CO405H

Computing in Space with OpenSPL
Topic 7: Programming DFEs
(Counters, Offsets and DFE mapping)

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/openspl16/
WebIDE: http://openspl.doc.ic.ac.uk
OpenSPL consortium page: http://www.openspl.org

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

Lecture Overview

- Counters / loop iteration variables
- Getting data in and out of the chip
- Stream offsets
- MaxCompiler hardware mapping

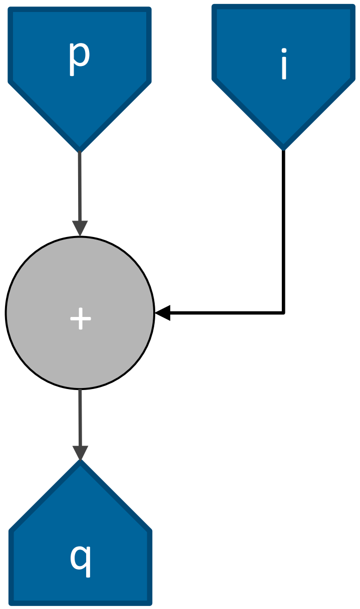
Working with Loop Counters

- How can we implement this in MaxCompiler?

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

How about this?

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

DFEVar q = p + i;

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



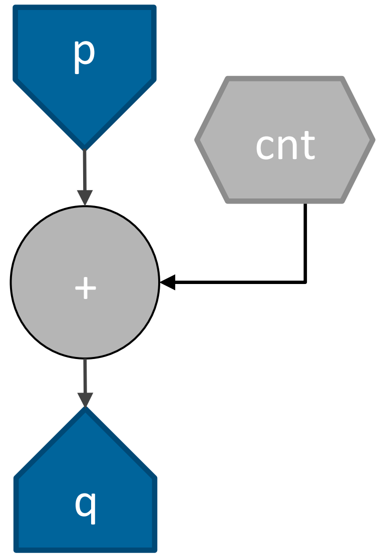
Yes…. But, now we need to create an array *i* in software and send it to the DFE as well

Working with Loop Counters

- There is very little 'information' in the *i* stream.
  - Could compute it directly on the DFE itself

```
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:

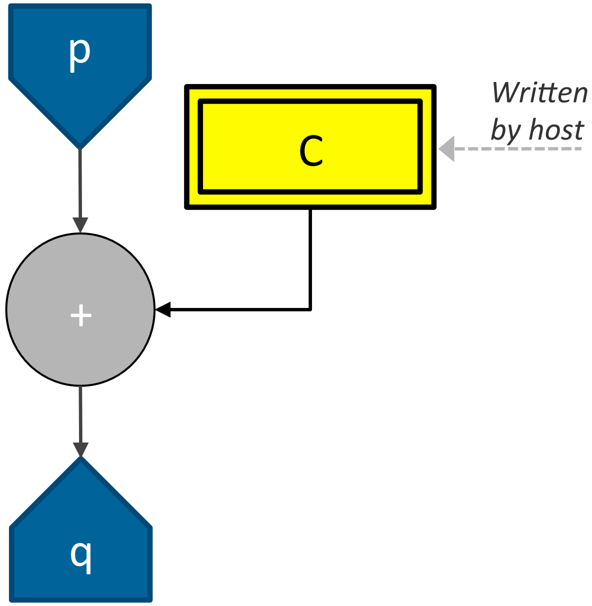
```
void fn1(int N, int *q, int *p) {
    for (int i = 0; i < N; i++)
        q[i] = p[i] + 4;
}
```

vs.

```
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:

```
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.

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.

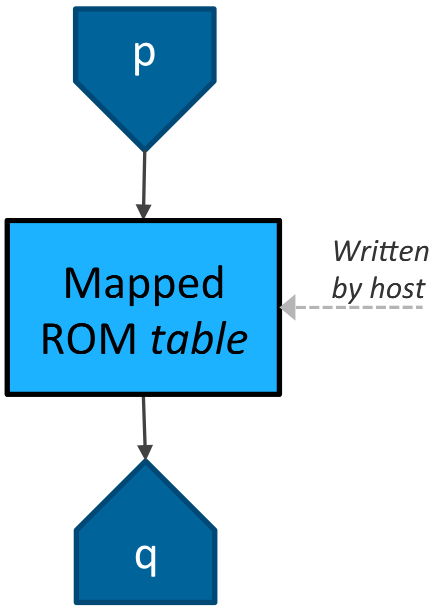
- Constants in expressions
- Flags to switch between two behaviours
  – result = **enabled** ? x+7 : x;
- Control parameters to counters, e.g. max, stride, etc
  – if (cnt==**cnt_max**) cnt=0; else cnt = cnt + **cnt_step**;

On-chip memories / tables

- A DFE has a few MB of very fast SRAM on the chip
- Can be used to explicitly store data on chip:
  - Lookup tables
  - Temporary Buffers
- *Mapped* ROMs/RAMs can also be accessed by host

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

