CO405H

Computing in Space with OpenSPL
Topic 9: Programming DFEs (Loops II)

Oskar Mencer **Georgi Gaydadjiev**
special thanks: Jacob Bower

Department of Computing
Imperial College London

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

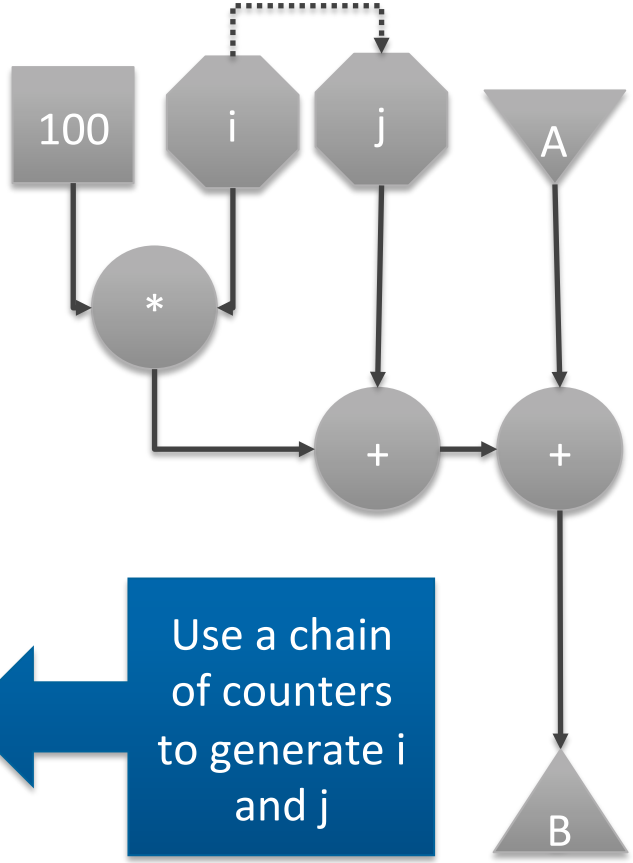
CO405H course page: http://cc.doc.ic.ac.uk/openspl14/
WebIDE: http://openspl.doc.ic.ac.uk
OpenSPL consortium page: http://www.openspl.org

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

From Loops I: Loop Nest without Dependence

```
int count = 0;
for (int i=0; i<N; ++i) {
   for (int j=0; j<M; ++j) {
        B[count] = A[count]+(i*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));
```

