Introduction to Systolic Design

Wayne Luk
wl@doc.ic.ac.uk
Imperial College
March 2002

Overview

- history and motivation for systolic arrays
- systolic array features
- systolic design techniques
  - composing regular components
  - retiming
  - slowdown
  - clustering
  - bit-level design
  - systolic state machines
- topics not covered
- further reading

History and motivation

- introduced by Kung and Leiserson, 1978
- designs for matrix computations
- illustrated by snapshots of operation

motivations:
- improve performance of special-purpose systems
  - e.g. maximise processing per memory access
- reduce their design and implementation costs
  - e.g. exploit latest technology: FPGAs

Systolic arrays

Systolic: rhythmical contraction; describes the contraction of the heart forcing blood onward and keeping up the circulation.

Array: multiple PEs to maximise processing per memory access.

Systolic array features

- multiple use of each input data item
- extensive concurrency; usually by pipelining
- a few types of simple cells
- simple and regular data and control flow

these result in:
- simple and reduced costs
- high performance
- modular and expandable

Field-Programmable Gate Arrays (FPGAs)

- combine software flexibility and hardware performance
- off-the-shelf parts, factory-tested, many varieties
- matrix of cells, each has programmable function unit
- programmable connections
  - nearest neighbour / local / global routing
- technology: 10 million-gate FPGA, GHz clock speed
- good platform for implementing systolic designs
  - array structure
  - increased flexibility, adaptable at run time
  - reduced design/implementation time and cost
Applications

- signal, image, video, multimedia, numerical processing
  - add, multiply, divide, square root...in various number systems
  - recursive and non-recursive, linear and non-linear filtering
  - DFT, FFT, FHT, DCT, DWT, FNT
  - matrix and graph algorithms, algebraic path problem
  - neural nets, motion estimation, shading, texture mapping

- non-numerical processing
  - sorting, searching, matching, priority queue, LRU
  - dynamic programming
  - data compression and encryption
  - discrete event simulation
  - database operations

Array shapes: linear and rectangular

Linear array: chain

Rectangular array

Hexagonal array

Hexagonal array

Triangular-shaped arrays

Example: matrix vector multiplier

\[ A \mathbf{x} = \mathbf{y}, \quad y_i = a_{i0} x_0 + a_{i1} x_1 + a_{i2} x_2 + a_{i3} x_3 \]

- D=delay=register
- constant multiplier:
  \[ x_i \rightarrow \frac{a_i}{a_0} x_i \]
- does it work?
- are there alternatives?
Example: bit-level convolver

Systolic design techniques

- systematic design of systolic arrays
  - transform obvious design to efficient but less obvious designs
  - circuit-oriented block diagram approach
  - simple ideas behind design automation algorithms

- techniques
  - composing regular components: focus on desired behaviour
  - retiming: relocate latches in a circuit
  - slowdown: replicate latches
  - clustering: arrange hierarchy for pipelining
  - bit-level design: useful for hardware libraries
  - systolic state machines: pipeline state transition functions

- illustrated by simple examples
  - convolution, matrix vector multiplication, sorting

Convolver: composing regular components

Pipelining

- clock speed depends on longest combinational path

Pipelining

- insert latches between circuits to increase throughput
- but may also increase
  - area, power consumption, latency
- retiming: graphical method, introduce/relocate latches to improve performance/regularity and preserve behaviour
- may apply this method several times; avoid overkill

Retime a chain

- idea: introduce anti-latch which cancels effect of a latch
- OK to have anti-latch at inputs or outputs
- graphical contours linking introduction of latch/anti-latch
- pre-condition:
  - given
  - then
Retime a row

- pre-condition:
  given  
  \[ R \rightarrow \]

- then  
  \[ R \rightarrow R \rightarrow R \rightarrow \]

Remove triangular-shaped array of registers

Cu0: functional description
\[ y_i = \sum_{j=0}^{N-1} x_i \cdot w_j \]
\[ = x_i \cdot w_0 + x_i \cdot w_1 + \ldots \]

Retime top part of convolver

Uni-directional flow convolver

- semi-systolic
- regular connection
- one type of cell: CuCell1
- speed?
- impact of array size?
- concurrency?

