Compiling Parallel Algorithms to Memory Systems

Stephen A. Edwards

Columbia University

ENS DI Group, June 26th, 2012

$$(\lambda x. ?) f = \text{FPGA}$$
Functional Programs to FPGAs

\[ \lambda f. (\lambda x. (f (x x))) \lambda x. (f (x x)) \]
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
“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.
Bacon et al.’s Liquid Metal

**Fig. 2.** Block level diagram of DES and Lime code snippet

```
public static Unsigned64 DEScoder(KeySchedule keys,
                                   Unsigned64 text)
{
    Unsigned64 block = text.permute(IP.Permutation);
    Unsigned32 R = block.extractBits(sixtyfour.b0 ,
                                     sixtyfour.b31);
    Unsigned32 L = block.extractBits(sixtyfour.b32,
                                     sixtyfour.b63);

    for (sixteen round) {
        Unsigned32 F = Fiestel(keys, round, R);
        Unsigned32 X = F ^ L;
        L = R;
        R = X;
    }
    Unsigned64 LR = makeUnsigned64(R, L);
    return LR.permute(FP.Permutation);
}
```

JITting Lime (Java-like, side-effect-free, streaming) to FPGAs

Goldstein et al.’s Phoenix

```c
int squares()
{
    int i = 0,
    sum = 0;
    for (;i<10;i++)
        sum += i*i;
    return sum;
}
```

**Figure 3:** C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)

**Figure 8:** Memory access network and implementation of the value and token forwarding network. The LOAD produces a data value consumed by the oval node. The STORE node may depend on the load (i.e., we have a token edge between the LOAD and the STORE, shown as a dashed line). The token travels to the root of the tree, which is a load-store queue (LSQ).

C to asynchronous logic, monolithic memory

Algol-like imperative language to handshake circuits

In this section we demonstrate how a circuit that performs communication over an I2C bus can be expressed using the Kiwi library. The motivation for tackling such an example arises from the fact that the typical coding style for such circuits involves hand coding state machines using nested case statements in VHDL (or equivalent features in Verilog). In particular, the sequencing of operations is implemented in the following C# code:

```csharp
public static void SendDeviceID()
{
    int deviceId = 0x76;
    for (int i = 7; i > 0; i--)
    {
        scl = false;
        sda_out = (deviceId & 64) != 0;
        Kiwi.Pause(); // Set it i-th bit of the device ID
        scl = true; Kiwi.Pause(); // Pulse SCL
        scl = false; deviceId = deviceId << 1;
        Kiwi.Pause();
    }
}
```

C# with a concurrency library to FPGAs

Arvind, Hoe et al.’s Bluespec

**GCD Mod Rule**
\[ \text{Gcd}(a, b) \text{ if } (a \geq b) \land (b \neq 0) \rightarrow \text{Gcd}(a - b, b) \]

**GCD Flip Rule**
\[ \text{Gcd}(a, b) \text{ if } a < b \rightarrow \text{Gcd}(b, a) \]

---

**Figure 1.3**  Circuit for computing \( \text{Gcd}(a, b) \) from Example 1.

Guarded commands and functions to synchronous logic

Hoe and Arvind, *Term Rewriting*, VLSI 1999
Sheeran et al.'s Lava

```
bfly :: CmplxArithmetic m
    => [CmplxSig] -> m [CmplxSig]
bfly [i1, i2] =
    do o1 <- csubtract (i1, i2)
o2 <- cplus (i1, i2)
    return [o1, o2]

bflys :: CmplxArithmetic m
    => Int -> [CmplxSig] -> m [CmplxSig]
bflys n =
    riffle >>= raised n two bfly >>= unriffle
```

Functional specifications of regular structures

Kuper et al.’s CλaSH

\[ \text{fir} (\text{State} (xs, hs)) x = (\text{State} (\text{shiftInto} x xs, hs), (x \triangleright xs) \bullet hs) \]