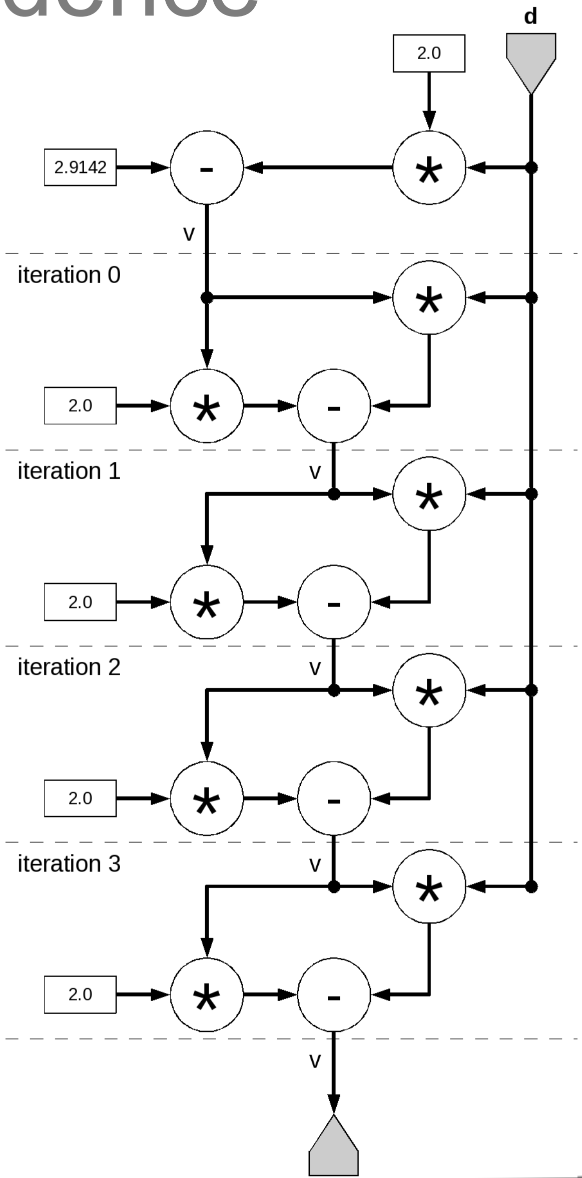


Use a chain of counters to generate i and j

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));
```



Overview of Spatializing Loops

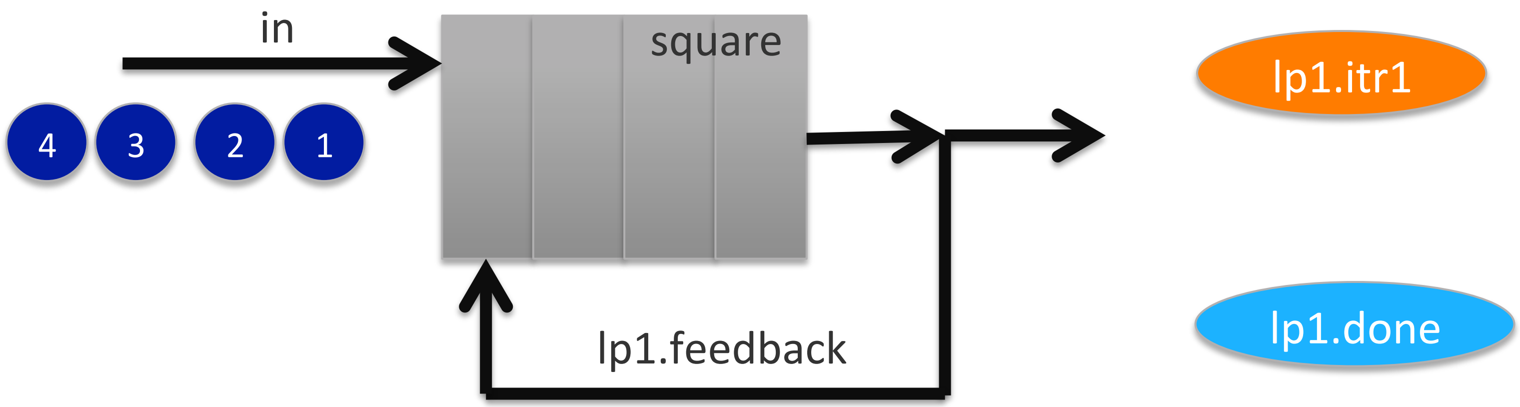
- Classifying Loops
  - Attributes and measures
- Simple Fixed Length Stream Loops
  - Example vector add
  - Custom memory controllers
- Nested Loops
  - Counter chains
  - Streaming and unrolling
  - How to avoid cyclic graphs
- Variable Length Loops
  - Convert to fixed length
- Loops with data dependencies
  - DFESeqLoop: with a data parallel streaming loop
  - DFEParLoop: with a sequential streaming loop

DFESeqLoop: Data Parallel Streaming Loop

Instead of unrolling, create a sequential inner loop to save space => resource sharing

Assume 4 stages implement 1 multiply *

<Click to advance>

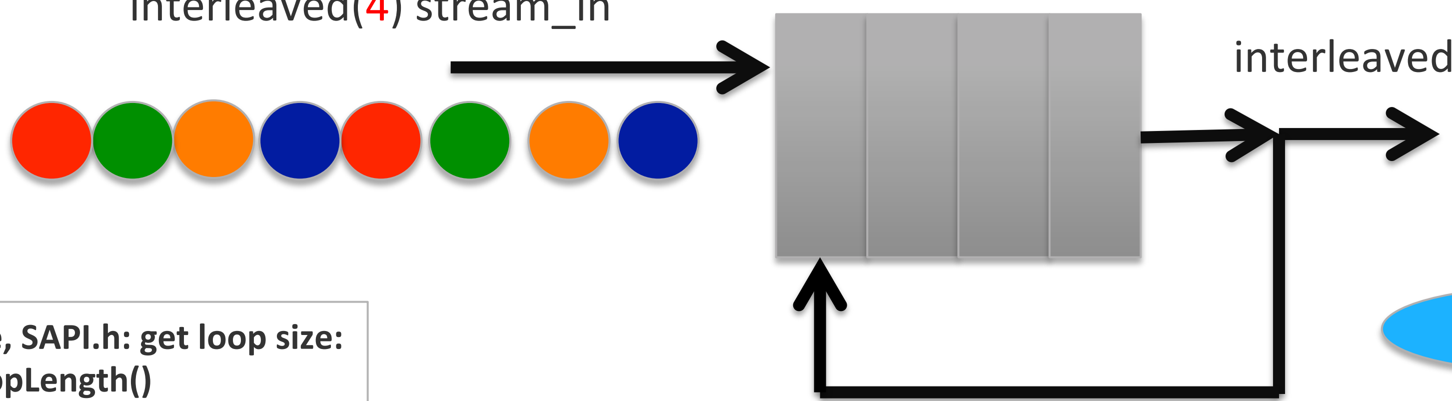



```
for i=1 to N      // data parallel loop
  in = stream_in[i];
  for j=1 to 2 { // sequential loop lp1
      square = in * in;
      in = square;  //  feedback
  }
stream_out[i] = square;
```

```
DFESeqLoop lp1= new DFESeqLoop(this, “lp1”, 2);
DFEVar in = io.input("in", dfeFloat(8, 24), lp1.itr1);
lp1.set_input(in);
  DFEVar square = lp1.feedback * lp1.feedback;
lp1.set_output(square);
io.output(”square", square, dfeFloat(8, 24), lp1.done);
```

DFEParLoop: Sequential Streaming Loop

Simple Accumulator Example (actually 4 concurrent accumulators)



CPU code, SAPI.h: get loop size:
mget_loopLength()
returns 4

