CO405H
Computing in Space with OpenSPL
Topic 8: Programming DFE Loops

Oskar Mencer
Georgi Gaydadjiev

Department of Computing
Imperial College London

http://www.doc.ic.ac.uk/~oskar/
http://www.doc.ic.ac.uk/~georgig/

CO405H course page: http://cc.doc.ic.ac.uk/openspl15/
WebIDE: http://openspl.doc.ic.ac.uk
OpenSPL consortium page: http://www.openspl.org

o.mencer@imperial.ac.uk g.gaydadjiev@imperial.ac.uk
Overview

- Loops and Transformations
- Variable length loops
- Dealing with cycles
Overview of Accelerating Loops

• **Classifying Loops**
  – Attributes and measures

• **Simple Fixed Length Stream Loops**
  – Example vector add
  – Custom memory controllers

• **Nested Loops**
  – Counter chains
  – Streaming and unrolling
  – How to avoid cyclic graphs

• **Variable Length Loops**
  – Convert to fixed length

• **Loops with dependence cycles**
  – How to build cyclic data-flow graphs
  – How to exploit pipelining in cyclic graphs

**Objective:** understand how to get from source-code loops to MaxCompiler configurations

**That go fast**
Classifying Loops

Loop Attributes:
• Array access
  – pattern is important (stride)
• Iteration: loop-carried dependency
  – distance is important
  – fixed distance (good) or variable distance (bad)

Loop performance metrics:
• Computation to memory-access ratio
• Working set size => custom memory hierarchy
• Loop bottlenecks: CPU or Memory or IO

```java
for (i=0; i<N; ++i) {
    A[i] = ...B[i]...
    ...B[i*M]...
    ...B[C[i]]...
}
```

```java
for (i=0; i<N; ++i) {
    A[i] = ...A[i-1]...
    ...A[i-100]...
    ...A[i-k]...
}
```
Overview of Accelerating Loops

• Classifying Loops
  – Attributes and measures

• **Simple Fixed Length Stream Loops**
  – Example vector add
  – Custom memory controllers

• Nested Loops
  – Counter chains
  – Streaming and unrolling
  – How to avoid cyclic graphs

• Variable Length Loops
  – Convert to fixed length

• Loops with dependence cycles
  – How to build cyclic data-flow graphs
  – How to exploit pipelining in cyclic graphs
The Stream Loop

uint A[...]; uint B[...];
for (int count=0; ; count += 1)

DFEVar A = io.input("input", dfeUInt(32));
DFEVar B = A + 1;
io.output("output", B, dfeUInt(32));
Adding a Loop Counter

```c
for (int count=0; ; count += 1)
```

DFEVar A = io.input("input", dfeUInt(32));
DFEVar count = control.count.simpleCounter(32);
DFEVar B = A + count;
io.output("output", B, dfeUInt(32));
Adding a Loop Counter

for (int count=0; ; count += 1)

DFEVar A = io.input(“input”, dfeUInt(32));
DFEVar count =
control.count.simpleCounter(32);
DFEVar B = A + count;
io.output(“output”, B, dfeUInt(32));

If the array subscripts are more complicated, you need to think about how to generate addresses for the DRAM.
Overview of Accelerating Loops

• **Classifying Loops**
  – Attributes and measures

• **Simple Fixed Length Stream Loops**
  – Example vector add
  – Custom memory controllers

• **Nested Loops**
  – Counter chains
  – Streaming and unrolling
  – How to avoid cyclic graphs

• **Variable Length Loops**
  – Convert to fixed length

• **Loops with dependence cycles**
  – How to build cyclic data-flow graphs
  – How to exploit pipelining in cyclic graphs
int count = 0;
for (int i=0; i<N; ++i) {
    for (int j=0; j<M; ++j) {
        count += 1;
    }
}

DFEVar A = io.input(“input”, dfeUInt(32));
CounterChain chain = control.count.makeCounterChain();
DFEVar i = chain.addCounter(N, 1).cast(dfeUInt(32));
DFEVar j = chain.addCounter(M, 1).cast(dfeUInt(32));
DFEVar B = A + i*100 + j;
io.output(”output”, B, dfeUInt(32));
Loop Unrolling with Dependence

for (i = 0; ; i += 1) {
    float d = input[i];
    float v = 2.91 – 2.0*d;
    for (iter=0; iter < 4; iter += 1)
        v = v * (2.0 - d * v);
    output[i] = v;
}

DFEVar d = io.input(”d”, dfeFloat(8, 24));
DFEVar TWO= constant.var(dfeFloat(8,24), 2.0);
DFEVar v = constant.var(dfeFloat(8,24), 2.91) – TWO*d;

for ( int iteration = 0; iteration < 4; iteration += 1) {
    v = v*(TWO– d*v);
}
io.output(”output” , v, dfeFloat(8, 24));
for (i = 0; ; i += 1) {
    float d = input[count];
    float v = 2.91 - 2.0*d;
    for (iter=0; iter < 4; iter += 1)
        v = v * (2.0 - d * v);
    output[i] = v;
}

