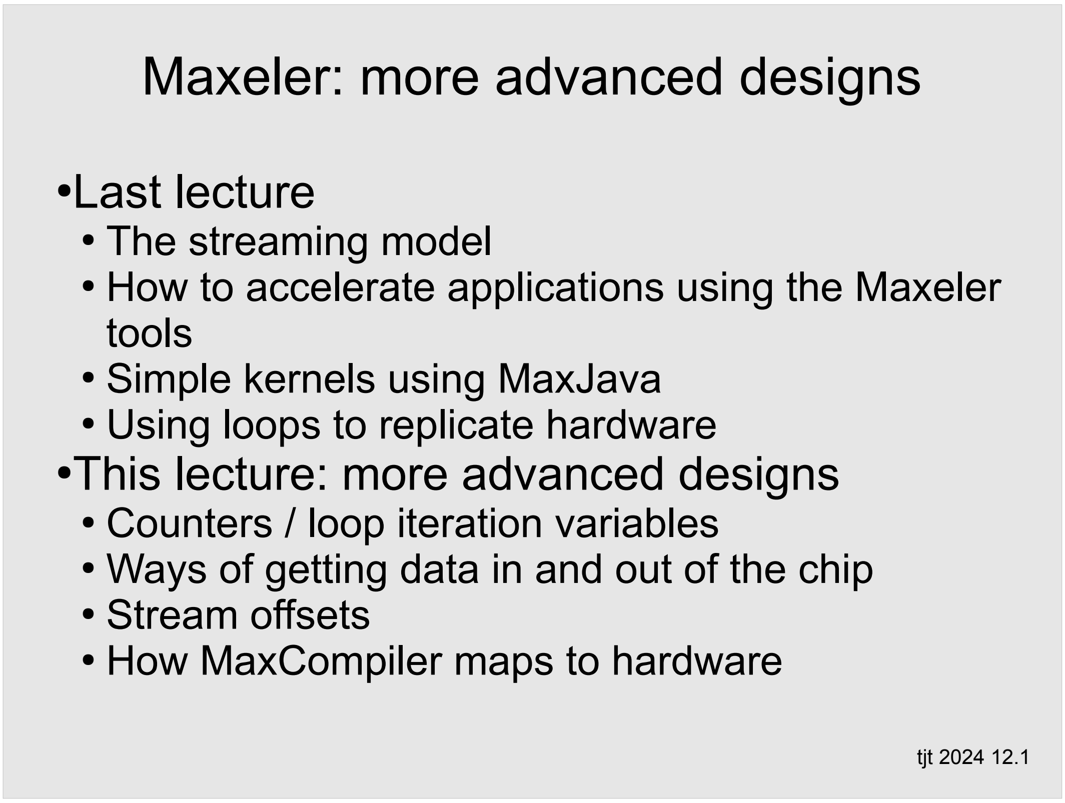
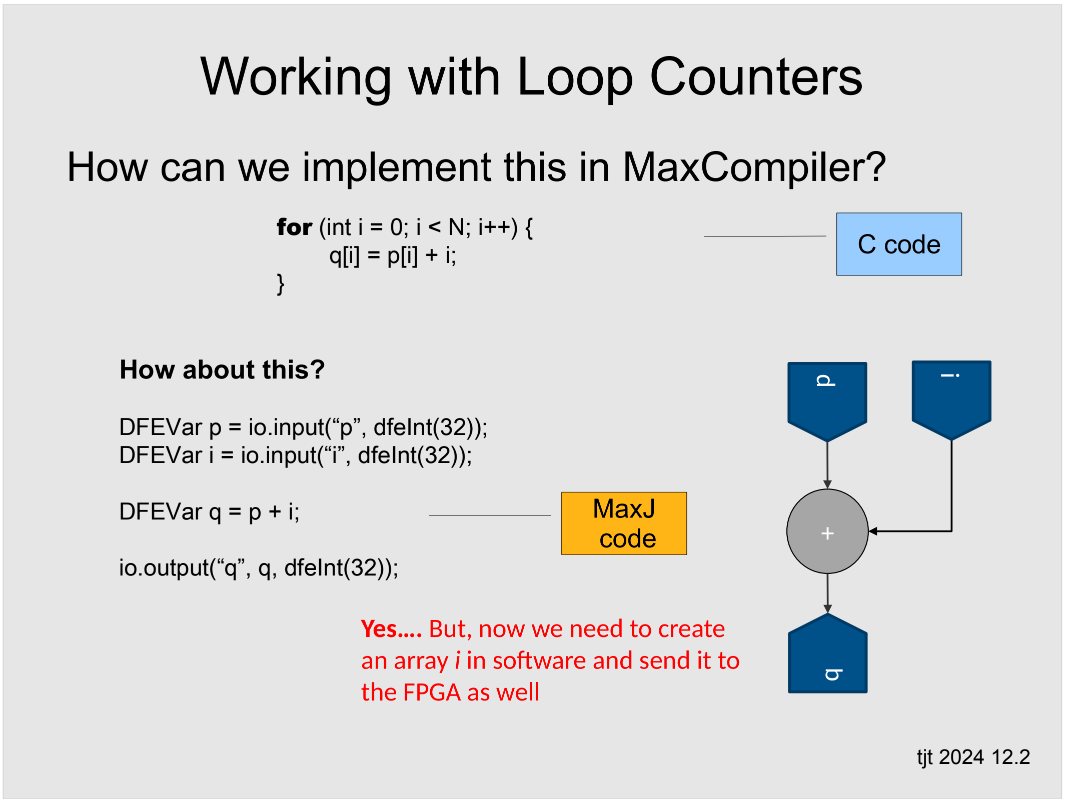