```
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

Type	Size (items)	Host write speed	Chip area cost
Scalar input/output	1	Slow	Low
Mapped memory (ROM / RAM)	Up to a few thousand	Slow	Moderate
Stream input/output	Thousands to billions	Fast	Highest

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:

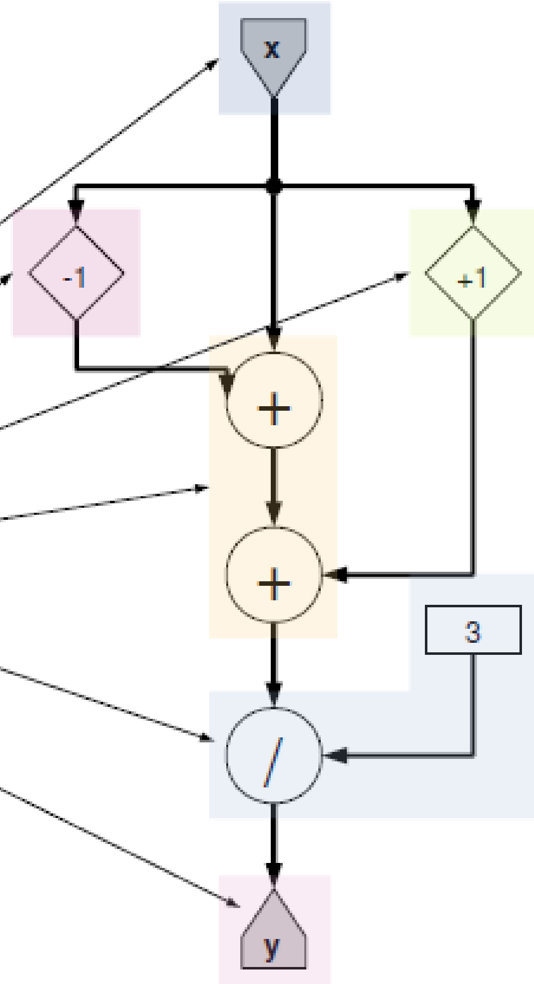
$$y_i = (x_{i-1} + x_i + x_{i+1}) / 3$$

Stream Offsets

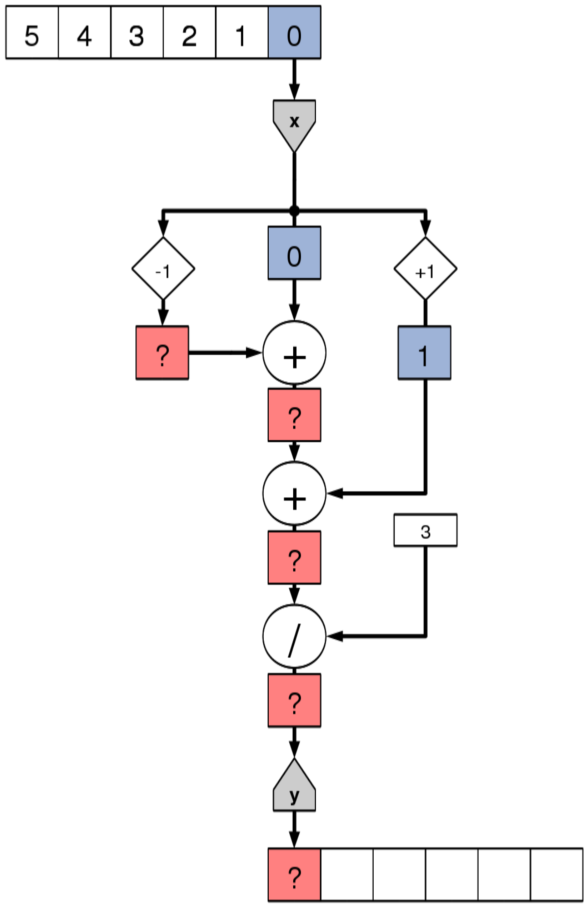
- *Stream offsets* allow us to compute on values in a stream other than the current value.
- Offsets are relative to the *current position* in a stream; *not* the start of the stream
- Stream data will be buffered on-chip in order to be available when needed → uses fast memory (FMEM)
  – Maximum supported offset size depends on the amount of on-chip SRAM available. Typically 10s of thousands of points.

