Compiling Parallel Algorithms to Memory Systems

Stephen A. Edwards
and Martha Kim

Columbia University

Spring 2012

\((\lambda x.?) f = \text{FPGA}\)
Parallelism is the Big Question
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
The Future is Already Here

Altera Stratix IV FPGA
The Memory Hierarchy is the Interesting Part
The Big Question

How Do Algorithms Manipulate Data?
Our Hypothesis

How Do Algorithms Manipulate Data?

We will only be able to answer this in very disciplined languages.

E.g., pure functional languages with immutable data structures
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
Where our work fits

**Trend 1:** Languages are becoming more abstract

**Trend 2:** ISAs are becoming less abstract and expose more hardware realities

- C et al.
- gcc et al.
- x86 et al.
- A Functional IR
- FPGAs
Multiprocessor Memory is a Headache

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

Immutable data structures simplify these
I Don’t Think We Want Laziness

Laziness has certain semantic advantages, but the bookkeeping is probably not worth it.
Approach

- 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.
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>0</th>
<th>8-bit character</th>
<th>(unused)</th>
<th>Leaf Char</th>
</tr>
</thead>
<tbody>
<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>0</th>
<th>(unused)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bit</td>
<td>8-bit tail pointer</td>
</tr>
</tbody>
</table>

### Character output stream: (19 bits)

<table>
<thead>
<tr>
<th>0</th>
<th>(unused)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8-bit character</td>
<td>10-bit tail pointer</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”

\[
\text{program} ::= \text{island}^* \\
\text{island} ::= \text{island} \ \text{name arg}^* = \text{expr state}^* \\
\text{state} ::= \text{label arg}^* = \text{expr} \\
\text{expr} ::= \text{name var}^* \\
| \ \text{let (var = expr)}^+ \ \text{in expr} \\
| \ \text{case var of (pattern -> expr)}^+ \\
| \ \text{var} \\
| \ \text{literal} \\
| \ \text{recurse label var}^* ( \ \text{var}^* ) \\
| \ \text{return var} \\
| \ \text{goto label var}^* \\
\text{pattern} ::= \text{name var}^* | \ \text{literal} | _
\]
Huffman as a Recursive Island

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

decode :: HTree -> [\text{Bool}] -> [\text{Char}]

decode table str = decoder table str
where
    decoder (Leaf s) i =
        s : (decoder table i)
    decoder _ [] = []
    decoder (Branch f _) (False:xs) =
        decoder f xs
    decoder (Branch _ t) (True:xs) =
        decoder t xs

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

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

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.

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

Memory access functions turn pointers into data.

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

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

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

After duplicating functions:

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

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

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

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:

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

Unrolled Huffman tree type:

```
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

Simon Peyton Jones