduce Operations
Stream Programming

Supporting Features
Reifiable Generics
Ranges, Bounded “for”
User-defined operators

Immutable Types
Bounded Types
Bounded Arrays
Primitive Supertypes

BIT-LEVEL PARALLELISM

PIPELINE PARALLELISM

DATA PARALLELISM

Graph Construction
Isolation Enforcement
Closed World Support
Rate Matching
Messaging

Typedefs
Local type inference
Tuples
STREAMING COMPUTATION
PIPELINE PARALLELISM
local char work(char c) {
    return toUpper(c);
}

var upperscaser = task work;
var pipeline = task worker1 => task worker2 => task worker3;
SOURCES AND SINKS

Heap

source

filter

sink

File System

reader(…) { … }

worker(…) { … }

writer(…) { … }

char

tmp/mydata

dev/tty
public static void main(string[][] args) {

    char[][] msg = {
        'H', 'E', 'L', 'L', 'O', ',', ',', ',',
        'W', 'O', 'R', 'L', 'D', '!', ',', '\n'};

    var hello = msg.source(1) =>
        task Character.toLowerCase(char) =>
            task System.out.print(char);

    hello.finish();
}
DEMO
HELLO WORLD
LIME/ECLIPSE ENVIRONMENT
The "=>' operator, called "connect", connects the output of the task on the left to the input of the task on the right.

When the "task" operator is applied to a "void" method, the result is a sink task. Source and sinks are special because they are allowed to perform global side-effects -- like printing something on the console.

A series of connected tasks is called a pipeline.

```javascript
var hello = msg.source(1) =>
    task Character.toLowerCase(char) =>
    task System.out.print(char);

/*
* The finish() method is used to run the pipeline and wait for it to process all of its input data.
*/

hello.finish();
```

```
hello, world!
```
var averager = task Averager().avg;

double avg(double x) {
    total += x;
    return total/++count;
}
Rate Matching

```javascript
var matchedpipe = task AddStuff().work1 => # => task work2;
```
DEMO

N-BODY SIMULATION: CPU VS. GPU
9x Speedup (9.26 GFLOPS) on Laptop

REPLACING GalileoSingleAccCalculator.computeForces-1 (GalileoSingleAccCalculator.computeForces:float[][][4]
VIRTUALIZATION OF DATA MOVEMENT
MAP & REDUCE OPERATIONS

DATA PARALLELISM
Array Parallelism

```c
float[ ] a = ...;
float[ ] b = ...;
float[ ] c = a @+ b;
float sum = + ! c;
```

Indexable<int,float>
Collectable<int,float>
package lime.util.synthesizable;

public class FixedHashMap<K extends Value, V extends Value,
    BUCKETS extends ordinal<BUCKETS>, LINKS extends ordinal<LINKS>>
    extends AbstractMap<K,V>
{
    protected final nodes = new Node<K,V>[BUCKETS][LINKS];
public local V get(K key) {
    Node[LINKS] row = nodes[hash(key)];
    boolean[LINKS] selections = row @ compareKey(key);
    V[LINKS] vals = row @ getValueOrDefault(selections);
    return !vals;
}
public local V get(K key) {
    Node[LINKS] row = nodes[hash(key)];
    boolean[LINKS] selections = row @ compareKey(key);
    V[LINKS] vals = row @ getValueOrDefault(selections);
    return ! vals;
}
public local V get(K key) {
    Node[LINKS] row = nodes[hash(key)];
    boolean[LINKS] selections = row @ compareKey(key);
    V[LINKS] vals = row @ getValueOrDefault(selections);
    return (! vals);
}
public local V get(K key) {
    Node[LINKS] row = nodes[hash(key)];
    boolean[LINKS] selections = row @ compareKey(key);
    V[LINKS] vals = row @ getValueOrDefault(selections);
    return ! vals;
}
VIRTUALIZATION OF DATA PARALLELISM

![Diagram]

- [Diagram 1: Circuit Diagram]
- [Diagram 2: Vector Register File (VRF) and Vector Unit]
- [Diagram 3: Rasterizer and Stencil Test]
CURRENT RESULTS
**How Do we Evaluate Performance?**

- **Speedup for Naïve Users**
  - How much faster than Java?

- **Slowdown for Expert Users?**
  - How much slower than hand-tuned low-level code?

- **Our methodology:**
  - Write/tune/compare 4 versions of each benchmark:
    - Java, Lime, OpenCL, Verilog
  - Doesn’t address flops/watt, flops/watt/$, productivity
Expert vs Naïve Speedup: Kernel Time

(JAVA Baseline)

<table>
<thead>
<tr>
<th></th>
<th>Expert</th>
<th>Lime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photomosaic</td>
<td>37x</td>
<td>26x</td>
</tr>
<tr>
<td>n-body</td>
<td>193x</td>
<td>126x</td>
</tr>
<tr>
<td>idct</td>
<td>10x</td>
<td>10x</td>
</tr>
<tr>
<td>des</td>
<td>150x</td>
<td>92x</td>
</tr>
</tbody>
</table>

The chart shows the speedup normalized to manual implementations for different tasks and hardware platforms:
- Photomosaic: 37x Expert, 26x Lime
- n-body: 193x Expert, 126x Lime
- idct: 10x Expert, 10x Lime
- des: 150x Expert, 92x Lime

The chart compares the performance of expert and naïve implementations on GPU and FPGA platforms.
**Expert vs Naïve Speedup: End-to-End**

*(Java Baseline)*

<table>
<thead>
<tr>
<th>Task</th>
<th>Expert</th>
<th>Lime</th>
</tr>
</thead>
<tbody>
<tr>
<td>photomosaic GPU</td>
<td>6.67x</td>
<td>6.10x</td>
</tr>
<tr>
<td>n-body GPU</td>
<td>136x</td>
<td>66x</td>
</tr>
<tr>
<td>idct FPGA</td>
<td>1.06x</td>
<td>1.06x</td>
</tr>
<tr>
<td>des FPGA</td>
<td>1.0x</td>
<td>0.04x</td>
</tr>
</tbody>
</table>
**LIQUID METAL: SUMMARY**

- Can we program HW with an object-oriented language?
  - Yes we can!
  - Steadily increasing feature set (e.g. dynamic allocation/GC)

- Many hurdles remain
  - Quality of code, area, predictability, …
  - FPGA tool flow, culture, and business model (vs. GPU)

- We’re hiring! Permanent staff, post-docs, and interns
Questions?