The Power of Streams on the SRC MAP®

Wim Bohm
Colorado State University

MAP C

- Pure C runs on the MAP
  - Generated code: circuits
    - Basic blocks in outer loops become special purpose hardware "function units"
    - Basic blocks in inner loop bodies are merged and become pipelined circuits
- Sequential semantics obeyed
  - One basic block executed at the time
  - Pipelined inner loops are slowed down to disambiguate read/write conflicts if necessary
  - MAP C compiler identifies (cause of) loop slowdown
- You want more parallelism?
  - Unroll loops
  - Use (OpenMP style) parallel sections
Memory Model

- **Variable = register**
- **Array = memory**
  - **OBM**
    - Declare OB(x) to be array with element type t
      - where t is some 64 bits type
      - SRC6: 6 OBMs
  - **Block RAM**
    - An array lives by default in Block RAM(s)
    - More freedom for element type
      - SRC6: 444 Block RAMs (VP100)
- **1 array access = 1 clock cycle**
  - but compiler can slow down loops for read / write disambiguation
  - loop slowdown: loop runs at >1 clock cycle rate

Programming with loops and arrays

- **Transformational Approach**
  - Start with pure C code
    - Compiler indicates (reasons for) loop slow downs
  - Performance tune (removing inefficiencies)
    - avoid re-reading of data from OBMs using Delay Queues
    - avoid read / write conflicts in same iteration
  - Partition Code and Data
    - distribute data over OBMs and Block RAMs, unroll loops
    - distribute code over two FPGAs, MPI type communication
      - only one chip at the time can access a particular OBM
- **Goal 1**: 1 loop iteration = 1 clock cycle
  - Can very often be achieved
- **Goal 2**: Exploit as much bandwidth as possible
  - Through loop unrolling and task parallelism
  - Balance act: chip area versus bandwidth
How to performance tune: Macros

- C code can be extended using macros allowing for program transformations that cannot be expressed straightforwardly in C
- Macros have semantics unlike C functions:
  - they have a rate and a delay (so they can be integrated in loops)
  - rate (#clocks between inputs)
  - delay (#clocks between in and output)
  - macros can have state (kept between macro calls)
- Two types of macros
  - system provided
  - user provided (Verilog or schematic capture)

Delay Queues for efficient Window Access

- Keep data on chip using Delay Queues
  - Delay queue = system macro implemented with SRL16s (small, fixed size) and BRAMs (large, variable size)
- Simple example:
  - 3x3 window stepping 1 by 1 in column major order
  - image 16 deep

input array traversal
Delay Queues 1

Data access input(i)

two 16-word Delay Queues
shift and store previous columns

Delay Queues 2

Data access input(i)
Delay Queues 3

... Delay Queues 16
...Delay Queues 32

Data access input(i)

...Delay Queues 35

Compute first window

Data access input(i)
Delay Queues: Performance

3x3 window access code
512 x 512 pixel image

<table>
<thead>
<tr>
<th>Routine Style</th>
<th>Number of clocks</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straight Window</td>
<td>2,376,617</td>
<td>close to 9 clocks per iteration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2,340,900: the difference is pipeline prime effect</td>
</tr>
<tr>
<td>Delay Queue</td>
<td>279,999</td>
<td>close to 1 clock per iteration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>262,144: theoretical limit</td>
</tr>
</tbody>
</table>

Streams

- A stream is a flexible communication mechanism
  - Providing parallelism, locality and implicit synchronization
  - Freeing the programmer of timing headaches
  - On the MAP it provides a uniform communication model
    - From µ-proc to MAP (DMA stream)
    - From Global Common Memory to MAP
    - From MAP to MAP (GPIO stream)
    - From FPGA to FPGA (Bridge stream)
    - Inside FPGA between parallel sections

- Stream buffer = FIFO
  - Element can travel through in one clock
  - Processes, FPGAs, MAPs, GCM are all a few clock cycles away from each other
Streams cont’

• **Stream producers and consumers**
  – Live in their own parallel sections
  – Loop based
• **MAP streams are producer driven**
  – Producer can conditionally put elements in the stream
  – Consumer cannot conditionally get
    • but can run at its own speed
  – Conditional gets would cause loop slowdown
• **Two case studies**
  – Dilate Erode (Image Processing kernel)
  – LU Decomposition (Numerical kernel)

---

Dilate / Erode image operators

• Dilate enlarges light objects and shrinks dark objects by taking max pixel value in (masked) window
• Erode enlarges dark objects and shrinks light objects by taking min pixel value in (masked) window
• together they simplify the image, leaving only dominant trends
• Focus Of Attention Codes often use a number of dilates followed by same number of erodes
  – ideal scenario for multiple stream connected loops
  – internally the loops use delay queues for windows
Erode dilate example

\[
D \oplus = \text{MAX(●●●)}
\]
\[
D \square = \text{MAX(●●●)}
\]
\[
E \oplus = \text{MIN(●●●)}
\]
\[
E \square = \text{MIN(●●●)}
\]
D+D□ D+D□ D+D□ D+D□

E+E□

D+D□ D+D□ D+D□ D+D□

E+E□ E+
D+D  D+D  D+D  D+D  D+D
E+E  E+E  E+E  E+E

D+D  D+D  D+D  D+D  D+D  D+D
E+E  E+E  E+E  E+E  E+E
Down Sampling

- **Complication:**
  In FOA the image is often first down-sampled for (sequential) efficiency reasons.