```
for j=1 to 4: out[j]=0.0;
for i=1 to N:    // sequential loop
  for j=1 to 4: // data parallel loop
       out[j] = out[j]+stream_in[i];
```

```
DFEParLoop lp2 = new DFEParLoop(this, “lp2”);
  DFEVar in = io.input("in", dfeFloat(8, 24), lp2.ndone);
  lp2.set_input(dfeFloat(8,24), 0.0);
    DFEVar result = in + lp2.feedback;
  lp2.set_output(result);
  io.output(”result", result, dfeFloat(8, 24), lp2.done);
```

Of course j could be a lot larger, but we do
4 at a time here since we assume 4 stages in a **+**

DFESeqLoop details (optional)

```
class DFESeqLoop extends KernelLib {
    DFEVar itr1, done, feedback, feed_in;
    OffsetExpr loop;

    DFESeqLoop(Kernel owner, String loop_name, int loop_itrs) {
      super(owner);
      loop = stream.makeOffsetAutoLoop(loop_name);
      DFEVar pipe_len = loop.getDFEVar(this, dfeUInt(32));
      DFEVar global_pos = control.count.simpleCounter(32);
      itr1 = global_pos < pipe_len;
      done = global_pos >= (pipe_len * loop_itrs);
    }

    void set_input(DFEVar loop_in) {
       feed_in = loop_in.getType().newInstance(this);
       feedback = itr1 ? feed_in : loop_in; // feed_in in the first iteration
    }

    void set_output(DFEVar result) {
       feed_in <== stream.offset(result, -loop); // connect the loop
    }
}
```

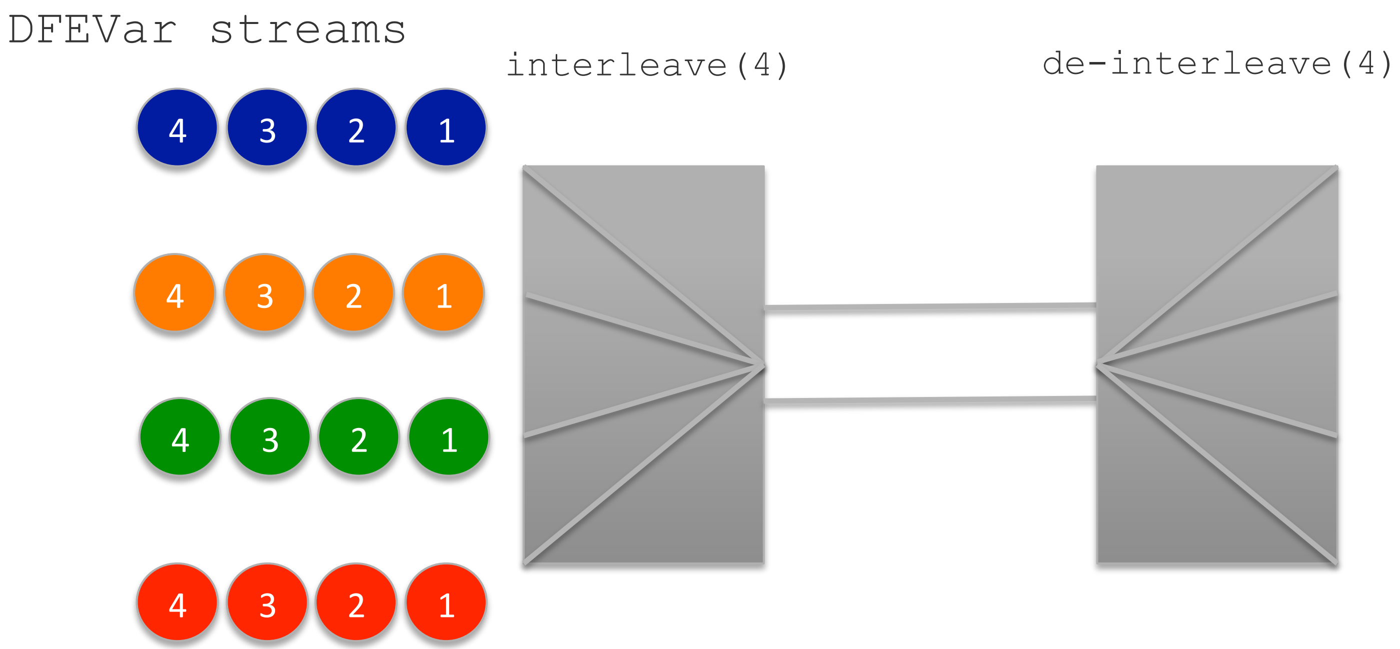
```
DFESeqLoop lp1= new DFESeqLoop(this, “lp1”, 2);
DFEVar in = io.input("in", dfeFloat(8, 24), lp1.itr1);
lp1.set_input(in);
DFEVar square = lp1.feedback * lp1.feedback;
lp1.set_output(square);
io.output(”square", square, dfeFloat(8, 24), lp1.done);
```

DFEParLoop details (optional)