Moving Average in MaxCompiler

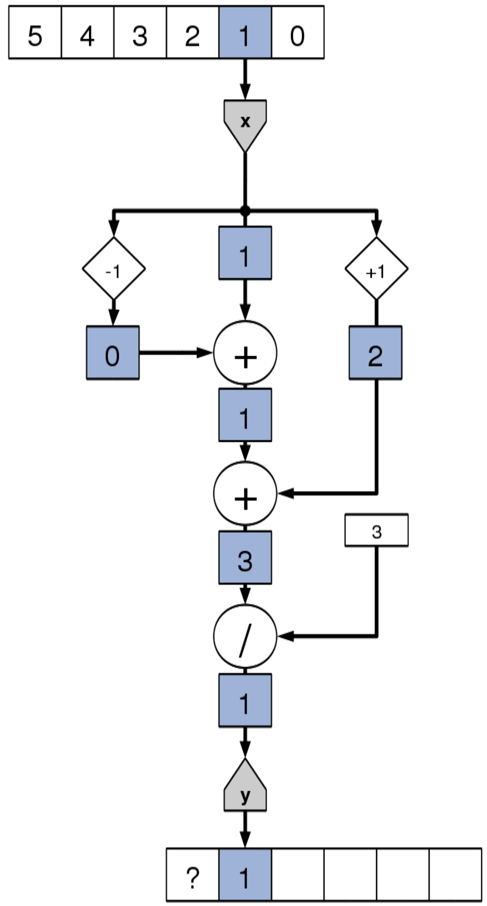
```
class MovingAverageSimpleKernel extends Kernel {

    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));
    }
}
```



