Compiling Parallel Algorithms to Memory Systems: Some Preliminary Results

Stephen A. Edwards

Columbia University

CSL Group Seminar, March 25, 2013

\((\lambda x. ?) f = \text{FPGA}\)
Functional Programs to FPGAs
Functional Programs to FPGAs
Functional Programs to FPGAs
Functional Programs to FPGAs
Functional Programs to FPGAs
Functional Programs to FPGAs
Functional Programs to FPGAs
Moore’s Law: Lots of Cheap Transistors...

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.”

Closer to every 24 months

Single-core processor performance follows the square root of area.

It takes $4 \times$ the transistors to give $2 \times$ the performance.

Fred J. Pollack, MICRO 1999 keynote. Graph from Borkar, DAC 2007
Dally: Calculation is Cheap; Communication is Costly

“Chips are power limited and most power is spent moving data

Performance = Parallelism

Efficiency = Locality

Bill Dally’s 2009 DAC Keynote, *The End of Denial Architecture*
Parallelism for Performance and Locality for Efficiency

Dally: “Single-thread processors are in denial about these two facts”

We need different programming paradigms and different architectures on which to run them.
Massive On-Chip Parallelism is Here

NVIDIA GeForce GTX-400/GF100/Fermi:
3 billion transistors, 512 CUDA cores, 16 geometry units, 64 texture units, 48 render output units, 384-bit GDDR5
The Future is Wires and Memory
A Modern High-End FPGA: Altera’s Stratix V

2500 dual-ported 2.5KB 600 MHz memory blocks; 6 Mb total
350 36-bit 500 MHz DSP blocks (MAC-oriented datapaths)
300000 6-input LUTs; 28 nm feature size
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

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

Future Languages

Higher-level languages

Future ISAs

More hardware reality

today

time

abstraction
What We are Doing About It

Future Languages

Higher-level languages

A Functional IR

More hardware reality

FPGAs

Future ISAs

today

C et al.
gcc et al.
x86 et al.
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 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
The Type System: Algebraic Data Types

Types are primitive (Boolean, Integer, etc.) or other ADTs:

\[
type ::= \text{Type} \mid Constr\ Type^* \mid \cdots \mid Constr\ Type^*
\]

Named type/primitive
Tagged union

Subsume C structs, unions, and enums
Comparative power to C++ objects with virtual methods
“Algebraic” because they are sum-of-product types.
The Type System: Algebraic Data Types

Types are primitive (Boolean, Integer, etc.) or other ADTs:

\[ \text{type ::= Type} \]
\[ \quad \mid \text{Constr Type}^* \mid \cdots \mid \text{Constr Type}^* \]

Named type/primitive
Tagged union

Examples:

\textbf{data} Intlist = Nil
\quad \mid \text{Cons Int Intlist} \quad -- Linked list of integers

\textbf{data} Bintree = Leaf Int
\quad \mid \text{Branch BinTree Bintree} \quad -- Binary tree w/ integer leaves

\textbf{data} Expr = Literal Int
\quad \mid \text{Var String}
\quad \mid \text{Binop Expr Op Expr} \quad -- Arithmetic expression

\textbf{data} Op = Add | Sub | Mult | Div
Representing Recursive Algebraic Data Types