```
class DFEParLoop extends KernelLib {
    DFEVar feed_in, pipe_len, global_pos, feedback, done, ndone;
    OffsetExpr loop;

    DFEParLoop(Kernel owner, String loop_name) {
      super(owner);
      loop = stream.makeOffsetAutoLoop(loop_name);
      pipe_len = loop.getDFEVar(this,dfeUInt(32));
      // ParLoop iterates as long as there is data
      DFEVar stream_len = io.scalarInput(loop_name + "_len", dfeUInt(32));
      global_pos = control.count.simpleCounter(32);
      done = global_pos >= (stream_len + pipe_len);
      ndone = global_pos < stream_len;
    }

    void set_input(DFEType fb_type, double init) {
       feed_in = fb_type.newInstance(this);
       DFEVar start_feedback = global_pos < pipe_len;
       feedback = start_feedback ? feed_in : init;
    }

    void set_output(DFEVar result) {
       feed_in <== stream.offset(result, -loop); // connect the loop
    }
```

```
DFEParLoop lp2 = new DFEParLoop(this, “lp2”);
DFEVar in = io.input("in", dfeFloat(8, 24), lp2.ndone);
lp2.set_input(dfeFloat(8,24), 0.0);
DFEVar result = lp2.feedback + in;
lp2.set_output(result);
io.output(”result", result, dfeFloat(8, 24), lp2.done);
```

Data Loops need Cyclic (Round Robin) interleaved Data



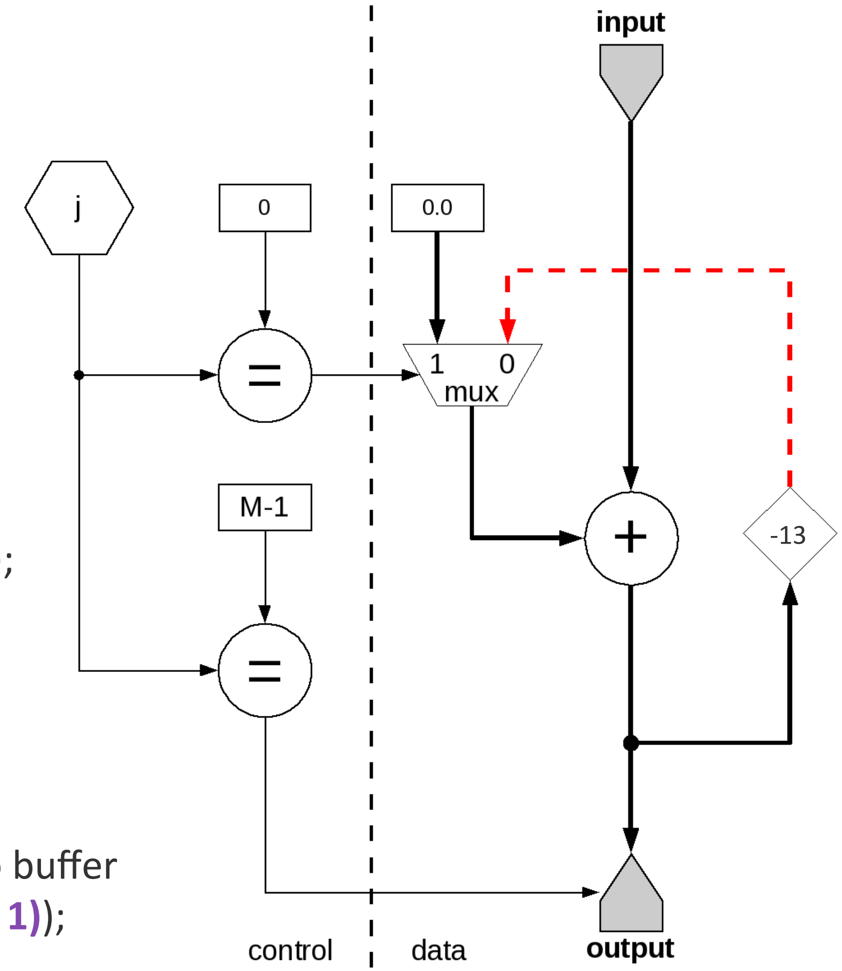
Conversion can be done at runtime on the CPU in Software or on the DFE as Dataflow
OR interleaving and de-interleaving can be pre-computed and stored in memory

Multiple Loops: n-Body problem

```
[…]
// all the below are interleaved data streams
DFEVar rx = pjX - piX;
DFEVar ry = pjY - piY;
DFEVar rz = pjZ - piZ;
DFEVar dd = rx*rx + ry*ry + rz*rz + scalars.EPS;
DFEVar d = 1 / (dd * KernelMath.sqrt(dd));
DFEVar s = pjM * d;