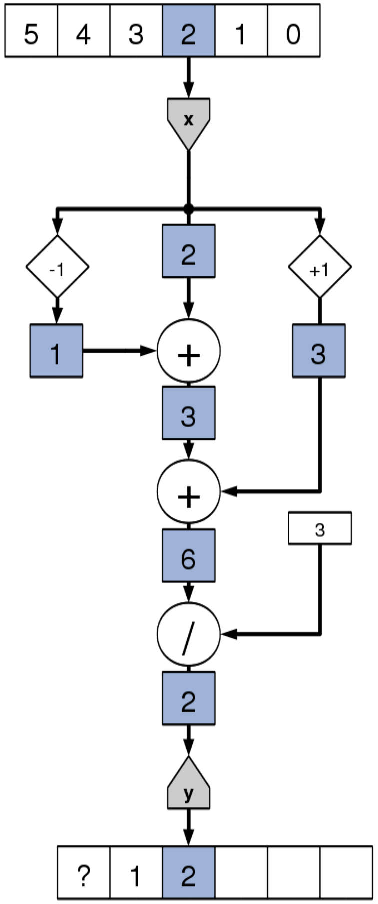
Kernel Execution



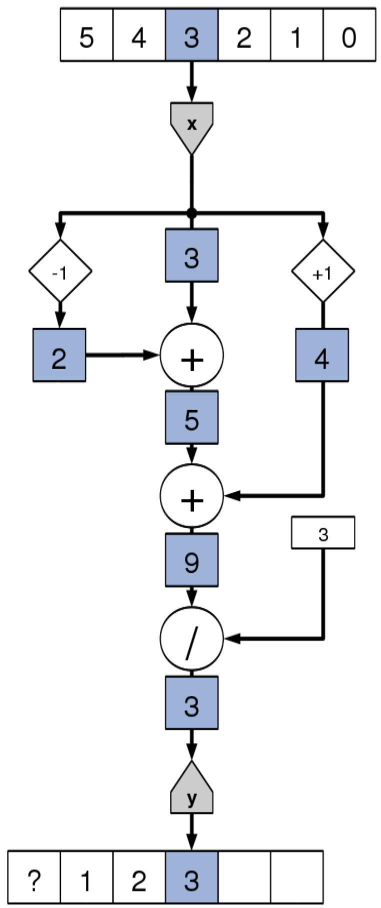
Kernel Execution



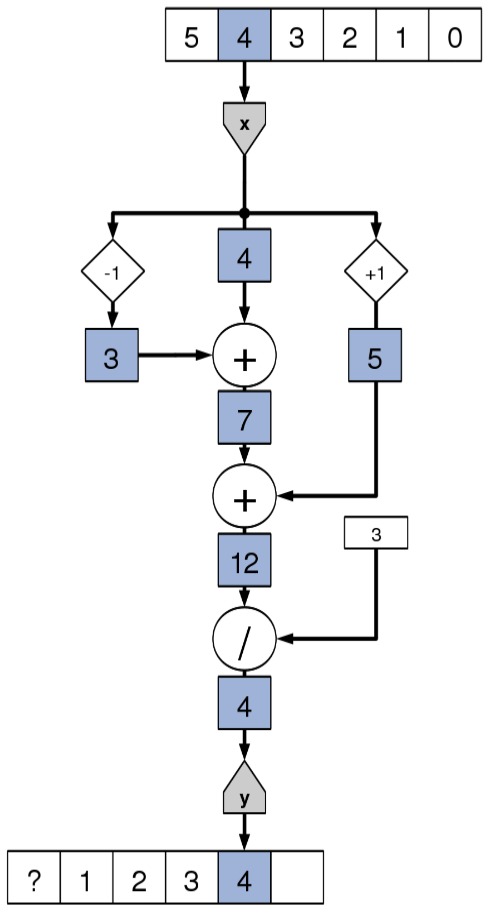
Kernel Execution



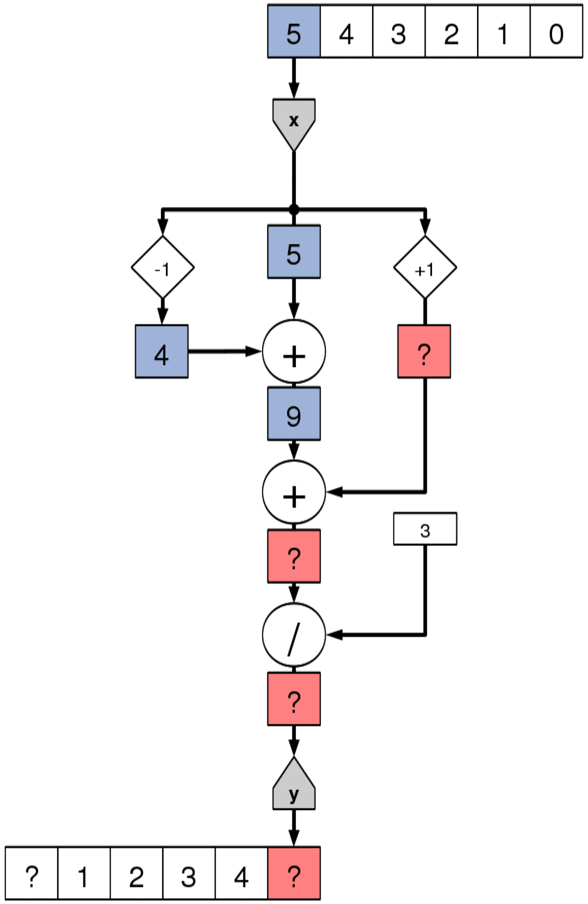
Kernel Execution



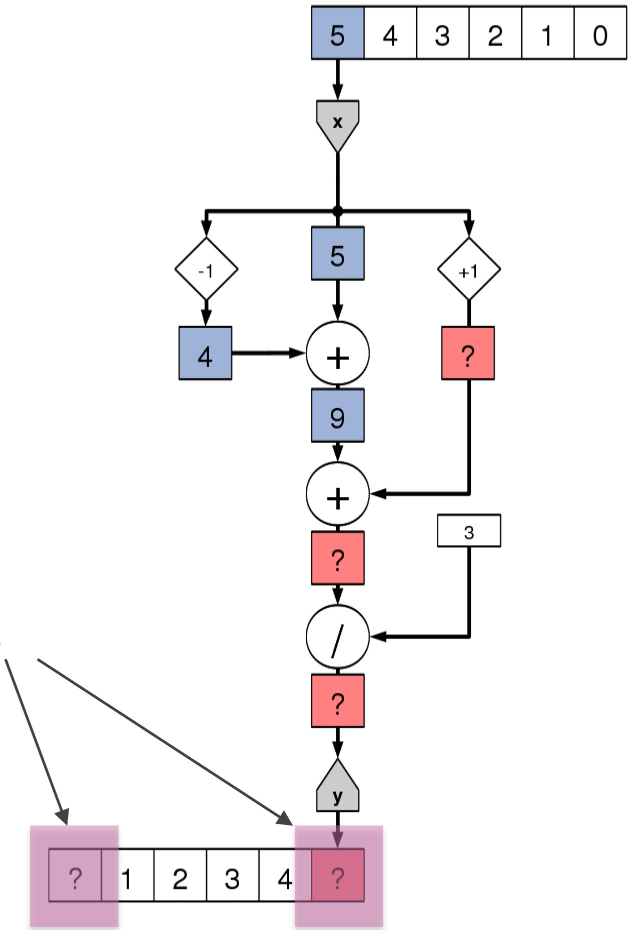
Kernel Execution



Kernel Execution



Boundary Cases



What about the boundary cases?