## Maxeler: more advanced designs

- Last lecture
  - The streaming model
  - How to accelerate applications using the Maxeler tools
  - Simple kernels using MaxJava
  - Using loops to replicate hardware
- This lecture: more advanced designs
  - Counters / loop iteration variables
  - Ways of getting data in and out of the chip
  - Stream offsets
  - How MaxCompiler maps to hardware

In the last lecture, we saw how to build FPGA designs with Maxeler. Remember with Maxeler you build your design as a dataflow graph using Maxeler’s Java compiler and class library.

The designs we’ve seen so far are simple: on each cycle, the graph reads one item from each stream input, and writes one item to each stream output. This is limiting; in this lecture we look at more complicated designs.



## Working with Loop Counters

How can we implement this in MaxCompiler?

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

C code

**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));
```

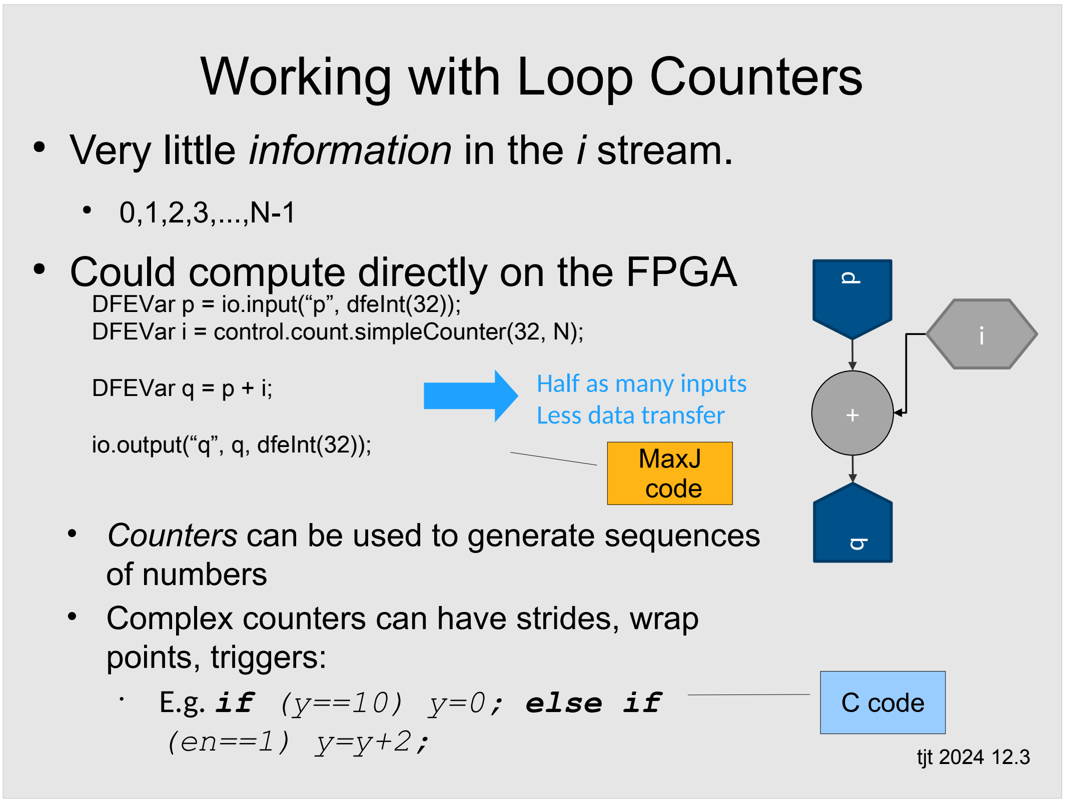
MaxJ code

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

tjt 2024 12.2

Simple example: how can you implement something depending on a loop's iteration variable ("i" in this example)? You already know how to do this: generate the values of I in software and pass it in as another stream variable.

This works, but is inefficient.



## Working with Loop Counters

- Very little *information* in the *i* stream.
  - 0,1,2,3,...,N-1
- Could compute directly on the FPGA

