Circuit Parallelism in Haskell

Andreas Koltes and John O'Donnell

University of Glasgow

IFL, September 2007

Thesis

- *Circuit parallelism is a useful complement to task and data parallelism.*
- Two effective ways to exploit increasing chip density are
  - Multicore processors, good for task parallelism
  - General programmable logic (e.g. FPGA), good for circuit parallelism

How circuits can help

- As functional units that speed up common functions
- As high performance engine for fine-grain data parallelism
- To perform exotic computations (randomised circuits, asynchronous relaxation, . . .)

The approach

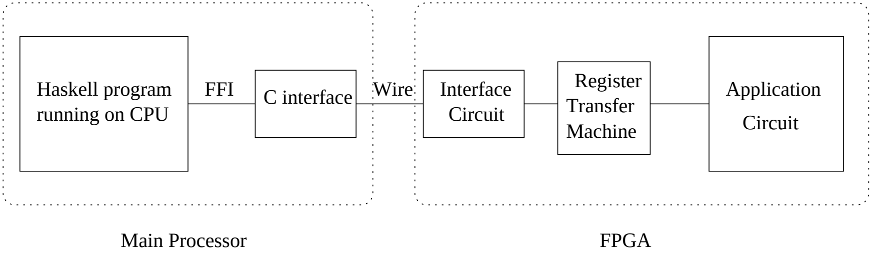
You partition your program into

- a portion that runs on an ordinary computer
- a portion that runs as a digital circuit

Both parts can be written in Haskell. The software part is compiled, the circuit part is transformed into an FPGA program.

No automatic support! But you can still reason about the whole program, refactor it, etc.

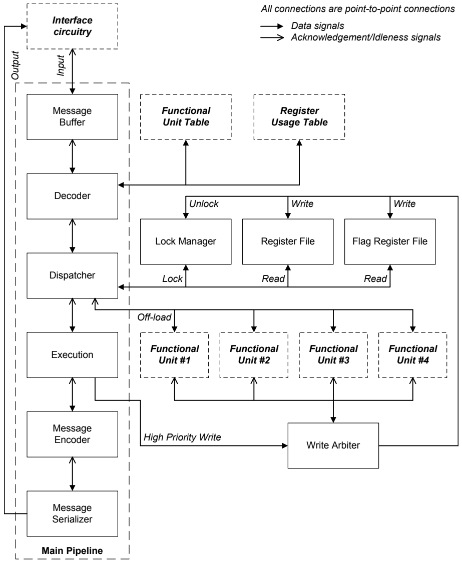
Organisation of the prototype system



Using an Altera Cyclone FPGA

This is only a running prototype! Recent high performance FPGAs will be a lot faster.

Register transfer machine



A case study

We have developed a running system, and applied it to index interval selection sort, a data parallel algorithm.

Programming Models

- Task parallelism. Partition computation into subtasks that can be computed simultaneously.
- Data parallelism. Partition computation into operations over aggregate data structures.

Each can be implemented on top of the other

Implementing data parallelism

There are many ways to achieve parallelism organised around data structures

- Threads organised to operate on aggregates
- Data parallelism on top of tasks
- SIMD architecture
- Programmable circuits
- Circuit design

Obvious data parallelism

Usually data parallelism has been limited to the most obvious cases:

- iterations across dense arrays
- vector architectures
- obvious algorithms for SIMD architectures

Some real systems betray this limitation, e.g. instead of providing scanl as a primitive, a family of first order primitives are given:

- scanl $(+)$
- scanl (*)
- scanl $(\vee)$
- scanl $(\wedge)$

Beyond basic iterations

The most interesting data parallel algorithms require

- higher order functions – e.g. the function argument to scan is fairly complex, not just $(+)$
- more general combinators – sweep, bidirectional scan, etc, not just scan

These are among the reasons that a functional language is well suited for data parallelism

Index interval selection

- an associative array data structure
- a selection algorithm
- a data parallel sort algorithm, reminiscent of both quicksort and selection sort

(John O'Donnell, Glasgow FP Workshop, 1988)

The problem

Given an array $x_0, \ldots, x_{n-1}$.

We want to perform various operations:

- Sort it
- Selection: given index i (for $0 \leq i < n$), find the ith largest element.
- Location: given an element x, find its index in the sorted array.

The idea

An associative algorithm: identify by other data associated with it, rather than by where it is in the machine

For each value in the array, attach its index.

But initially we don't know its index!

$\Rightarrow$ represent partial information about index: an index interval (low,high) means that the true index lies in this interval

Initially, give *every* element the interval $(0, n-1)$, since it could be anywhere!