DFEParLoop lp = new DFEParLoop(this, “lp”);
lp.set_inputs(3, dfeFloat(8,24), 0.0);
DFEVar accX = lp.feedback[0] + rx*s;
DFEVar accY = lp.feedback[1] + ry*s;
DFEVar accZ = lp.feedback[2] + rz*s;

lp.set_outputs(accX, accY, accZ);
[…]
```

Ok, so how does this really work…

```
int count = 0;
for (int i=0; ; i += 1)  {
  sum[i] = 0.0;
  for (int j=0; j<M; j += 1) {
        sum[i] = sum[i] + input[count];
        count += 1;
  }
  output[i] = sum[i];
}
```




```
DFEVar LoopCount = control.count.simpleCounter(32, M);
DFEVar carry = scalarType.newInstance(this);

DFEVar sum = LoopCount.eq(0) ? 0.0 : carry;
sum = input + sum;

carry.connect(stream.offset(sum, −13));  // feedback fifo buffer
io.output(”output” , sum, scalarType, LoopCount.eq(M − 1));
```

Pipeline Depth, Why -13

The multiplexer has a pipeline depth of 1

The floating-point adder has
a pipeline depth of 12

Total loop latency = 13 ticks,

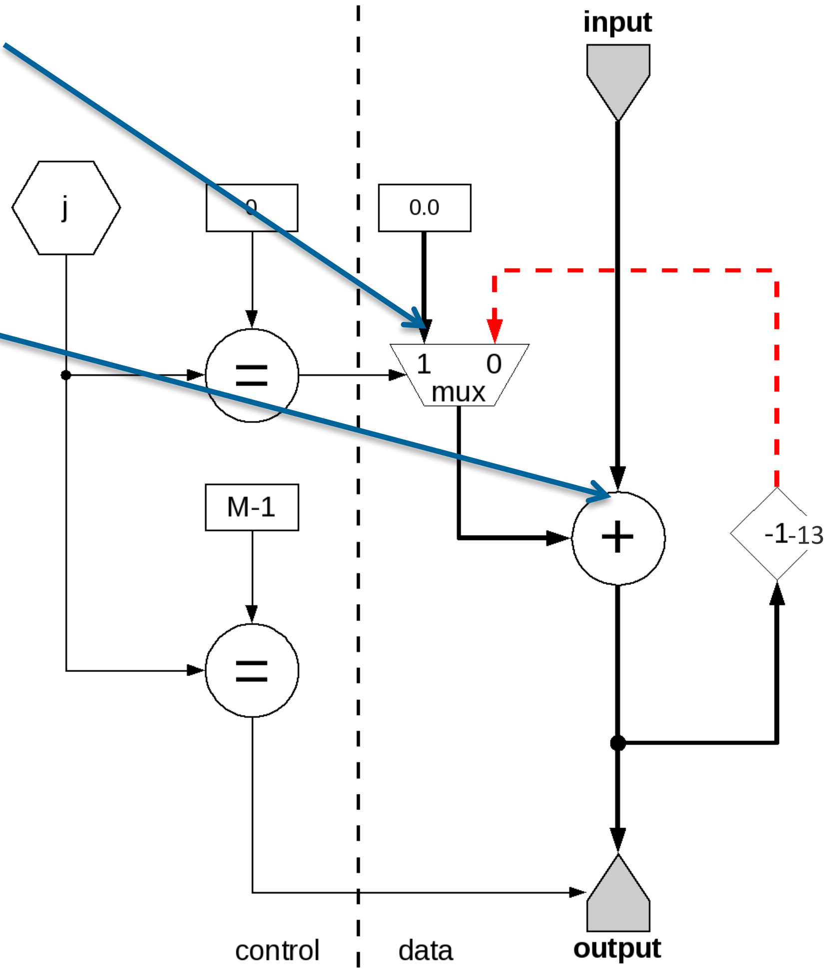
```
carry.connect(stream.offset(sum,-13));
```

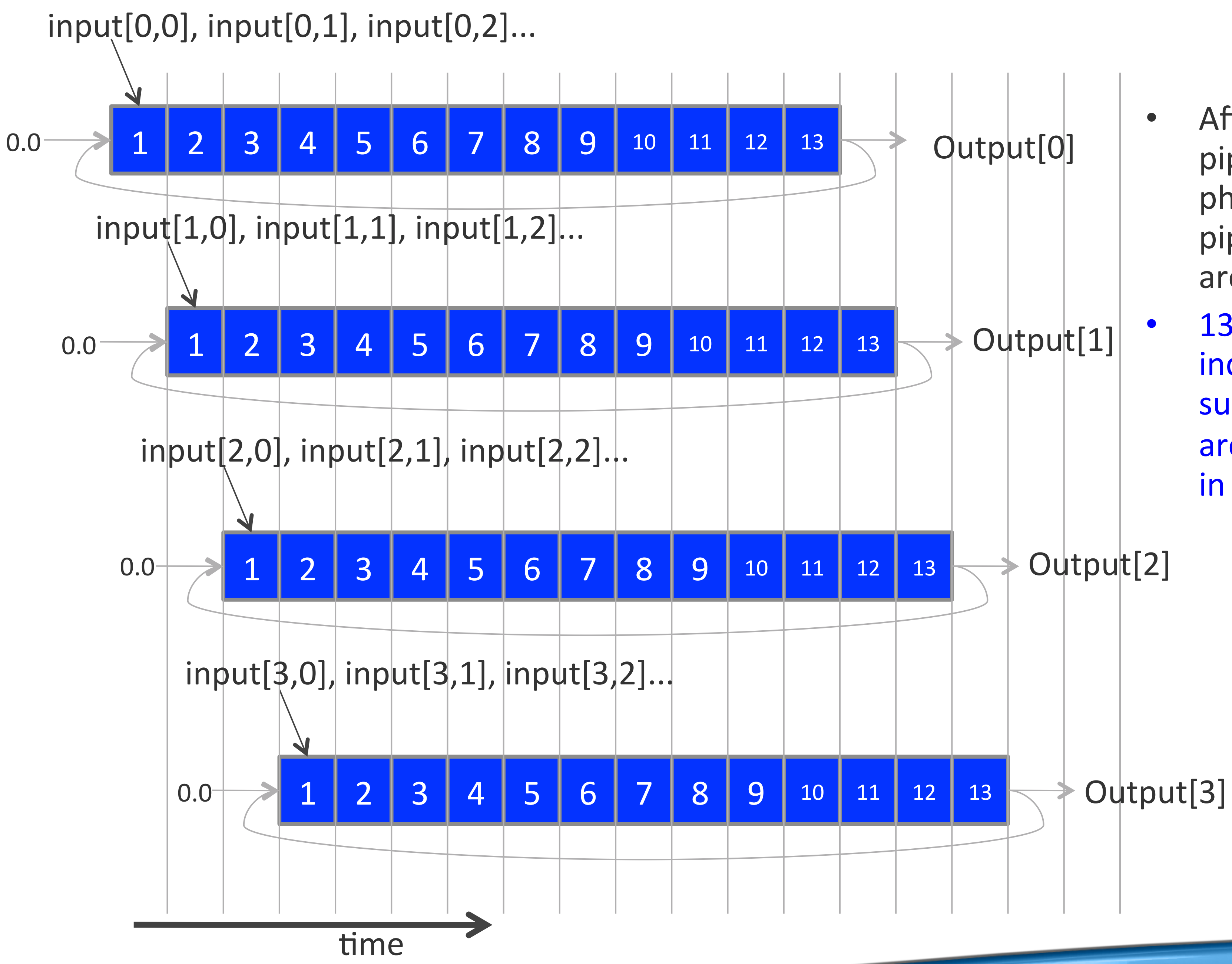
luckily `stream.makeOffsetAutoLoop()`
figures out the loop length for us.

Now on the software side we need to interleave
the input stream with a stride of 13.

CPU call: `get_loopLength()`
Generated by the compiler
for every loop in the Kernel
will return 13!

See SAPI.h interface…

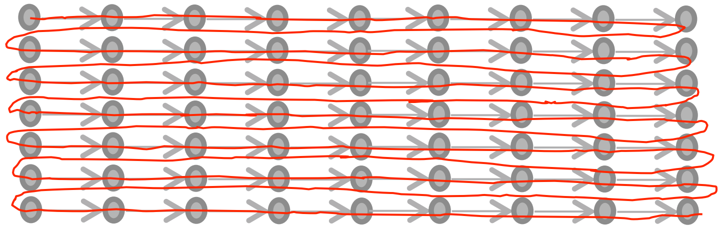




- After an initial pipeline fill phase, all 13 pipeline stages are occupied
- 13 independent summations are computed in parallel

Pipeline Depth and Loop-Carry Distance

