Compiling Parallel Algorithms to Memory Systems

Stephen A. Edwards

Columbia University

Presented at Jane Street, April 16, 2012

\((\lambda x.?)f = \text{FPGA}\)
Parallelism is the Big Question
Massive On-Chip Parallelism is Inevitable

Intel’s 48-core “Single Chip Cloud Computer”
The Future is Wires and Memory
...and it’s Already Here

Altera Stratix IV FPGA
What We are Doing About It

C et al.
gcc et al.
x86 et al.

Future Languages
Higher-level languages
Future ISAs
More hardware reality
A Functional IR
FPGAs
What We are Doing About It

Future Languages

Future ISAs

C et al.
gcc et al.
x86 et al.

Higher-level languages

More hardware reality

abstraction

time

today
What We are Doing About It

C et al.
gcc et al.
x86 et al.

Future Languages

Higher-level languages

Future ISAs

More hardware reality

A Functional IR

FPGAs

today

time
Why Functional Specifications?

- Referential transparency/side-effect freedom make formal reasoning about programs vastly easier

- Inherently concurrent and race-free (Thank Church and Rosser). If you want races and deadlocks, you need to add constructs.

- Immutable data structures makes it vastly easier to reason about memory in the presence of concurrency
Why FPGAs?

- We do not know the structure of future memory systems
  Homogeneous/Heterogeneous?
  Levels of Hierarchy?
  Communication Mechanisms?

- We do not know the architecture of future multi-cores
  Programmable in Assembly/C?
  Single- or multi-threaded?

Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.
The Memory Hierarchy is the Interesting Part
Multiprocessor Memory is a Headache

- Cache Coherency
- Write buffers
- Sequential Memory Consistency
- Memory barriers
- Data Races
- Atomic operations

Immutable data structures simplify these
The Practical Question

How do we synthesize hardware from pure functional languages for FPGAs?

Control and datapath are easy; the memory system is interesting.
To Implement Real Algorithms in Hardware, We Need

Structured, recursive data types

Recursion to handle recursive data types

Memories

Memory Hierarchy
Example: Huffman Decoder in Haskell

```haskell
data HTree = Branch HTree HTree | Leaf Char

decode :: HTree -> [Bool] -> [Char] -- Huffman tree & bitstream to symbols

decode table str = decoder table str
  where
    decoder (Leaf s) i = s : (decoder table i) -- Identified symbol; start again
    decoder _ [] = []
    decoder (Branch f _) (False:xs) = decoder f xs -- 0: follow left branch
    decoder (Branch _ t) (True:xs) = decoder t xs -- 1: follow right branch
```

Three data types: Input bitstream, output character stream, and Huffman tree
Planned Optimizations

Split Memories

Use Streams

Unroll for locality

Speculate
One Way to Encode the Types

**Huffman tree nodes: (19 bits)**

<table>
<thead>
<tr>
<th></th>
<th>8-bit character</th>
<th>(unused)</th>
<th>Leaf Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9-bit tree ptr.</td>
<td>9-bit tree ptr.</td>
<td>Branch Tree Tree</td>
</tr>
</tbody>
</table>

**Boolean input stream: (10 bits)**

<table>
<thead>
<tr>
<th></th>
<th>(unused)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bit</td>
<td>Cons Bool List</td>
</tr>
<tr>
<td></td>
<td>8-bit tail pointer</td>
<td></td>
</tr>
</tbody>
</table>

**Character output stream: (19 bits)**

<table>
<thead>
<tr>
<th></th>
<th>(unused)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8-bit character</td>
<td>Cons Char List</td>
</tr>
<tr>
<td></td>
<td>10-bit tail pointer</td>
<td></td>
</tr>
</tbody>
</table>
Intermediate Representation Desiderata

Mathematical formalism convenient for performing “parallelizing” transformations, a.k.a. parallel design patterns

- Pipeline
- Speculation
- Multiple workers
- Map-reduce
Intermediate Representation: Recursive “Islands”

\[
\text{program ::= island}\]

\[
island ::= \text{island name arg* = expr state*}
\]

\[
\text{state ::= label arg* = expr}
\]

\[
\text{expr ::= name var*}
\]

\[
| \text{let (var = expr)* in expr}
| \text{case var of (pattern -> expr)*}
| \text{var}
| \text{literal}
| \text{recurse label var* ( var* )}
| \text{return var}
| \text{goto label var*}
\]

\[
\text{pattern ::= name var* | literal | _}
\]

