More MaxCompiler Programming

Presenter
MaxAcademy Lecture Series – V1.0, September 2011
Lecture Overview

• Counters / loop iteration variables
• Ways of getting data in and out of the chip
• Stream offsets
• How MaxCompiler maps to hardware
How can we implement this in MaxCompiler?

```java
for (int i = 0; i < N; i++) {
    q[i] = p[i] + i;
}
```

How about this?

```java
DFEVar p = io.input("p", dfeInt(32));
DFEVar i = io.input("i", dfeInt(32));

DFEVar q = p + i;

io.output("q", q, dfeInt(32));
```

**Yes…. But, now we need to create an array i in software and send it to the FPGA as well**
Working with Loop Counters

- There is very little ‘information’ in the input stream.
  - Could compute it directly on the FPGA itself

```c
DFEVar p = io.input("p", dfeInt(32));
DFEVar i = control.count.simpleCounter(32, N);
DFEVar q = p + i;
io.output("q", q, dfeInt(32));
```

Half as many inputs
Less data transfer

- Counters can be used to generate sequences of numbers
- Complex counters can have strides, wrap points, triggers:
  - E.g. if (y==10) y=0; else if (en==1) y=y+2;
Scalar Inputs

• Stream inputs/outputs process arrays
  – Read and write a new value each cycle
  – Off-chip data transfer required: $O(N)$

• Counters can compute intermediate streams on-chip
  – New value every cycle
  – Off-chip data transfer required: None

• Compile time constants can be combined with streams
  – Static value through the whole computation
  – Off-chip data transfer required: None

• What about something that changes occasionally?
  – Don’t want to have to recompile → Scalar input
  – Off-chip data transfer required: $O(1)$
Scalar Inputs

• Consider:

```c
void fn1(int N, int *q, int *p) {
    for (int i = 0; i < N; i++)
        q[i] = p[i] + 4;
}
```

```c
void fn2(int N, int *q, int *p, int C) {
    for (int i = 0; i < N; i++)
        q[i] = p[i] + C;
}
```

• In fn2, we can change the value of C without recompiling, but it is constant for the whole loop

• MaxCompiler equivalent:

```c
DFEVar p = io.input("p", dfeInt(32));
DFEVar C = io.scalarInput("C", dfeInt(32));
DFEVar q = p + C;
io.output("q", q, dfeInt(32));
```

A scalar input can be changed once per stream, loaded into the chip before computation starts.

Written by host
Common uses for Scalar Inputs

• Things that do not change every cycle, but do change sometimes and we do not want to rebuild the .max file.

• Constants in expressions

• Flags to switch between two behaviours
  – result = enabled ? x+7 : x;

• Control parameters to counters, e.g. max, stride, etc
  – if (cnt==cnt_max) cnt=0; else cnt = cnt + cnt_step;
On-chip memories / tables

• An FPGA has a few MB of very fast block RAM
• Can be used to explicitly store data on chip:
  – Lookup tables
  – Temporary Buffers
• *Mapped* ROMs/RAMs can also be accessed by host

```c
DFEVar p = io.input("p", dfeInt(10));

DFEVar q = mem.romMapped("table", p,
                        dfeInt(32), 1024);

io.output("q", q, dfeInt(32));
```
Getting data in and out of the chip

- In general we have streams, ROMs (tables) and scalars
- Use the most appropriate mechanism for the type of data and required host access speed.
- Stream inputs/outputs can operate for a subset of cycles using a control signal to turn them on/off

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (items)</th>
<th>Host write speed</th>
<th>Chip area cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar input/output</td>
<td>1</td>
<td>Slow</td>
<td>Low</td>
</tr>
<tr>
<td>Mapped memory (ROM / RAM)</td>
<td>Up to a few thousand</td>
<td>Slow</td>
<td>Moderate</td>
</tr>
<tr>
<td>Stream input/output</td>
<td>Thousands to billions</td>
<td>Fast</td>
<td>Highest</td>
</tr>
</tbody>
</table>
Stream Offsets

• So far, we’ve only performed operations on each individual point of a stream
  – The stream size doesn’t actually matter (functionally)!
  – At each point computation is independent

• Real world computations often need to access values from more than one position in a stream
  – For example, a 3-pt moving average filter:

\[ y_i = (x_{i-1} + x_i + x_{i+1}) / 3 \]
Stream Offsets

- *Stream offsets* allow us to compute on values in a stream other than the current value.
- Offsets are relative to the *current position* in a stream; *not* the start of the stream.
- Stream data will be buffered on-chip in order to be available when needed → uses BRAM
  - Maximum supported offset size depends on the amount of on-chip BRAM available. Typically 10s of thousands of points.
class MovingAverageSimpleKernel extends Kernel {
    
    MovingAverageSimpleKernel(KernelParameters parameters) {
        super(parameters);
        
        DFEVar x = io.input("x", dfeFloat(8, 24));
        DFEVar prev = stream.offset(x, -1);
        DFEVar next = stream.offset(x, 1);
        DFEVar sum = prev + x + next;
        DFEVar result = sum / 3;
        io.output("y", result, dfeFloat(8, 24));
    }
}
Kernel Execution
Kernel Execution
Kernel Execution
Kernel Execution
Kernel Execution
Boundary Cases