Improving speed: retime the macs

Retime the macs: pipelined bottom
Retiming both top and bottom

Uni-directional flow systolic convolver

Pipeline the multiplier and adder

Design tree

Characterise designs

Pipelining a grid
Pipelining a grid

Combinational matrix vector multiplier
- \( \mathbf{Ax} = \mathbf{y} \), \( y_i = a_{i0} x_0 + a_{i1} x_1 + a_{i2} x_2 + a_{i3} x_3 \)
- unpipelined
- constant multiplier:
  \[
  \begin{align*}
  x_i & \to a_{i} x_i \\
  x_0 & \to a_{i0} x_0 \\
  x_1 & \to a_{i1} x_1 \\
  x_2 & \to a_{i2} x_2 \\
  x_3 & \to a_{i3} x_3 \\
  y_i & \to y_i
  \end{align*}
  \]

Matrix vector multiplier: contours
- \( \mathbf{Ax} = \mathbf{y} \), \( y_i = a_{i0} x_0 + a_{i1} x_1 + a_{i2} x_2 + a_{i3} x_3 \)
- constant multiplier:
Matrix vector multiplier: adding registers

- \( Ax = y \)
- \( D = \text{delay=register} \)
- constant multiplier:

\[
\begin{align*}
\mathbf{x}_i \rightarrow a_{ij} \mathbf{x}_i \\
\end{align*}
\]

Pipelined matrix vector multiplier

- \( Ax = y \)
- \( D = \text{delay=register} \)
- constant multiplier:

\[
\begin{align*}
\mathbf{x}_i \rightarrow a_{ij} \mathbf{x}_i \\
\end{align*}
\]

Uni-directional flow systolic convolver

Convolver with counter-flowing data

\[
\begin{align*}
\mathbf{w}_0 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_1 & \rightarrow \text{mac}^n \\
\mathbf{w}_2 & \rightarrow \text{mac}^n \\
\mathbf{w}_3 & \rightarrow \text{mac}^n \\
\end{align*}
\]

Condensor with counter-flowing data

\[
\begin{align*}
\mathbf{w}_0 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_1 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_2 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_3 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\end{align*}
\]

\[ y \]

Ch1

Convolver with counter-flowing data

\[
\begin{align*}
\mathbf{w}_0 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_1 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_2 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\mathbf{w}_3 & \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \rightarrow \text{mac}^n \\
\end{align*}
\]

\[ y \]

Ch2

still not fully pipelined!
**Slowdown**

- n-slow: can replace each latch by n latches in series, provided that (n-1) extra values are inserted between successive inputs; similarly for outputs
- graphically: introduce additional D or D-1 by replacing each D (or D-1) by n copies in series
- interpretation:
  - interleaved n data streams/computations concurrently
  - sample output every n cycles to get result of each computation

**Derive fully-pipelined convolver: slowdown**

**Retime after slowdown**

**Fully-pipelined convolver**

**Pipelining may become less effective**

<table>
<thead>
<tr>
<th>Throughput (MHz)</th>
<th>Non-pipelined</th>
<th>Fully pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree of Pipelining</td>
<td>1/NT_{cell}</td>
<td>1/(NT_{cell}+T_{latch})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Throughput (MHz)</th>
<th>Non-pipelined</th>
<th>Fully pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree of Pipelining</td>
<td>1/NT_{cell}</td>
<td>1/(NT_{cell}+T_{latch})</td>
</tr>
</tbody>
</table>

- input-output speed limit
- clock skew
- clock rise and fall times significant
- control degree of pipelining

**Controlled pipelining: clustering**

- cluster elements into groups, and retime the groups
- vary size of groups to control degree of pipelining
- reason about regular patterns of pipelining

\[ R^{KN} = (R^2; D^2; D^2) \]

\[ R^{KN} = M \]

\[ K = 1, \quad N = M \]

| R | R | R | R | R | R | R | R | R | R | R |

\[ \text{fully-pipelined: } K = 1, \quad N = M \]

\[ \text{non-pipelined: } K = M, \quad N = 1 \]
Convolver with counter-flowing data

\[ x \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow y \]

Cb1

Partially-pipelined designs

- Cb4

\[ \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \]

- Cb5

\[ \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \rightarrow \text{mac}^w \]

- boundary conditions not shown

Clustering rectangular array

Retime around the contours

Result

Result
**Retiming through the contours**

**Result**

**Lead**

**Trail**

**Pipelined rectangular array**
Bit-level convolver: 1-bit $w$ and $x$

$\begin{align*}
x & \quad \rightarrow \quad D \\
\times & \quad \rightarrow \quad A \\
y & \quad \rightarrow \quad H \\
\end{align*}$

becomes

$\begin{align*}
x & \quad \rightarrow \quad A \\
\times & \quad \rightarrow \quad F \\
y & \quad \rightarrow \quad F \\
\end{align*}$