```
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

MaxJ code

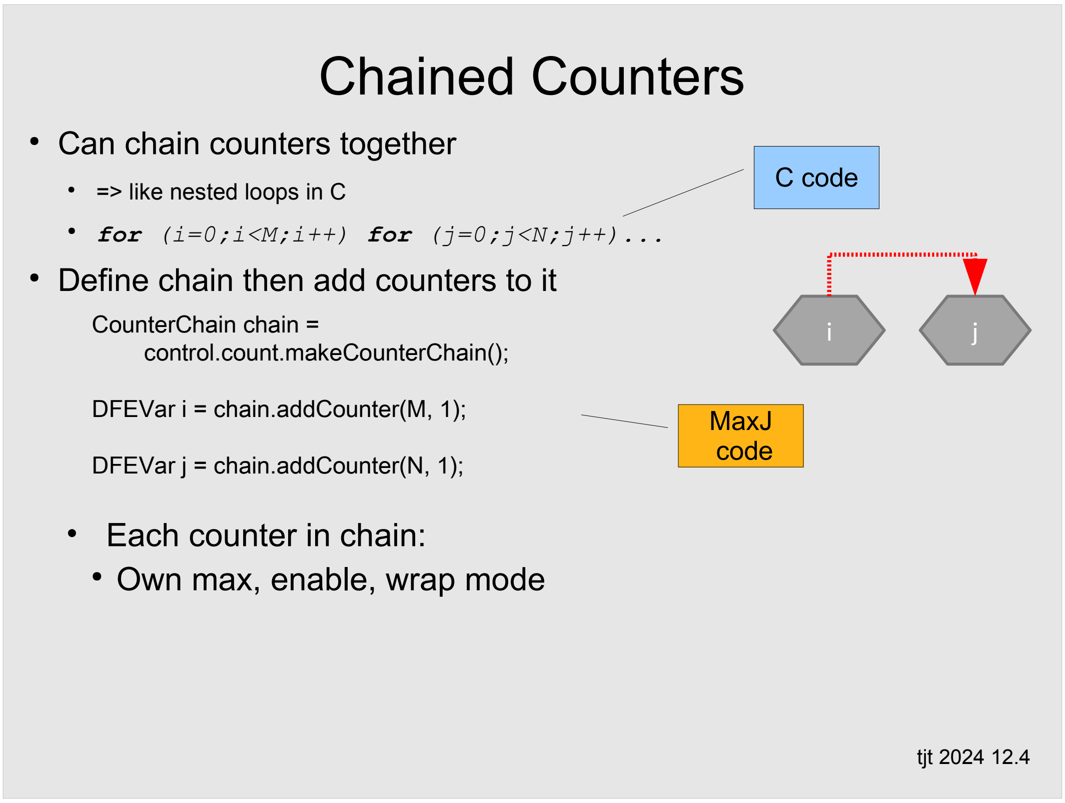
- *Counters* can be used to generate sequences of numbers
- Complex counters can have strides, wrap points, triggers:
  - E.g. ***if*** *(y==10) y=0;* ***else if*** *(en==1) y=y+2;*

C code

tjt 2024 12.3

Maxeler loop counters can generate sequences of numbers, one per cycle. In this example, using a loop counter saves half the input data transfer requirement (we'll look at performance in a couple of lectures' time).

Counters can have:
- strides (increment amount)
- wrap points (value before going back to start)
- triggers (Boolean condition when to count)



## Chained Counters

- Can chain counters together
  - => like nested loops in C
  - `for (i=0;i<M;i++) for (j=0;j<N;j++)...`
- Define chain then add counters to it

```
CounterChain chain =
        control.count.makeCounterChain();

DFEVar i = chain.addCounter(M, 1);

DFEVar j = chain.addCounter(N, 1);
```