Fig. 6. 4-taps FIR Filter

More operational Haskell specifications of regular structures

Baaij, Kooijman, Kuper, Boeijink, and Gerards. Cλash, DSD 2010
AutoESL (Xilinx, was Cong’s xPilot)

- **SSDM (System-level Synthesis Data Model)**
  - Hierarchical netlist of concurrent processes and communication channels
  - Each leaf process contains a sequential program which is represented by an extended LLVM IR with hardware-specific semantics
    - Port / IO interfaces, bit-vector manipulations, cycle-level notations

SystemC input; classical high-level synthesis for processes

Jason Cong, presentation at ISARS 2005
Optimization of Parallel “Programs” Enables Chip Design

Sun’s UltraSPARC T2
The “Niagara 2”
8 cores; 64 threads
Built 2007, 1.6 GHz, 65 nm
Released open-source as the OpenSPARC T2
www.opensparc.net

454 000 lines of synthesizable Verilog → 503 000 000 transistors
A mix of Boolean logic and structure
The Lesson of Logic Synthesis: the Enabling Technology

How do you compile and optimize a digital logic circuit?

\[
\begin{align*}
    f_1 &= abcd + abce + \overline{abcd} + \overline{ab\bar{c}d} + \overline{ac} + cd\bar{f} + ab\bar{c}d\bar{e} + ab\bar{c}d\bar{f} \\
    f_2 &= bdg + \overline{bd}f\bar{g} + \overline{bd}g + \overline{bd}eg
\end{align*}
\]

After Brayton et al.'s class on Multi-Level Logic Synthesis
The Lesson of Logic Synthesis: the Enabling Technology

How do you compile and optimize a digital logic circuit?
Use a simple, formal model and automate it.

\begin{align*}
  f_1 &= abcd + abce + \overline{abc}d + \overline{ab}cd + \overline{ac} + cd\overline{f} + ab\overline{cd}\overline{e} + ab\overline{cd}f \\
  f_2 &= bd\overline{g} + \overline{bd}f\overline{g} + \overline{bd}\overline{g} + \overline{bd}eg \\

  &\text{Minimize} \\
  f_1 &= bcd + bce + \overline{bd} + \overline{ac} + cd\overline{f} + ab\overline{cd}\overline{e} + ab\overline{cd}f \\
  f_2 &= bd\overline{g} + df\overline{g} + \overline{bd}g + \overline{deg} \\

  &\text{Factor} \\
  f_1 &= c(b(d + e) + \overline{b}(d + f) + \overline{a}) + a\overline{c}(bd\overline{e} + bd\overline{f}) \\
  f_2 &= g(d(b + f) + \overline{d}(b + e)) \\

  &\text{Decompose} \\
  f_1 &= c(x + \overline{a}) + acx \\
  f_2 &= gx \\
  x &= d(b + f) + \overline{d}(\overline{b} + e)
\end{align*}

After Brayton et al.'s class on Multi-Level Logic Synthesis
High-Level Synthesis: Adding Time Meant Scheduling

Figure 2: (a) VHDL description; (b) Separate control and data-flow graphs

The High-Level Synthesis Lessons

Don’t Start From C

“The so-called high-level specifications in reality grew out of the need for simulation and were often little more than an input language to make a discrete event simulator reproduce a specific behavior.”


Don’t Forget Memory

Goldstein et al.’s Phoenix synthesized asynchronous hardware from ANSI C. Required heroic work [CGO 2003] to recover any parallelism.
Our Approach

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

Today

Abstraction

Time
Our Approach

Future Languages

Higher-level languages

Future ISAs

More hardware reality

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

Future Languages

Time

Today
Our Approach

Future Languages

Future ISAs

A Functional IR

FPGAs

Higher-level languages

More hardware reality
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.
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
Let’s Talk Details
Let’s Talk Details
Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler’s “Core” representation

\[ expr ::= name \ var^* \]

Function call

Includes primitive arithmetic operators and type constructors

Non-tail-recursive calls generally inlined to improve parallelism; Mycroft and Sharp’s [IWLS 2000] propose sharing policies