```
for (i=0;i<N;i++){
   for (j=0;j<M;j++)
        v[i]=v[i]*v[i]+1; // distance 1
}
```

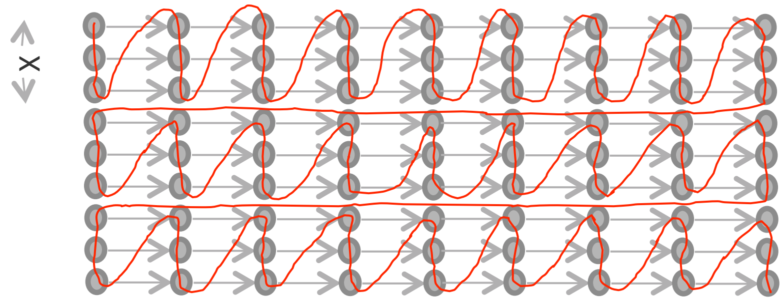


Now the j-loop has a loop-carried dependency with distance 1, i.e. each loop needs the v[i] result of the previous loop, BUT the v[i]*v[i]+1 operations have X stages and thus take X clock cycles.

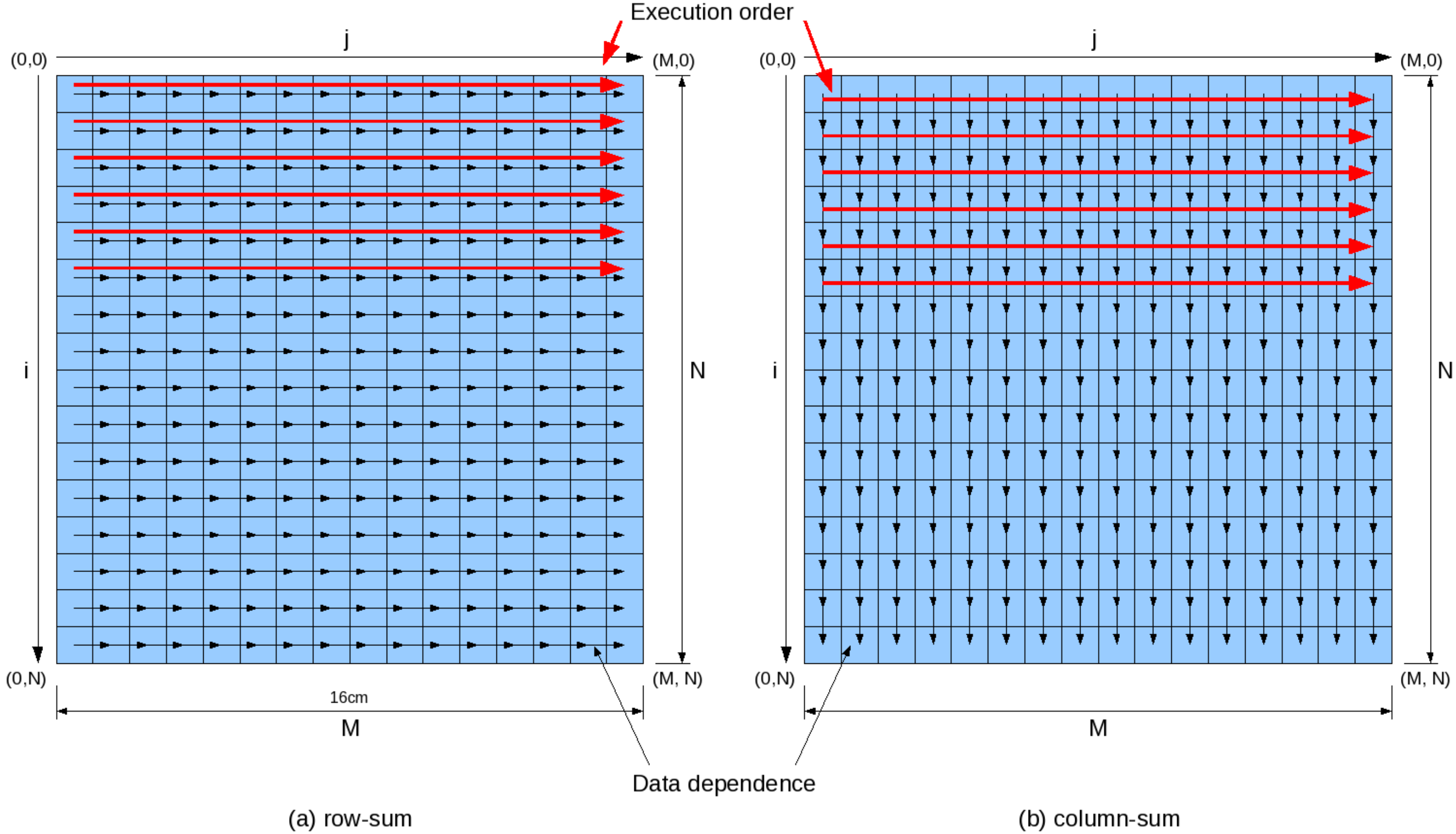
Note that v[i]s are independent, i.e. the i-loop has no dependency!

⇒ We thus need X activities (v[i]s) to be in the loop at all times to fully utilize all stages of the multiplication pipeline.