Group of states w/ stack
Arguments & expression
Apply a function
Parallel evaluation
Multiway conditional
Explicit continuation
Branch to another state
Constructor/literal/def.
Huffman as a Recursive Island

\[ \textbf{data} \ HTree = \text{Branch} \ HTree \ HTree \mid \text{Leaf} \ Char \]

\[ \text{decode} :: \ HTree \rightarrow [\text{Bool}] \rightarrow [\text{Char}] \]

\[ \text{decode table str} = \text{decoder table str} \]

\[ \text{where} \]

\[ \text{decoder} \ (\text{Leaf s}) \ i = \]
\[ s : (\text{decoder table} \ i) \]

\[ \text{decoder} \ _ \ [] = [] \]

\[ \text{decoder} \ (\text{Branch f} \ _) \ (\text{False}:xs) = \]
\[ \text{decoder} \ f \ xs \]

\[ \text{decoder} \ (\text{Branch} \ _ \ t) \ (\text{True}:xs) = \]
\[ \text{decoder} \ t \ xs \]

\[ \text{island decoder treep ip} = \]
\[ \text{let} \ r = \text{dec treep treep ip in return} \ r \]

\[ \text{island dec treep statep ip} = \]
\[ \text{let} \ i = \text{fetchi ip} \]
\[ \text{state} = \text{fetcht statep in} \]

\[ \text{case state of} \]
\[ \text{Leaf a} \rightarrow \text{recurse s1 a (treep treep ip)} \]
\[ \text{Branch f t} \rightarrow \]
\[ \text{case i of} \]
\[ \text{Nil} \rightarrow \text{let} \ np = \text{Nil in return} \ np \]
\[ \text{Cons x xsp} \rightarrow \]
\[ \text{case x of} \]
\[ \text{True} \rightarrow \text{goto dec treep t xsp} \]
\[ \text{False} \rightarrow \text{goto dec treep f xsp} \]

\[ \text{s1 a rp} = \text{let} \ rrp = \text{Cons a rp in return} \ rrp \]
The Basic Translation Template

Strobe-based interface: *go* indicates inputs are valid; *ready* pulses once when result is valid.
Translating Let and Case

Let makes new values available to an expression.

Case invokes one of its sub-expressions, then synchronizes.
Each island consists of expressions for each state, its own stack, and a controller that manages the stack and invokes the states.
Constructors and Memory

A constructor is a function that stores data in memory.

\[
\text{constructor } \alpha :: \alpha \to \text{Ptr } \alpha
\]

Memory access functions turn pointers into data.

\[
\text{fetch } \alpha :: \text{Ptr } \alpha \to \alpha
\]

Memory stores return an address, not take one as an argument.

Constructor is responsible for memory management.

By default, each data type gets its own memory.
Duplication for Performance

\[
\begin{align*}
\text{fib } 0 &= 0 \\
\text{fib } 1 &= 1 \\
\text{fib } n &= \text{fib } (n-1) + \text{fib } (n-2)
\end{align*}
\]
Duplication for Performance

\[
\begin{align*}
\text{fib } 0 &= 0 \\
\text{fib } 1 &= 1 \\
\text{fib } n &= \text{fib} (n-1) + \text{fib} (n-2)
\end{align*}
\]

After duplicating functions:

\[
\begin{align*}
\text{fib } 0 &= 0 \\
\text{fib } 1 &= 1 \\
\text{fib } n &= \text{fib'} (n-1) + \text{fib''} (n-2)
\end{align*}
\]

\[
\begin{align*}
\text{fib'} 0 &= 0 \\
\text{fib'} 1 &= 1 \\
\text{fib'} n &= \text{fib'} (n-1) + \text{fib'} (n-2)
\end{align*}
\]

\[
\begin{align*}
\text{fib''} 0 &= 0 \\
\text{fib''} 1 &= 1 \\
\text{fib''} n &= \text{fib''} (n-1) + \text{fib''} (n-2)
\end{align*}
\]

Here, \text{fib'} and \text{fib''} may run in parallel.
Unrolling Recursive Data Structures

Like a “blocking factor,” but more general. Idea is to create larger memory blocks that can be operated on in parallel.

Original Huffman tree type:

```haskell
data Htree = Branch Htree HTree | Leaf Char
```

Unrolled Huffman tree type:

```haskell
data Htree = Branch Htree’ HTree’ | Leaf Char

data Htree’ = Branch’ Htree” HTree’’ | Leaf Char

data Htree” = Branch” Htree HTree | Leaf’ Char
```

Recursive instances must be pointers; others can be explicit.

Functions must be similarly modified to work with the new types.
Acknowledgements

Project started while at MSR Cambridge

Satman Singh (now at Google)

Simon Peyton Jones (MSR)

Martha Kim (Columbia)