- Each counter in chain:
  - Own max, enable, wrap mode

C code

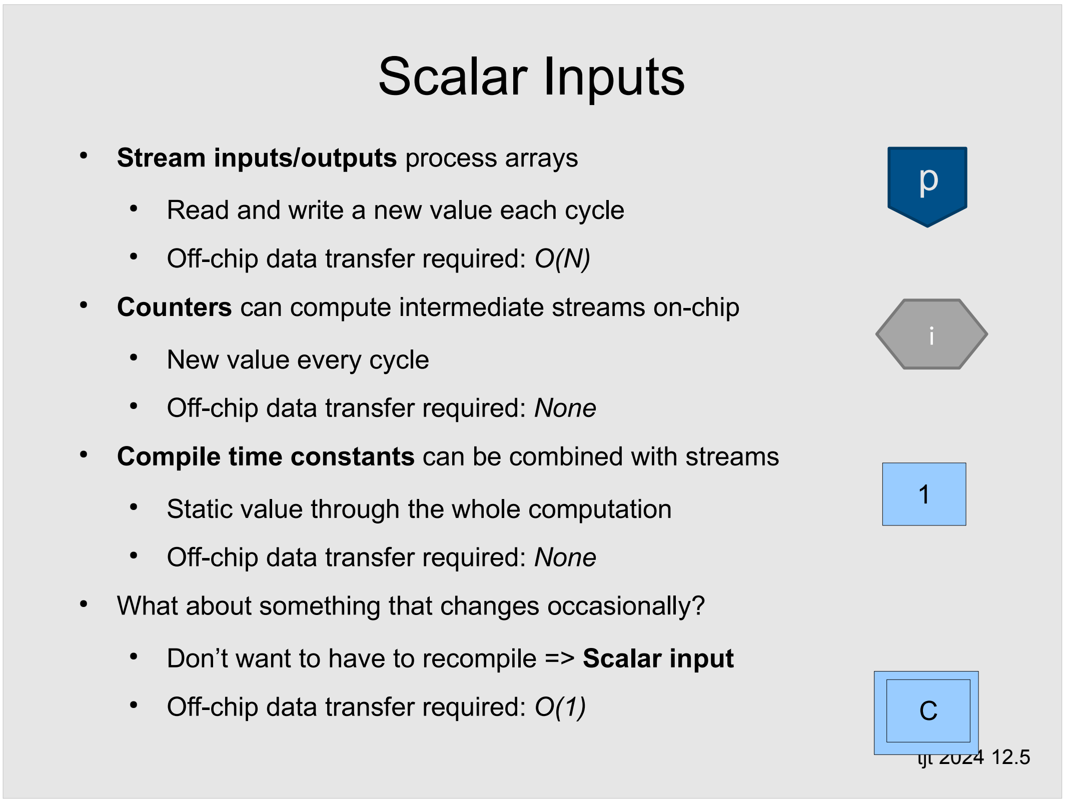
MaxJ code

tjt 2024 12.4

Counter chains give you the same patterns as the iteration variables of nested loops in C.

First build a counter chain, then add counters to it. Note the order: major counter added to the chain first.

Each counter in the chain can have separate:
- max value
- enable (Boolean trigger condition)
- wrap mode (saturate, loop back to zero, etc.)

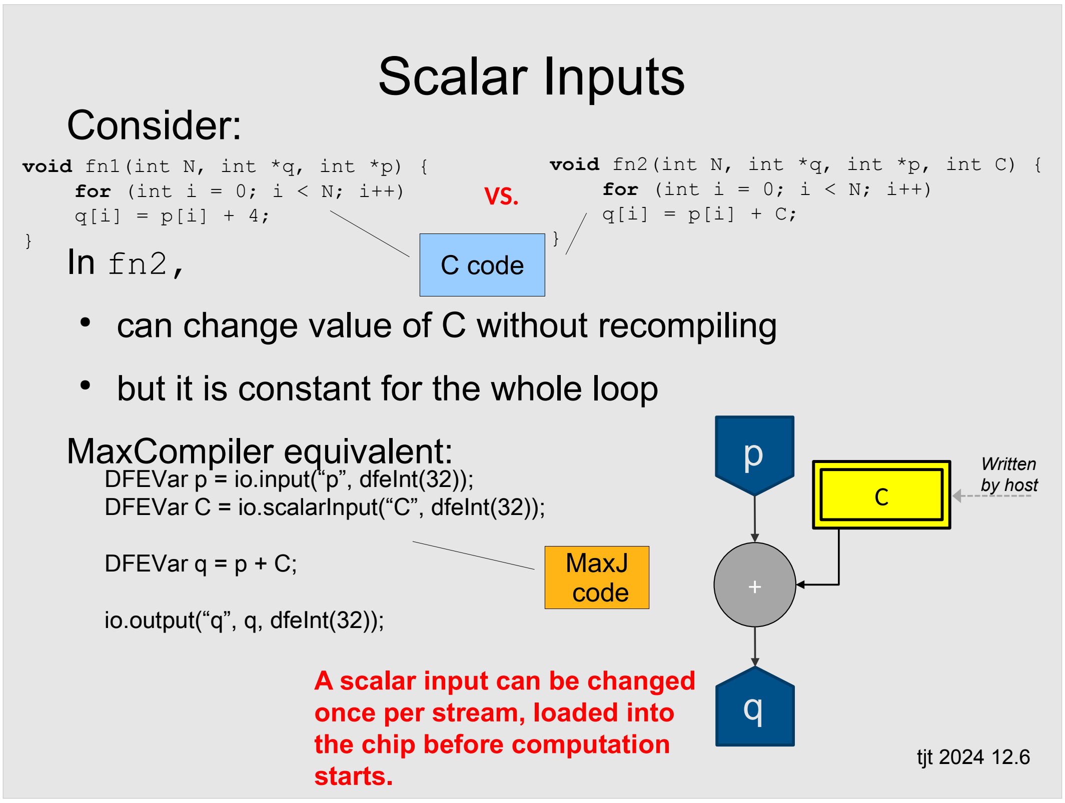


## Scalar Inputs

- **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)*

This slide summarises various different ways to get data into the kernel. Each has different tradeoffs: amount of data you can transfer versus on-chip size.