```
for (i=0;i<N/X;i++){
  for (j=0;j<M;j++)
    for (k=0;k<X;k++) // distance X
      v[i*X+k]=v[i*X+k]*v[i*X+k]+1;
}
```



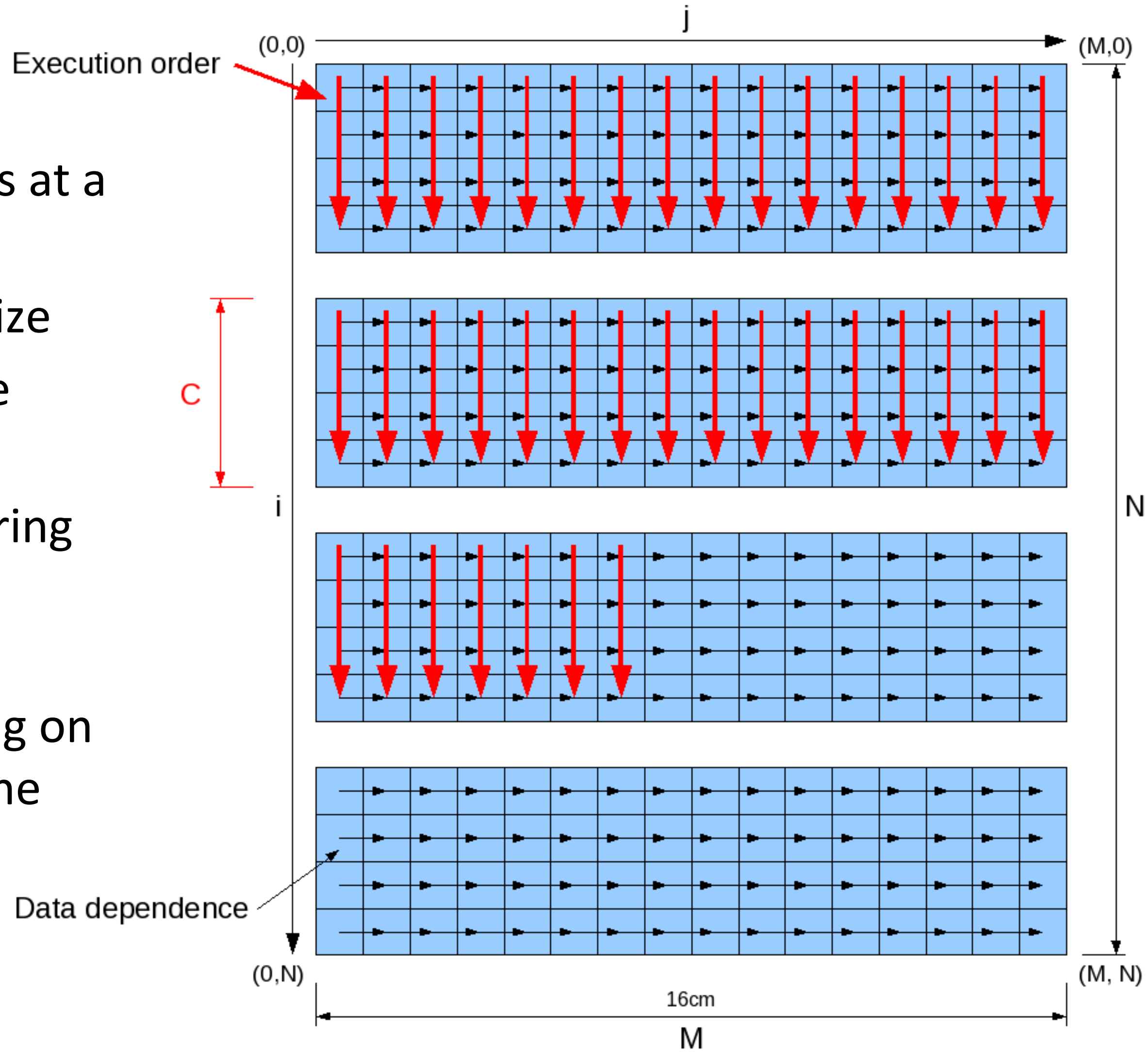
Computing with 2D Arrays: Loop interchange



- **Example**: Row-wise summation is serial due to chain of dependence
- Column-wise summation would be easy of course
- So we can keep the pipeline in a cyclic data datapath full by flipping the problem – ie by interchanging the loops

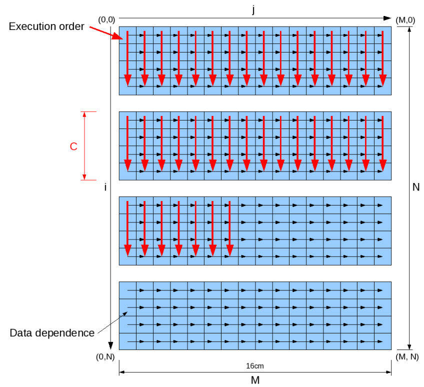
Loop Tiling reduces FMEM requirement

- Idea: sum a block of rows at a time (“tiling”)
- We can choose the tile size
- Just big enough to fill the pipeline
- so no unnecessary buffering is needed
- c is the length of the feedback loop, depending on the number format for the accumulator!



Loop Tiling reduces FMEM requirement

What if we need a particular loop length because of the particular size of our matrix?



We can set loopLength to any number larger than the minimum LoopLength:

```
DFEParLoop lp2 = new DFEParLoop(this, "lp2");
   DFEVar in = io.input("in", dfeFloat(8, 24), lp2.ndone);
   lp2.set_input(dfeFloat(8,24), 0.0);
      DFEVar result = lp2.feedback + in;
   lp2.set_output(result, 16); // set loopLength to 16
   io.output("result", result, dfeFloat(8, 24), lp2.done);
```

However, the larger the loopLength, the more resources are needed globally. Therefore, for maximal efficiency, loopLength should be as small as possible…

Summary: Feedback Loops for Computing in Space

- If an unrolled loop does not fit => DFESeqLoop
- For a loop with a loop-carried data dependence which cannot be unrolled, we need to create a loop in the data flow graph => DFEParLoop
- Interchanging loops and reorganising computations can reduce resource requirements
- Splitting loops into blocks ("tiling") allows us to control the amount of buffering required