Consider a list of integers:

```data```
```
Intlist = Nil |
    Cons Int Intlist
```

An obvious representation:

<table>
<thead>
<tr>
<th>0</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>Cons Int Intlist</td>
</tr>
</tbody>
</table>

- Usual byte-alignment unnecessary & wasteful in hardware
- Naturally stored & managed in a custom integer-list memory
- Width of pointer can depend on integer-list memory size
Syntax-Directed Translation of Expressions to Hardware

Combinational functions:

Sequential functions:
Translating Let and Case

*Let* makes all new variables available to its body.

*Case* invokes one of its sub-expressions, then synchronizes.
Removing Recursion: Recursive Fibonacci Example

\[ \text{fib } 1 = 1 \quad \text{-- Base case} \]
\[ \text{fib } 2 = 1 \quad \text{-- Base case} \]
\[ \text{fib } n = \text{fib } (n-1) + \text{fib } (n-2) \quad \text{-- Recurse twice and sum results} \]
Transform to Continuation-Passing Style

\[ \text{fib}' \ 1 \ k = k \ 1 \quad \text{-- Base case} \]
\[ \text{fib}' \ 2 \ k = k \ 1 \quad \text{-- Base case} \]
\[ \text{fib}' \ n \ k = \text{fib}' \ (n-1) \quad \text{-- First recursive call} \]
\[ \quad (\lambda n1 \to \text{fib}' \ (n-2) \quad \text{-- Second recursive call} \]
\[ \quad \quad (\lambda n2 \to k \ (n1 + n2))) \quad \text{-- Sum results} \]
\[ \text{fib} \ n = \text{fib}' \ n \ (\lambda x \to x) \]

Name intermediate results (e.g., call to \( \text{fib}' \ (n-1) \)). Pass them as arguments to \( \lambda \) terms.

Well-known technique; e.g., Appel et al.; SML/NJ compiler.
Name Lambda Terms; Capture Free Variables

call 1 k = k 1  -- Base case (return)
call 2 k = k 1  -- Base case (return)
call n k = call (n-1) (c1 n k)  -- First recursive call (call)
c1 n k n1 = call (n-2) (c2 n1 k)  -- Second recursive call (call)
c2 n1 k n2 = k (n1 + n2)  -- Sum Results (return)
c3 x = x  -- Return final result

fib n = call n c3

Each lambda term becomes its own function.
Represent Continuations with a Type; Merge Functions

\[
\begin{align*}
\text{fib}'(\text{Call } 1 \ k) &= \text{fib}'(\text{Cont } k \ 1) \\
\text{fib}'(\text{Call } 2 \ k) &= \text{fib}'(\text{Cont } k \ 1) \\
\text{fib}'(\text{Call } n \ k) &= \text{fib}'(\text{Call } (n-1) \ (C1 \ n \ k)) \\
\text{fib}'(\text{Cont } (C1 \ n \ k) \ n1) &= \text{fib}'(\text{Call } (n-2) \ (C2 \ n1 \ k)) \\
\text{fib}'(\text{Cont } (C2 \ n1 \ k) \ n2) &= \text{fib}'(\text{Cont } k \ (n1 + n2)) \\
\text{fib}'(\text{Cont } (C3) \ x) &= x
\end{align*}
\]

\[\text{fib } n = \text{fib}'(\text{Call } n \ C3)\]

\[\textbf{data} \ \text{Continuation} = \ C1 \ \text{Word8} \ \text{Continuation} \\
| \ C2 \ \text{Word32} \ \text{Continuation} \\
| \ C3\]

\[\textbf{data} \ \text{Call} = \text{Call} \ \text{Word8} \ \text{Continuation} \\
| \ \text{Cont} \ \text{Continuation} \ \text{Word32}\]
Replace Type Recursion with Pointers

Before:

```haskell
data Continuation = C1 Word8 Continuation |
                    C2 Word32 Continuation |
                    C3
```

After:

```haskell
type ContPtr = Word8        -- Pointer to a Continuation object

type ContRef = (ContPtr, ContMem)

data Continuation = C1 Word8 ContRef |
                    C2 Word32 ContRef |
                    C3
```
An Explicit “Store” Function

```haskell

type ContMem = Array ContPtr ContBits -- Model of memory

data ContBits = CB1 Word8 -- No need for “next” pointer
    | CB2 Word32 -- since these are on a stack
    | CB3

store :: Continuation -> ContRef
store c = let (p, m, c') = case c of
    C1 n (p, m) -> (p, m, CB1 n)
    C2 n1 (p, m) -> (p, m, CB2 n1)
    C3 -> (0, emptyMem, CB3) in
        let p' = p + 1 in -- Place in next memory location
        (p', m // [(p', c')]) -- Write memory