True recursion transformed to tail recursion with a stack
Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler’s “Core” representation

\[
expr ::= \text{name} \ var^* \\
| \text{let } (\text{var} = expr)^+ \text{ in } expr
\]

Function call
Parallel evaluation

Parallelism and sequencing:

\[
\begin{aligned}
\text{let } v_1 &= e_1 \\
v_2 &= e_2 \\
v_3 &= e_3 \text{ in } e
\end{aligned}
\]

\[
\begin{aligned}
e_1 & \\
e_2 & \\
e_3 & \\
\end{aligned}
\]

evaluated in parallel, then \( e \)
Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler’s “Core” representation

\[
expr ::= name \ var^* \\
| \ \textbf{let} \ (var = expr)^+ \ \textbf{in} \ expr \quad \text{Function call} \\
| \ \textbf{case} \ var \ \textbf{of} \ (pat \rightarrow expr)^+ \quad \text{Parallel evaluation} \\
\]

\[
pat ::= \text{literal} \quad \text{Exact match} \\
| \ _ \quad \text{Default} \\
| \ \textbf{Constr.} \ (var \mid \text{literal} \mid \_)^* \quad \text{Match a tagged union} \\
\]

Evaluate and return one of the expressions based on the pattern
Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler’s “Core” representation

\[
expr ::= \text{name \ var}^* \\
| \text{let } (\var = expr)^+ \text{ in } expr \\
| \text{case } var \text{ of } (pat -> expr)^+ \\
| \var \\
| \text{literal}
\]

\[
pat ::= \text{literal} \\
| _ \\
| \text{Constr. (var | literal | _)}^*
\]

Function call
Parallel evaluation
Multiway conditional
Variable reference
Literal value
Exact match
Default
Match a tagged union
The Type System: Tagged Unions

Types are primitive (Boolean, Integer, etc.) or tagged unions:

\[
\text{type ::= Type} \quad \text{Named type/primitive}
\]
\[
| \quad \text{Constr Type}^* \quad | \cdots \quad | \quad \text{Constr Type}^* \quad \text{Tagged union}
\]

Subsume C structs, unions, and enums

Comparable power to C++ objects with virtual methods

Sometimes called “algebraic data types”: sums of products
The Type System: Tagged Unions

Types are primitive (Boolean, Integer, etc.) or tagged unions:

\[
\text{type ::= Type} \quad \text{Named type/primitive}
\]
\[
\quad \mid \text{Constr Type}^{*} \mid \cdots \mid \text{Constr Type}^{*} \quad \text{Tagged union}
\]

Examples:

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

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

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

\[
\text{data Op} = \text{Add} | \text{Sub} | \text{Mult} | \text{Div}
\]
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.
Representing Recursive Algebraic Data Types

Consider a list of integers:

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

An obvious representation:

- 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
Removing Recursion: Recursive Fibonacci Example

Starting point: a dumb way to compute Fibonacci numbers

\[ fib \ 1 = 1 \]
\[ fib \ 2 = 1 \]
\[ fib \ n = fib \ (n-1) + fib \ (n-2) \]
Removing Recursion: Recursive Fibonacci

Reformatting

\[
\begin{align*}
\text{fib } 1 & = 1 \\
\text{fib } 2 & = 1 \\
\text{fib } n & = \text{fib } (n-1) + \\
& \quad \text{fib } (n-2)
\end{align*}
\]
Removing Recursion: Continuation-Passing Style

In continuation-passing style (the “and then?” transformation):

\[
\begin{align*}
\text{fib1} \ 1 \ c & = \ c \ 1 \\
\text{fib1} \ 2 \ c & = \ c \ 1 \\
\text{fib1} \ n \ c & = \text{fib1} \ (n-1) \\
& \quad \ (\ n1 \to \ \text{fib1} \ (n-2) \\
& \quad \quad \ (\ n2 \to \ \ c \ (n1 + n2)))
\end{align*}
\]

\[
\text{fib} \ n \ = \ \text{fib1} \ n \ (\ x \to x)
\]