Scalar inputs: you can change them once per stream run. When set, the value does not change for the entire stream run.



# 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;
}
```

C code

In `fn2`,

- can change value of C without recompiling
- but it is constant for the whole loop

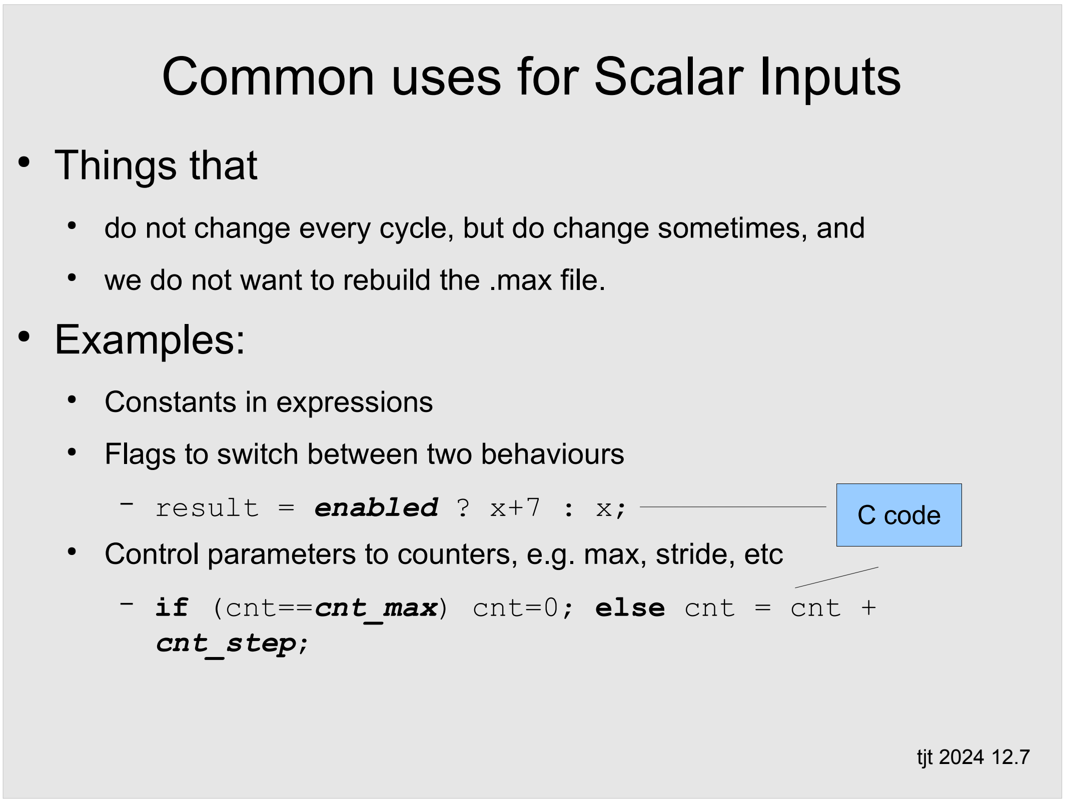
MaxCompiler equivalent:

```
DFEVar p = io.input(“p”, dfeInt(32));
DFEVar C = io.scalarInput(“C”, dfeInt(32));

DFEVar q = p + C;