Store is more like a constructor: data in; address out.
```
An Explicit “Load” Function

\[
\text{load} :: \text{ContRef} \rightarrow \text{Continuation}
\]
\[
\text{load} (p, \ m) = \text{let } p' = p - 1 \text{ in } \\
\text{loadp} (p', \ m, \ m \ ! \ p) \quad \text{--- \textit{Successor just below us}}
\]

\[
\text{loadp} :: (\text{ContPtr}, \text{ContMem}, \text{ContBits}) \rightarrow \text{Continuation}
\]
\[
\text{loadp} (p', \ m, \ d) = \text{case } d \text{ of } \\
\text{CB1 n } \rightarrow \text{C1 n} (p', \ m) \quad \text{--- \textit{Reconstruct}}
\text{CB2 n1} \rightarrow \text{C2 n1} (p', \ m)
\text{CB3 } \rightarrow \text{C3}
\]

Broken into two functions to model synchronous RAM:

\textit{Load} runs before the clock edge (prepare address)

\textit{Loadp} runs after the clock edge (handle returned data)
Version Suitable for Hardware Translation

\[
\begin{align*}
\text{fibp (Call } & 1 \text{ kr) = fibp (Cont (load kr) 1)} \\
\text{fibp (Call } & 2 \text{ kr) = fibp (Cont (load kr) 1)} \\
\text{fibp (Call } & n \text{ kr) = fibp (Call (n }-\text{1) (store (C1 n kr)))} \\
\text{fibp (Cont (C1 n kr) n1) = fibp (Call (n }-\text{2) (store (C2 n1 kr)))} \\
\text{fibp (Cont (C2 n1 kr) n2) = fibp (Cont (load kr) (n1 + n2))} \\
\text{fibp (Cont (C3 x) = x) = x} \\
\text{fib n = fibp (Call n (store C3))}
\end{align*}
\]
Block Diagram
Concrete Representation of Types

Word8

Word32

ContRef
Concrete Representation of Types

C1

41 3433

kr 9 2 0

C2

kr n

C3

n1

Continuation

CB1

33 9 2 0

n

CB2

n1

CB3

ContBits
Concrete Representation of Types

Call

Cont

Call

82 4140 3328 8 10

kr n

Continuation n1/n2

82 754 50 431

kr n C1

kr n1 C2

kr C3
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

After duplicating functions:

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

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:

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

Unrolled Huffman tree type:

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

Recursive instances must be pointers; others can be explicit.

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

Let $\mathbf{xs} = [1,2,3]$ in
let $\mathbf{ys} = 0:\mathbf{xs}$ in
let $\mathbf{zs} = -1:\mathbf{ys}$ in
$\mathbf{ys}$

Let $\mathbf{xs} = [1,2,3]$ in
let $\mathbf{ys} = 0:\mathbf{xs}$ in
let $\mathbf{zs} = -1:\mathbf{xs}$ in
$\mathbf{ys}$

One of these has a list that behaves like a stack; the other does not.
Identifying Stacks

Let $xs = [1,2,3]$ in
let $ys = 0:xs$ in
let $zs = -1:ys$ in

$ys$

One of these has a list that behaves like a stack; the other does not.

Hint:
<table>
<thead>
<tr>
<th>Speculation</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speculation as a parallelizing optimization</td>
<td>Applying speculation to divide-and-conquer algorithms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Harnessing Recursion</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing recursion in hardware</td>
<td>Statically unrolling recursion to improve parallelism</td>
<td>Sizing, sharing, and backing the stack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compiling with Abstract Datatypes</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing high-level data types in hardware</td>
<td>Statically unrolling data structures for locality</td>
<td>Working with type-specific accelerators</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization Configurable, Distributed Memories</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicating data to improve locality</td>
<td>Sizing type-specific memories</td>
<td>Splitting and combining type-specific memories</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type-Specific Optimizations</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for Streaming Types</td>
<td>Synchronous Dataflow Support</td>
<td>Pipelining Groups of Recursive Functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPGAs-4-Kids Outreach</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic GUI Programming Environment</td>
<td>Synthesis and Downloading Flow</td>
<td>Classroom Testing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Support for Streaming Types
- Synchronous Dataflow Support
- Pipelining Groups of Recursive Functions

Duplicating data to improve locality
- Sizing type-specific memories
- Splitting and combining type-specific memories

Dennis's immutable heap

Type-Specific Optimizations
- Support for Streaming Types
- Synchronous Dataflow Support
- Pipelining Groups of Recursive Functions