\-- Calls made sequential
\-- Intermediates named
\-- Add scheduled last
\-- Wrapper
Removing Recursion: Naming Functions

Naming functions; converting unbound variables to arguments:

\[
\begin{align*}
\text{fib1 } 1 & \quad c \quad = \quad c \; 1 \\
\text{fib1 } 2 & \quad c \quad = \quad c \; 1 \\
\text{fib1 } n & \quad c \quad = \quad \text{fib1} \ (n-1) \ (\text{fib2} \ n \ c) \quad \text{--- Unbound variables passed} \\
\text{fib2 } n & \quad c \; n1 \quad = \quad \text{fib1} \ (n-2) \ (\text{fib3} \ n1 \ c) \quad \text{--- Lambdas named} \\
\text{fib3 } n1 & \quad c \; n2 \quad = \quad c \ (n1 + n2) \\
\text{fib } n & \quad = \quad \text{fib1} \ n \ \text{fib0} \\
\text{fib0 } n & \quad = \quad n \quad \text{--- Identity function named}
\end{align*}
\]
Removing Recursion: True Recursion to Tail Recursion

Introducing a stack; merging functions

\[ f \ (Fib1\ 1\ c) = f \ (Cont\ c\ 1) \quad --\ Single\ function \]
\[ f \ (Fib1\ 2\ c) = f \ (Cont\ c\ 1) \quad --\ Continuation\ the\ stack \]
\[ f \ (Fib1\ n\ c) = f \ (Fib1\ (n-1)\ (Fib2\ n\ c)) \]
\[ f \ (Cont\ (Fib2\ n\ c)\ n1) = f \ (Fib1\ (n-2)\ (Fib3\ n1\ c)) \]
\[ f \ (Cont\ (Fib3\ n1\ c)\ n2) = f \ (Cont\ c\ (n1 + n2)) \]
\[ f \ (Fib\ n) = f \ (Fib1\ n\ Fib0) \]
\[ f \ (Cont\ Fib0\ n) = n \]

-- Continuations (references to the lambda expressions)

\textbf{data} Stack = Fib2 \ Int \ Stack \quad --\ fib2\ n\ c
\ |
\textbf{data} Stack = Fib3 \ Int \ Stack \quad --\ fib3\ n1\ c
\ |
\textbf{data} Stack = Fib0 \quad --\ identity\ function\ (bottom\ of\ stack)

-- Invoke a named function or a continuation

\textbf{data} Action = Fib \ Int \quad --\ fib\ n\ (outside\ call)
\ |
\textbf{data} Action = Fib1 \ Int \ Stack \quad --\ fib1\ n\ c\ (recursive\ call)
\ |
\textbf{data} Action = Cont \ Stack \ Int \quad --\ c\ (...)\ (invoke\ continuation)
Fibonacci Datapath

\[ f (Fib1 \ 1 \ c) = f (Cont \ c \ 1) \]
\[ f (Fib1 \ 2 \ c) = f (Cont \ c \ 1) \]
\[ f (Fib1 \ n \ c) = f (Fib1 (n-1) (Fib2 n c)) \]
\[ f (Cont (Fib2 n c) \ n1) = f (Fib1 (n-2) (Fib3 n1 c)) \]
\[ f (Cont (Fib3 n1 c) \ n2) = f (Cont c (n1 + n2)) \]
\[ f (Fib \ n) = f (Fib1 n Fib0) \]
\[ f (Cont \ Fib0 \ n) = n \]

\textbf{data} \ Stack = Fib2 \textbf{Int} \ Stack \quad \textbf{data} \ Action = Fib \textbf{Int} \\
\hspace{1cm} | \quad Fib3 \textbf{Int} \ Stack \\
\hspace{1cm} | \quad Fib0 \\
\hspace{1cm} | \quad Fib1 \textbf{Int} \ Stack \\
\hspace{1cm} | \quad Cont \ Stack \textbf{Int}
Implementing the Stack in Hardware