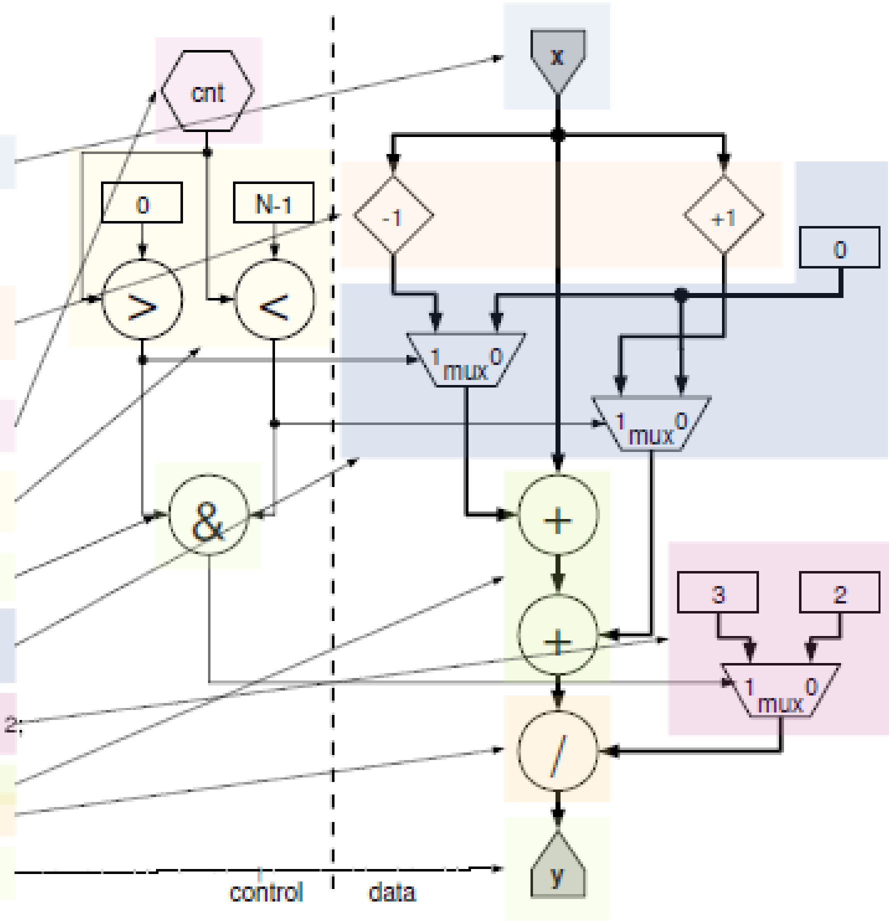
More Complex Moving Average

- 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

```
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 → linearize

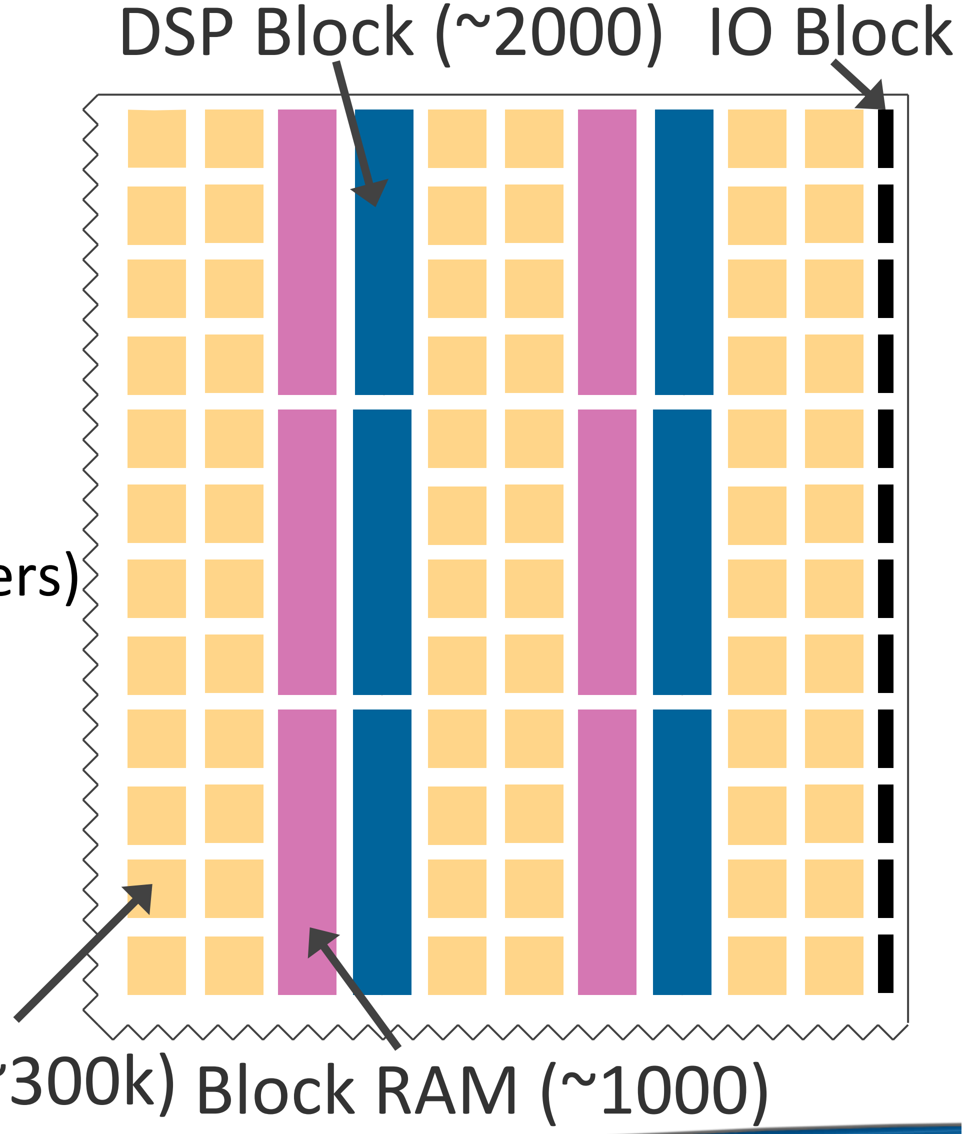
```
for (int y = 0; y < N; y++)
for (int x = 0; x < N; x++)
    p[y][x] = q[y-1][x] + q[y][x-1] + q[y][x] + q[y][x+1] + q[y+1][x]
```

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

And of course we now need to handle boundaries in both dimensions…

Optimisation of On-chip Resources

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



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

```
  LUTs      FFs    BRAMs     DSPs : MyKernel.java
   727      871      1.0        2 : resources used by this file
 0.24%    0.15%    0.09%    0.10% : % of available
71.41%   61.82%  100.00%  100.00% : % of total used
94.29%   97.21%  100.00%  100.00% : % of user resources
                                  :
                                  : public class MyKernel extends Kernel {
                                  :   public MyKernel (KernelParameters parameters) {
                                  :      super(parameters);
     1       31      0.0        0 :      DFEVar p = io.input("p", dfeFloat(8,24));
     2        9      0.0        0 :      DFEVar q = io.input("q", dfeUInt(8));
                                  :      DFEVar offset = io.scalarInput("offset", dfeUInt(8))
     8        8      0.0        0 :      DFEVar addr = offset + q;
    18       40      1.0        0 :      DFEVar v = mem.romMapped("table", addr,
                                  :                                    dfeFloat(8,24), 256);
   139      145      0.0        2 :      p = p * p;
   401      541      0.0        0 :      p = p + v;
                                  :      io.output("r", p, dfeFloat(8,24));
                                  :   }
                                  : }
```

Summary

- Counters help to reduce off chip traffic
- Choose the right variable type for your problem
- Offsets help but take care of boundary conditions
- Track the resource usage of your spatial code