DFEVar d = io.input("d", dfeFloat(8, 24));
DFEVar TWO= constant.var(dfeFloat(8,24), 2.0);
DFEVar v = constant.var(dfeFloat(8,24), 2.91) – TWO*d;

for ( int iteration = 0; iteration < 4; iteration += 1) {
    v = v*TWO– d*v;
}

io.output("output", v, dfeFloat(8, 24));

• The software loop has a cyclic dependence (v)
• But the unrolled datapath is acyclic
Overview of Accelerating Loops

• Classifying Loops
  – Attributes and measures

• Simple Fixed Length Stream Loops
  – Example vector add
  – Custom memory controllers

• Nested Loops
  – Counter chains
  – Streaming and unrolling
  – How to avoid cyclic graphs

• Variable Length Loops
  – Convert to fixed length

• Loops with dependence cycles
  – How to build cyclic data-flow graphs
  – How to exploit pipelining in cyclic graphs
Variable Length Loop

```c
for (count=0; ; count += 1) {
    int d = input[count];
    int shift = 0;
    while (d != 0 && ((d & 0x3FF) != 0x291)) {
        shift = shift + 1;
        d = d >> 1;
    }
    output[count] = shift;
}
```

• What do we do with a while loop (or a loop with a “break”)?

```c
// converted to fixed length
for (count=0; ; count += 1) {
    int d = input[count];
    int shift = 0;
    bool finished = false;
    for (int i = 0; i < 22; ++i) {
        bool condition = (d != 0 && ((d & 0x3FF) != 0x291));
        finished = condition ? true : finished; // loop-carried
        shift = finished ? shift : shift + 1; // dependencies
        d = d >> 1;
    }
    output[count] = shift;
}
```

• Find maximum number of iterations
• *Predicate* execution of loop body
• Using a bool that is set to false when the while loop condition fails
for (count=0; ; count += 1) {
    int d = input[count];
    int shift = 0;
    bool finished = false;
    for (int i = 0; i < 22; ++i) {
        bool condition=(d!=0&&((d&0x3FF)!=0x291));
        finished = condition ? true : finished;
        shift = finished ? shift + 1;
        d = d >> 1;
    }
    output[count] = shift;
}

DFEVar d = io.input(“d”, dfeUInt(32));
DFEVar shift = constant.var(dfeUInt(5), 0);
DFEVar finished = constant.var(dfeBool(), 0);
for ( int i = 0; i < 22; ++i) { // unrolled
    DFEVar condition = d.neq(0)&((d&0x3FF).neq(0x291));
    finished = condition ? constant.var(1) : finished;
    shift = finished ? shift + constant.var(1);
    d = d >> 1;
}

// Output
io.output(“output”, shift, dfeUInt(5));
Variable Length Loop – in hardware

for (count=0; ; count += 1) {
    int d = input[count];
    int shift = 0;
    bool finished = false;
    for (int i = 0; i < 22; ++i) {
        bool condition=(d!=0&&((d&0x3FF)!=0x291));
        finished = condition ? true : finished;
        shift = finished ? shift : shift + 1;
        d = d >> 1;
    }
    output[count] = shift;
}

DFEVar d = io.input("d", dfeUInt(32));
DFEVar shift = constant.var(dfeUInt(5), 0);
DFEVar finished = constant.var(dfeBool(), 0);
for ( int i = 0; i < 22; ++i) { // unrolled
    DFEVar condition = d.neq(0)&&((d&0x3FF).neq(0x291));
    finished = condition ? constant.var(1) : finished;
    shift = finished ? shift : shift + constant.var(1);
    d = d >> 1;
}

// Output
io.output("output", shift, dfeUInt(5));

• Again, the unrolled loop is acyclic
To Unroll or Not to Unroll

• **Loop Unrolling**
  – Gets rid of loop-carried dependency by creating a long pipeline
  – Requires O(N) space on the chip...what if it does not fit?
    – If we can’t unroll, we end up with a cycle in the dataflow graph
    – As we will see, we need to take care to make sure the cycle is compatible with the pipeline depth

• **Variable-length loop (with loop-carried dependency)**
  – Can be fully unrolled, BUT need to know maximal number of iterations
  – Utilization depends on actual data...
  – What if max iterations is much larger than average? Or max is not known? Or max iterations don’t fit on the chip?
Overview of Accelerating Loops

• Classifying Loops
  – Attributes and measures

• Simple Fixed Length Stream Loops
  – Example vector add
  – Custom memory controllers

• Nested Loops
  – Counter chains
  – Streaming and unrolling
  – How to avoid cyclic graphs

• Variable Length Loops
  – Convert to fixed length

• Loops with dependence cycles (next lecture)
  – How to build cyclic data-flow graphs
  – How to exploit pipelining in cyclic graphs
Summary

• Loops are where parallelism comes from
• The loop control variables are generated using counters
• In a nested loop this needs a counter chain
• Inner loops with small bounds can be unrolled
• Loops with dependences lead to acyclic graphs when unrolled
• Loops with variable/unknown bounds (while loops) need to be converted into for-loops with predication