io.output(“q”, q, dfeInt(32));
```

MaxJ code

p
C
*Written by host*
+
q

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

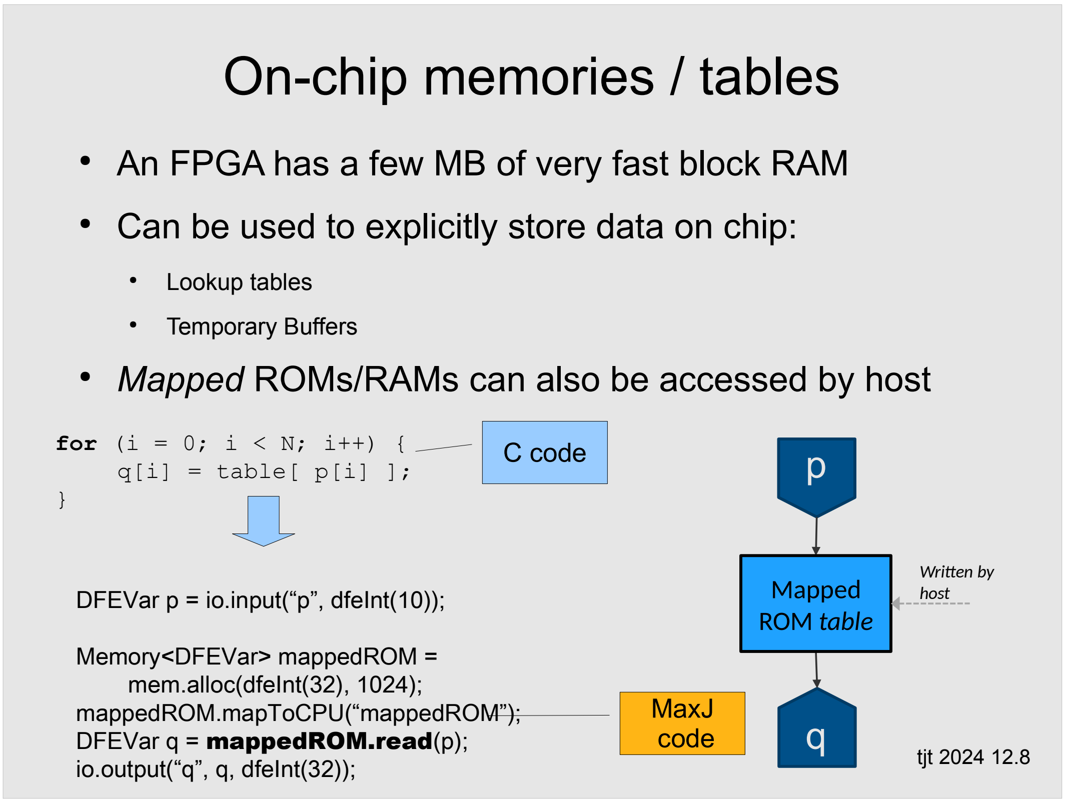


## Common uses for Scalar Inputs

- Things that
  - do not change every cycle, but do change sometimes, and
  - we do not want to rebuild the .max file.
- Examples:
  - 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;
      ```

C code

Remember that rebuilding the .max file can take hours to days for large designs.



# 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

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

C code

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

Memory<DFEVar> mappedROM =
       mem.alloc(dfeInt(32), 1024);
mappedROM.mapToCPU("mappedROM");
DFEVar q = mappedROM.read(p);
io.output("q", q, dfeInt(32));
```

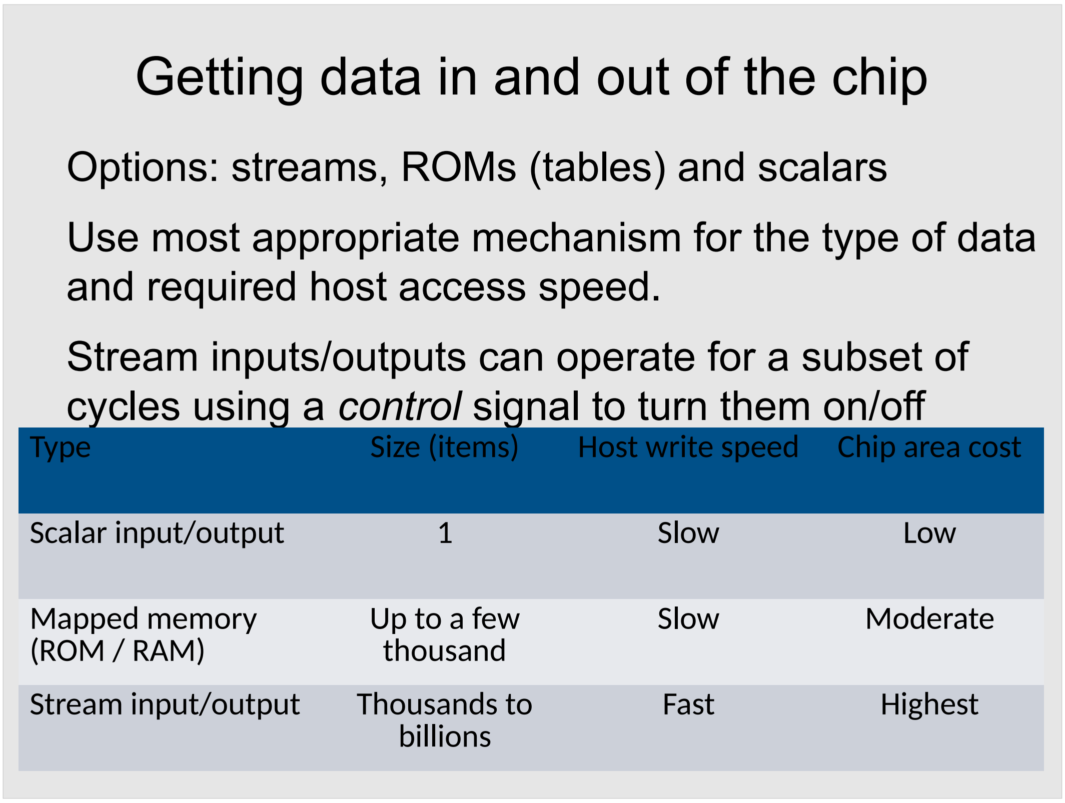
MaxJ code

tjt 2024 12.8

You can also store data on-chip in mapped ROMs and RAMs – “mapped” means they can be altered by the host. FPGAs have a few megabytes of on-chip memory.

Note the mappedROM.read call which connects the index (address) to the ROM.