This uses a list-like stack data type:

\[
data \ Stack = Fib2 \ \textbf{Int} \ \textbf{Stack} \\
\quad | \quad Fib3 \ \textbf{Int} \ \textbf{Stack} \\
\quad | \quad Fib0
\]

A naïve, but correct, way to implement it in hardware:

Encoded return address

Function activation record
Specializing Data Types: Recovering a Classical Stack

\[ \text{Fib3 42 (Fib2 17 (Fib3 8 (Fib3 2 Fib0))))} \]
Specializing Data Types: Recovering a Classical Stack

\[ Fib3 \ 42 \ (Fib2 \ 17 \ (Fib3 \ 8 \ (Fib3 \ 2 \ Fib0))) \]

The only “pop” operation discards the previous top-of-stack
\[ f \ (Cont \ (Fib3 \ n1 \ c) \ n2) = f \ (Cont \ c \ (n1 + n2)) \]
so this code will never generate a tree.
Sequential memory allocation is safe.
Specializing Data Types: Recovering a Classical Stack

\[ Fib_3 \ 42 \ (Fib_2 \ 17 \ (Fib_3 \ 8 \ (Fib_3 \ 2 \ Fib_0))) \]

| 4: | 10 | 42 | 3 |
| 3: | 01 | 17 | 2 |
| 2: | 10 | 8  | 1 |
| 1: | 10 | 2  | 0 |
| 0: | 00 |

Sequential memory allocation makes “next” pointers predictable...
Specializing Data Types: Recovering a Classical Stack

\[ \text{Fib3} \ 42 \ (\text{Fib2} \ 17 \ (\text{Fib3} \ 8 \ (\text{Fib2} \ 2 \ \text{Fib0}))) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td>3:</td>
<td>01</td>
<td>17</td>
</tr>
<tr>
<td>2:</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>1:</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>0:</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

...so there is no need to store them.

Constructor (Fib0) always returns 0.

Constructors (Fib2/3 \( n \) \( s \)) writes (Fib2/3 \( n \)) at \( s + 1 \) and returns \( s + 1 \).

Reading 0 returns Fib0; reading \( s \) returns (Fib2/3 \( n \) \( s - 1 \)).
Specializing Data Types

Stacks are the tip of the iceberg

Synthesizing custom memory systems for specific types is a key goal of this project

Shape Analysis relevant here

This is a simple case; a simple, mathematical IR enables such clever optimizations.

Imagine trying to do this in C.
Unrolling Code for Better Parallelism

$fib\ 0 = 0$
$fib\ 1 = 1$
$fib\ n = fib\ (n-1) + fib\ (n-2)$

Yet because they share $fib$, they are performed sequentially.

$fib\ (n-1)$ and $fib\ (n-2)$ are functionally independent.
Unrolling Code for Better Parallelism

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

By unrolling the recursion once, \(\text{fib}'\) and \(\text{fib}''\) run in parallel.

A further improvement: balance the work done by \(\text{fib}'\) and \(\text{fib}''\)
Unrolling Types for Better Locality

```haskell
data Stack = Fib2 Int Stack
           | Fib3 Int Stack
           | Fib0
```

Each Stack object naturally represents a single activation record.
A similar unrolling amounts to packing records that can be processed in parallel.

Abstract data types enables this.

Imagine trying to do this safely in a C compiler.
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 = decode 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
```
Optimizations

1. Split Memories
2. Use Streams
3. Unroll for locality
4. Speculate
Target Applications

- “Data-parallel irregular applications [that] manipulate large pointer-based data structures like graphs”
  [Pingali et al.’s Galois project]

- Datatype accelerators
  Hash tables, Balanced trees, Heaps

- Application-domain accelerators
  Relational databases, Cryptography, Data compression

- Non-scientific computing: the stuff that’s hard for vector units and GPGPUs
Acknowledgements

Project started while at MSR Cambridge

Satnam Singh (now at Google)

Simon Peyton Jones (MSR)

Martha Kim (Columbia)