Bit-level convolver: 1-bit $w$

$\begin{align*}
x & \quad \rightarrow \quad D \\
\times & \quad \rightarrow \quad A \\
y & \quad \rightarrow \quad H \\
\end{align*}$

becomes

$\begin{align*}
x & \quad \rightarrow \quad A \\
\times & \quad \rightarrow \quad F \\
y & \quad \rightarrow \quad F \\
\end{align*}$

Bit-level convolver: increase regularity

$\begin{align*}
x & \quad \rightarrow \quad \text{Array of } A \\
\times & \quad \rightarrow \quad \text{Array of } F \\
y & \quad \rightarrow \quad \text{Array of } F \\
\end{align*}$

becomes

$\begin{align*}
x & \quad \rightarrow \quad \text{Array of } A \\
\times & \quad \rightarrow \quad \text{Array of } F \\
y & \quad \rightarrow \quad \text{Array of } F \\
\end{align*}$

Refinement to bit level

implement word level cell (assume single bit $w$)

$\begin{align*}
x & \quad \rightarrow \quad \text{CbCell} \\
\times & \quad \rightarrow \quad \text{CbCell} \\
y & \quad \rightarrow \quad \text{CbCell} \\
\end{align*}$

where

$\begin{align*}
x & \quad \rightarrow \quad \text{CbCell} \\
\times & \quad \rightarrow \quad \text{CbCell} \\
y & \quad \rightarrow \quad \text{CbCell} \\
\end{align*}$

Pipelining strategy 1

1. Slowdown by 2 (double all latches)
2. Retime to get fully-pipelined circuit

Design Cbb1

$\begin{align*}
y_n^1 & = \sum w_i x_{n \times i} \\
\end{align*}$
Pipelining strategy 2

- pipeline clusters of \( K \) by \( K \) cells (\( K > 1 \)) e.g. \( K = 2 \)

Design Cbb2

Summary: optimising digital designs

useful rules:
- retiming: can add a latch at all inputs, provided adding an anti-latch at all outputs
- use clustering to control degree of pipelining
- \( n \)-slow: can replace each latch by \( n \) latches in series, provided that \((n-1)\) additional values are inserted between successive inputs; similarly for outputs

Perfomance and resource usage

<table>
<thead>
<tr>
<th>Design</th>
<th>min clock period</th>
<th>latency</th>
<th>number of latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cu2</td>
<td>( T_m + T_a )</td>
<td>( N-1 )</td>
<td>( N(N+1)/2 )</td>
</tr>
<tr>
<td>Cu3</td>
<td>( T_m + T_a )</td>
<td>( 2N-1 )</td>
<td>( N(N+5)/2 )</td>
</tr>
<tr>
<td>Cu4</td>
<td>( \max(T_m, T_a) )</td>
<td>( 2N )</td>
<td>( N(N+7)/2 )</td>
</tr>
</tbody>
</table>

Note that the minimum clock period for Cu2 should be \((N-1)T_p + T_m + T_a\), where \( T_p \) is the delay across the wiring cell

State machines

- state-transition function \( R \) usually includes an output part \( y \) and a next state part \( s \)

Simple state machines

- counter
- sorter
- priority queue
- LRU processor
Simple state machines

Decomposing state machines

Simple state machines

Simple state machines
Example: inserter

- Insert an element into an ordered list to form an ordered list: insert <3, <1, 2, 5, 6>> = <<1, 2, 3, 5>, 6>

Insertion sort

- State registers initialised with +∞
- Load n values to be sorted cycle by cycle
- Load n -∞ values to extract the sorted result

Insertion sort (cont)

- To extract the sorted sequence cycle by cycle, input -∞

Decomposing the sorter
Decomposing the sorter

Summary: systolic state machines
- start with state-transition function
- include loop and state registers to ensure computing the desired function
- make sure that registers are initialised appropriately
- decompose a large state machine into a collection of small state machines
- pipeline the collection of state machines as required

Topics not covered
- composite and hybrid systolic systems: ensure boundary conditions match
- multi-dimensional arrays: nearest-neighbour connections in 3D
- reconfigurable designs: pipeline morphing and virtual pipelines
- space optimisation techniques: digit serial
- systolic array implementations and platforms: iWarp, Splash, Pitchard, RC1000, and Sonic
- languages and tools for systolic design and verification: Ruby, Pebble, Lava, CSP, CirCal, Alpha

Further reading
  - excellent introductions to systolic architectures
  - comprehensive reference textbook
  - research on theory and practice of systolic design
- Proceedings of IEEE International Conference on Field-Programmable Custom Computing Machines.
  - research on systolic systems implemented by FPGAs