- When the Dilate/Erode pipe is preceded by a down sample, elements are accessed out of (streaming) order. This can be achieved by
  - Storing image in 1 or 4 memories
    - this requires DMA-ing the data in before the computation starts
    - 1 memory: loop slowdown
  or
  - Streaming image through delay queues
    - The computation now runs fully pipelined at stream speed
Erode Dilate Results

- No problem with producer controlled streams
- Pipeline parallelism works great here
  - 8 D/Es (16 sections) run about as fast as 1 D/E
  - the delay is ~2 microseconds per D/E
- 16 operators plus 1 down sample in parallel pipeline
  - each 8 iterations loop-unrolled (it all fits in one FPGA)
  - 8 down samples: 24 maxs
  - 128 dilates/erodes every clock cycle
    - dilate / erode: 4 mins / maxs for cross mask
      - 8 mins / maxs for square mask
  - totalling 792 ops / clock cycle
  - at 100Mhz: 79.2 GigaOpS (ideal)
- No time difference for 2 FPGAs vs 1 FPGA
  - bridge stream just as fast as internal stream
  - But there is no need for the second FPGA (27% LUTs on 1 FPGA)

Downsample, dilate, erode performance on a 640 x 480 pixel image

Down sample: $4 \times 640 \times 480 = 1,228,800$
Dilates Erodes: $16 \times 6 \times 320 \times 240 = 14,645,600$
Total number of ops: $15,974,400$

<table>
<thead>
<tr>
<th>Code versions:</th>
<th>DMA into one OBM</th>
<th>DMA into four OBMs</th>
<th>stream in from GCM</th>
<th>(units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP</td>
<td>782 &gt; 20</td>
<td>343 &gt; 46</td>
<td>230 &gt; 69</td>
<td>(µ-secs) ( GOPS )</td>
</tr>
<tr>
<td>MAP Speedup Vs Pentium</td>
<td>&gt; 46 &gt; 104</td>
<td>&gt; 69 &gt; 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium speed:</td>
<td>36 milliseconds (~ 444 MOPS)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**LU Decomposition**

- You know it:
  
  \[ \text{Cormen et. al. pp 750... } \]

```plaintext
for k = 1 to n {
    for i = k+1 to n
        Aik /= Akk
    for i = k+1 to n
        Aij -= Aik*Akj
}
```

**Pipelining**

- Iteration k+1 can start after iteration k has finished row k+1
  - neither row k nor column k are read or written anymore

- This naturally leads to a streaming algorithm
  - every iteration becomes a process
  - results from iteration/process k flow to process k+1
  - process k uses row k to update sub matrix A[k+1:n, k+1:n]

- Actually, only a fixed number P of processes run in parallel
  - after that, data is re-circulated
Streaming LU

- Pipeline of P processes
  - parallel sections in OpenMP lingo
  - algorithm runs in ceil(N/P) sweeps
- In one sweep P outer k iterations run in parallel
- After each sweep P rows and columns are done
  - Problem size n is reduced to (n-P)^2
  - Rest of the (updated) array is re-circulated
- Each process owns a row
  - Process p owns row p + (i-1)P in sweep i, i=1 to ceil(N/P)
  - Array elements are updated while streaming through the pipe of processes
- Total work O(n^3), speedup ~P
- Forward / backward substitution are simple loops

First, Flip the matrix vertically
Feed matrix to Processor 1

P1

P2
Feed matrix to Processor 2

P2

P3

P2

P3
Process Pipeline

OBMS:
- OBM1 -> OBM2 on even sweeps
- OBM2 -> OBM1 on odd sweeps
- OBM3: siphoning off finished results

Processes are grouped in a parallel sections construct each process is a section

Process 0
- reads either from OBM1 or OBM2
- writes to stream S0

Process i
- reads from Si-1 writes to Si

Process P-1
- reads from Sp-1
- writes finished data to OBM3
- writes rest to OBM1 or OBM0
Inside a process

```c
for(i = (s-1)*P; i < n; i++) {
    for(j = (s-1)*P; j < n; j++) {
        //ping-pong: read from OBM 1 or 2
        if(k&0x1) w = OBM1 [ i * n + j ];
        else w = OBM2 [ i * n + j ];
        // if (i < me) leave data unchanged
        if (i == me) { // store my row
            if (j == i) piv = w;
            myRow[j] = w;
        }
        else
            if (i > me) { // update this row with my row
                // if (j < me) leave data unchanged
                if (j == me) { w /= piv; mul = w; }
                else if (j > me) w -= mul*myRow[j ];
            }
    }
    put_stream( &S0, w);
}
```

LU (plus solvers) Performance

<table>
<thead>
<tr>
<th></th>
<th>n=64</th>
<th>n=128</th>
<th>n=256</th>
<th>n=512</th>
<th>(unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium IV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2.8 GHz )</td>
<td>1205</td>
<td>1469</td>
<td>1109</td>
<td>694</td>
<td>(MFlops)</td>
</tr>
<tr>
<td>MAP (P=12)</td>
<td>492</td>
<td>1104</td>
<td>1718</td>
<td>2087</td>
<td>(MFlops)</td>
</tr>
<tr>
<td>MAP (P=23*)</td>
<td>591</td>
<td>1405</td>
<td>2599</td>
<td>3563</td>
<td>(MFlops)</td>
</tr>
<tr>
<td>MAP speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P=23)</td>
<td>.49</td>
<td>.95</td>
<td>2.3</td>
<td>5.1</td>
<td></td>
</tr>
</tbody>
</table>

*: FPGA1: 10 LU processes + forward and backward solvers
FPGA2: 13 LU processes
Conclusions

- C (and FTN) run on FPGA based HPEC system
  - DEBUG Mode allows most code development and performance tuning on workstation
  - We can apply standard software design
    - stepwise refinement using macros
    - parallel tasks connected by streams (put get: implicit timing)
- Streams allow for massive bandwidth (800MBs/stream) and very short latency (few clock cycles)
  - between on chip processes
  - between processors and processors
  - between processors and memories
- Future work
  - multi-MAP stream and barrier codes