Representation

Each "processor" cell consists of a few registers (on the order of 100 bits), and a little logic (an adder/comparitor, some multiplexors).

```
data Cell = Cell
      { val      :: Value,      — data value
        lb       :: Bound,      — lower bound of index interval
        ub       :: Bound,      — upper bound of index interval
        select   :: Bool,       — cell is currently active
        flag     :: Bool }      — temp

type State = [Cell]
```

Refining an interval

1. Activate all the cells with interval (l, h)
2. Pick a splitter
3. Count the number n of active values smaller than splitter, let $k = l + n$.
4. Refinement: (l, h) becomes one of $(l, k - 1)$, (k, k), or $(k + 1, h)$.

Each of these steps takes just a small constant number of clock cycles.

Organisation of the algorithm

- A microinstruction set
- A control algorithm (expressed as monadic operation, transformed into microcontroller in the register transfer machine on the FPGA)
- A datapath (specified in Hydra, transformed into FPGA program)

The microinstruction set

They operate in parallel on all the cells.

Several groups: input/output, selecting cells to operate on, conditional operations, and global operations (using a tree network)

There are a lot of tradeoffs here:

- If the microinstructions do a lot, the number of cycles is reduced but the circuit density suffers and cycle time may grow
- There are many other data parallel algorithms you might want to combine with the index interval family

Input/Output

- shift :: IORef State → (Value,Bound,Bound) → IO (Value,Bound,Bound)
  The cell states are shifted right; enables the computer to initialise and read out state
- readSel :: IORef State → IO (Maybe (Value,Bound,Bound))
  Read out the state of the cell with select flag set

Selecting cells

The select flag determines whether cells perform conditional operations.

- setSelect :: IORef State → Bool → IO ()
  initialise the flag
- save :: IORef State → IO ()
  copy select flag into a temporary
- restore :: IORef State → IO ()
  copy temporary back into select

Conditionals

Comparisons

- compare :: IORef State → (Value→Value→Bool) → Value → IO ()
  compare local value with broadcast value, set select
- match :: IORef State → (Cell→Bound) → Bound → IO ()
  compare index interval bound

Conditional operations

- condSetUB :: IORef State → Bound → IO ()
  set upper bound if select
- condSetLB :: IORef State → Bound → IO ()
  set lower bound if select

Global operations

These use a tree network, with the cells as leaves, to perform computations involving all the cells.

- count :: IORef State → IO Int
  returns the number of cells that have select set to True
- resolve :: IORef State → IO Bool
  clear select in all cells except leftmost one where it was set
- imprecise :: IORef State → IO ()
  set select where index interval is imprecise

XiQuicksort algorithm

Each sort step chooses an imprecise index interval and refines it.

```
xiSort :: IORef State → IO ()
xiSort sr =
    do a ← choose sr
       case a of
          Nothing → return ()
          Just (v,l,u) →
              do split sr v l u
                 xiSort sr
```

Choosing an interval to refine

Search for an imprecise interval, pick one cell with this interval if there are several, and read out its data value. This will be the splitter.

```
choose :: IORef State → IO (Maybe (Value,Bound,Bound))
choose sr =
    do imprecise sr
       resolve sr
       result ← readSel sr
       return result
```

Splitting the interval

```
split :: IORef State → Value → Bound → Bound → IO ()
split sr s l u =
    do setSelect sr True
        match sr lb l               – check lower bound
        match sr ub u               – and upper bound
        save sr                     – iff cell matchs (lb,ub) bounds
        compareVal sr (<) s         – select indicates match with val<splitter
        n ← count sr                – number of matches < splitter
        let k = l + n               – calculate exact index of splitter
        condSetUB sr (k − 1)        – where val<s, update the upper bound
        restore sr
        compareVal sr (>) s         – where val>s,
        condSetLB sr (k + 1)        – update the lower bound
        restore sr
        compareVal sr (==) s        – where val=s,
        condSetLB sr k              – update the lower bound
        condSetUB sr k              – and the upper bond
```

Performance

- Cycle time
  - Clock speed on FPGA is 14.3 MHz (up to 46 MHz possible)
  - Circuit is not optimised, 100MHz may be reachable
  - Recent high performance FPGAs are a lot bigger and faster
- Number of cycles
  - Selection step: 17 cycles
  - Sorting step: 16 cycles
  - Find sorted index of value: 16 cycles

Some observations

- There is a complexity speedup from the parallelism.
- In analysing complexity, you have to be careful about the cost model you're using!
- The selection algorithm is automatically memoising: the work done to perform a selection has the side effect of refining many index intervals, making subsequent selections faster.
- There are many more rich data parallel algorithms!

Conclusion

- You can accelerate parts of a program with circuit parallelism, and both parts of the program can be written in the same language.
- Some data parallel algorithms are massively parallel with very fine grain, and are better suited for circuit parallelism than threads
- Haskell can express such algorithms as circuits — graph reduction is not the only way to run a program!
- Some data parallel algorithms only make sense with circuit parallelism: on a coarse grain multicore they would be inefficient.