What about the boundary cases?
More Complex Moving Average

- To handle the boundary cases, we must explicitly code special cases at each boundary

\[
y_i = \begin{cases} 
(x_i + x_{i+1})/2 & \text{if } i = 0 \\
(x_{i-1} + x_i)/2 & \text{if } i = N - 1 \\
(x_{i-1} + x_i + x_{i+1})/3 & \text{otherwise}
\end{cases}
\]
Kernel Handling Boundary Cases

```java
class MovingAverageKernel extends Kernel {
  MovingAverageKernel(KernelParameters parameters) {
    super(parameters);
    // Input
    DFEVar x = io.input("x", dfeFloat(8, 24));
    DFEVar size = io.scalarInput("size", dfeUInt(32));
    // Data
    DFEVar prevOriginal = stream.offset(x, -1);
    DFEVar nextOriginal = stream.offset(x, 1);
    // Control
    DFEVar count = control.count.simpleCounter(32, size);
    DFEVar aboveLowerBound = count > 0;
    DFEVar belowUpperBound = count < size - 1;
    DFEVar withinBounds = aboveLowerBound & belowUpperBound;
    DFEVar prev = aboveLowerBound ? prevOriginal : 0;
    DFEVar next = belowUpperBound ? nextOriginal : 0;
    DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;
    DFEVar sum = prev + x + next;
    DFEVar result = sum / divisor;
    io.output("y", result, dfeFloat(8, 24));
  }
}
```
Multidimensional Offsets

• Streams are one-dimensional but can be interpreted as multi-dimensional structures
  – Just like arrays in CPU memory

• A multidimensional offset, is the distance between the points in the one dimensional stream ➔ linearise

```c
for (int y = 0; y < N; y++)
for (int x = 0; x < N; x++)
    p[y][x] = q[y-1][x] + q[y][x-1] + q[y][x] + q[y][x+1] + q[y+1][x]
```

And of course we now need to handle boundaries in both dimensions...
MaxCompiler generates VHDL ready for FPGA vendor tools

Synthesis transforms VHDL into logical “netlist” – sets of basic logic expressions

Map fits basic logic into N-input look-up tables

Place puts LUTs, DSPs, RAMs etc at specific locations on chip

Route sets up wiring between blocks
How it maps to hardware

Tue 15:00: MaxCompiler version: 2010.1
Tue 15:00: Build "MovingAverage" start time: Tue Feb 16 15:00:27 GMT 2010
Tue 15:00: Instantiating manager
Tue 15:00: Instantiating kernel "MovingAverageKernel"
Tue 15:00: Compiling manager (PCIe Only)
Tue 15:00: Compiling kernel "MovingAverageKernel"
Tue 15:00: Generating hardware for kernel "MovingAverageKernel"
Tue 15:00: Generating VHDL + netlists (including running CoreGen)
Tue 15:00: Running back-end hardware build (10 build phases)

Tue 15:00: (1/10) - GenerateMaxFileDataFile
Tue 15:00: (2/10) - XST
Tue 15:02: (3/10) - NGCBuild
Tue 15:02: (4/10) - ResourceCounter
Tue 15:03: (5/10) - NGDBuild
Tue 15:03: (6/10) - MPPR
Tue 15:19: (7/10) - GenerateMaxFile
Tue 15:21: (8/10) - XDLBuild
Tue 15:22: (9/10) - ResourceUsageBuild
Tue 15:22: (10/10) - ResourceAnnotationBuildPass

Tue 15:22: FINAL RESOURCE USAGE
Tue 15:22: LUTs: 9154 / 149760 (6.11%)
Tue 15:22: FFs: 10736 / 149760 (7.17%)
Tue 15:22: BRAMs: 21 / 516 (4.07%)
Tue 15:22: DSPs: 0 / 1056 (0.00%)

Tue 15:22: MAX file: /oliver/builds/MovingAverage/results/MovingAverage.max
On-chip Resources

- Different operations use different resources
- Main resources
  - LUTs
  - Flip-flops
  - DSP blocks (25x18 multipliers)
  - Block RAM (36Kbit)
  - Routing!

![Diagram of on-chip resources](image)
Resource Usage Reporting

• Allows you to see what lines of code are using what resources and focus optimization
  - Separate reports for each kernel and for the manager

<table>
<thead>
<tr>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Resources used by this file</th>
</tr>
</thead>
<tbody>
<tr>
<td>727</td>
<td>871</td>
<td>1.0</td>
<td>2</td>
<td>0.24% 0.15% 0.09% 0.10%</td>
</tr>
<tr>
<td>0.24%</td>
<td>0.15%</td>
<td>0.09%</td>
<td>0.10%</td>
<td>% of available</td>
</tr>
<tr>
<td>71.41%</td>
<td>61.82%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>% of total used</td>
</tr>
<tr>
<td>94.29%</td>
<td>97.21%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>% of user resources</td>
</tr>
</tbody>
</table>

```java
public class MyKernel extends Kernel {
    public MyKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar p = io.input("p", dfeFloat(8, 24));
        DFEVar q = io.input("q", dfeUInt(8));
        DFEVar offset = io.scalarInput("offset", dfeUInt(8));
        DFEVar addr = offset + q;
        DFEVar v = mem.romMapped("table", addr,
                                   dfeFloat(8, 24), 256);

        p = p * p;
        p = p + v;
        io.output("r", p, dfeFloat(8, 24));
    }
}
```
Exercises

1. Write a MaxCompiler kernel program (using \textit{dfeFloat}\textsubscript{8,24}) that computes:

\[
q_i = \begin{cases} 
  p_i + c & \text{if } p_i > k \\
  p_i \times c & \text{otherwise}
\end{cases}
\]

Draw the kernel dataflow graph.

2. Write a MaxCompiler kernel program that computes a 3x3 2D moving average on a single input stream of 1024x1024 values, ignoring any boundary conditions.

3. Extend the 3x3 moving average kernel from (2) to support a variable problem size at run-time, from 256\textsuperscript{2} to 2048\textsuperscript{2}. Hint: you will need to use \textit{variable stream offsets} (see MaxCompiler Kernel Compiler tutorial for more information).