Why is the input of size 10 bits? Because the ROM is of size 1024, and we need 10 bits to address 1024 elements (2^10=1024).



## Getting data in and out of the chip

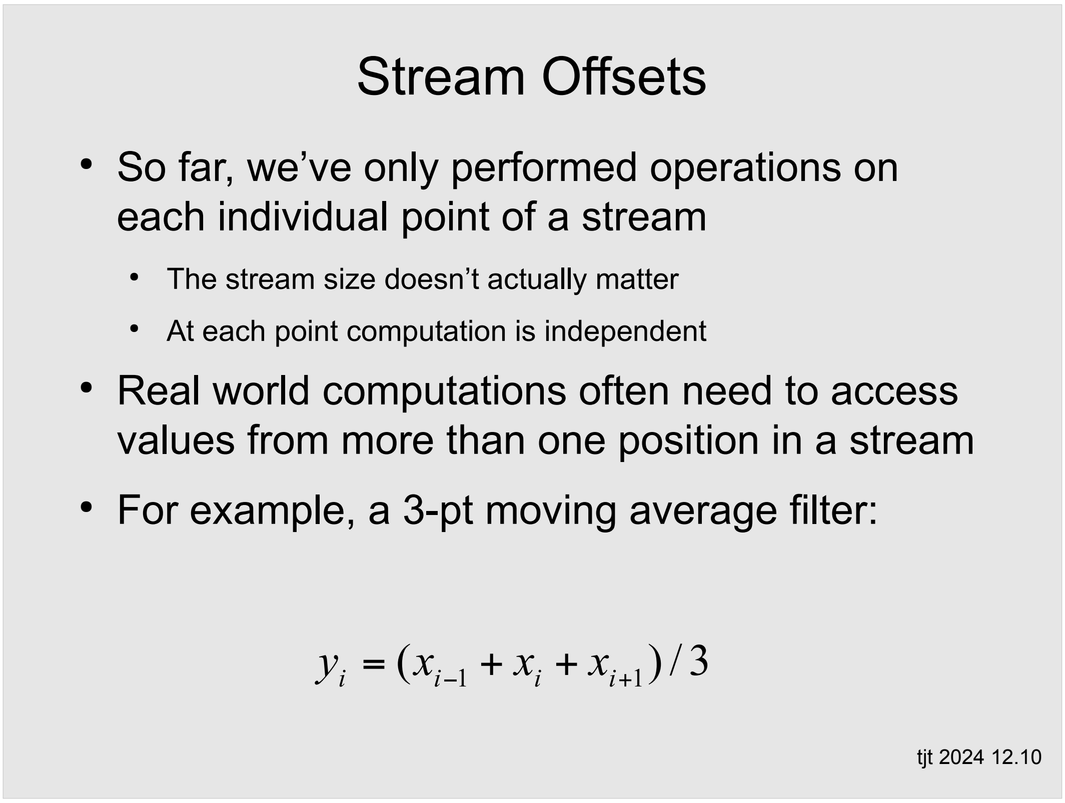
Options: streams, ROMs (tables) and scalars

Use 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 |

The table summarises various tradeoffs for getting data to and from the FPGA. Don’t learn the table by heart, instead understand the tradeoffs involved: size versus speed versus area cost.



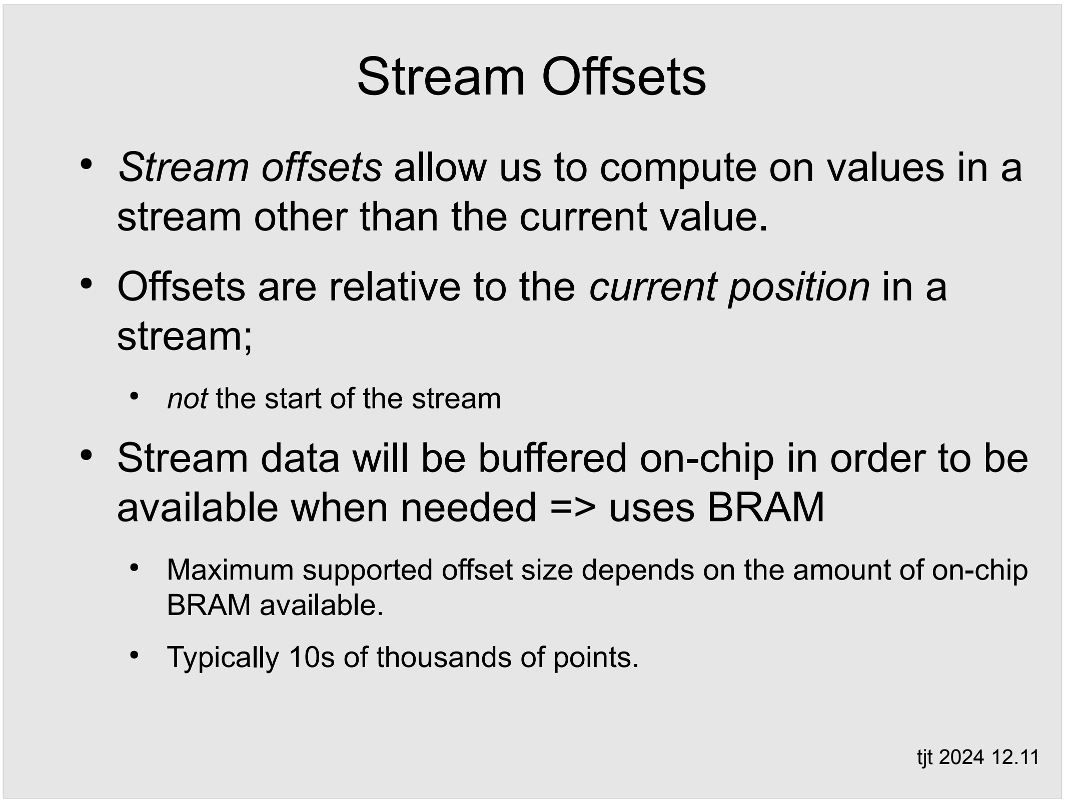
## Stream Offsets

- So far, we've only performed operations on each individual point of a stream
  - The stream size doesn't actually matter
  - 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 let you compute on the neighbours of each stream element.

Filters have many uses:
- smooth noisy signals (e.g. control, finance)
- remove high frequency information
- machine learning (convolution steps)



## Stream Offsets

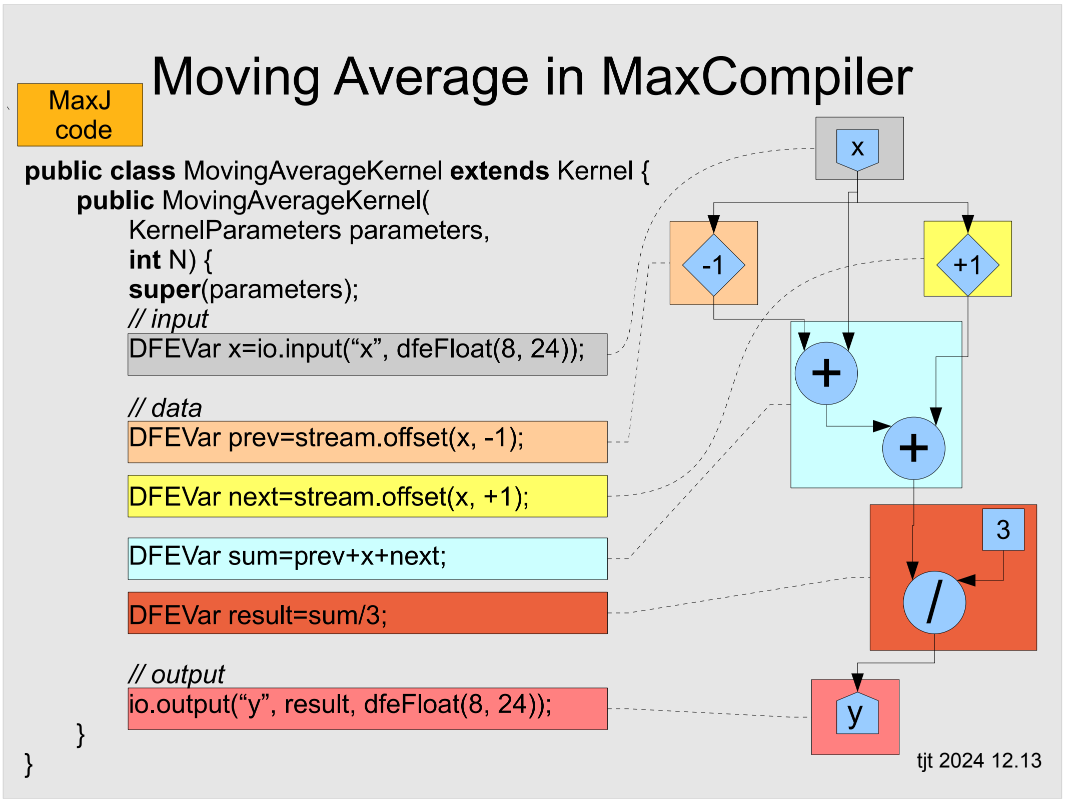
- *Stream offsets* allow us to compute on values in a stream other than the current value.
- Offsets are relative to the *current position* in a stream;
  - *not* the start of the stream
- Stream data will be buffered on-chip in order to be available when needed => uses BRAM
  - Maximum supported offset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

tjt 2024 12.11

Note that the Maxeler system automatically optimises the implementation of stream offsets for you (choosing shift registers or RAMs depending on offset size).

The offset can be fixed or variable – in the latter case, you need to give a value for the maximum offset.

You can also compare stream offsets to delays and antidelays in Ruby (negative stream offsets work like delays, positive like antidelays, but can always be implemented).



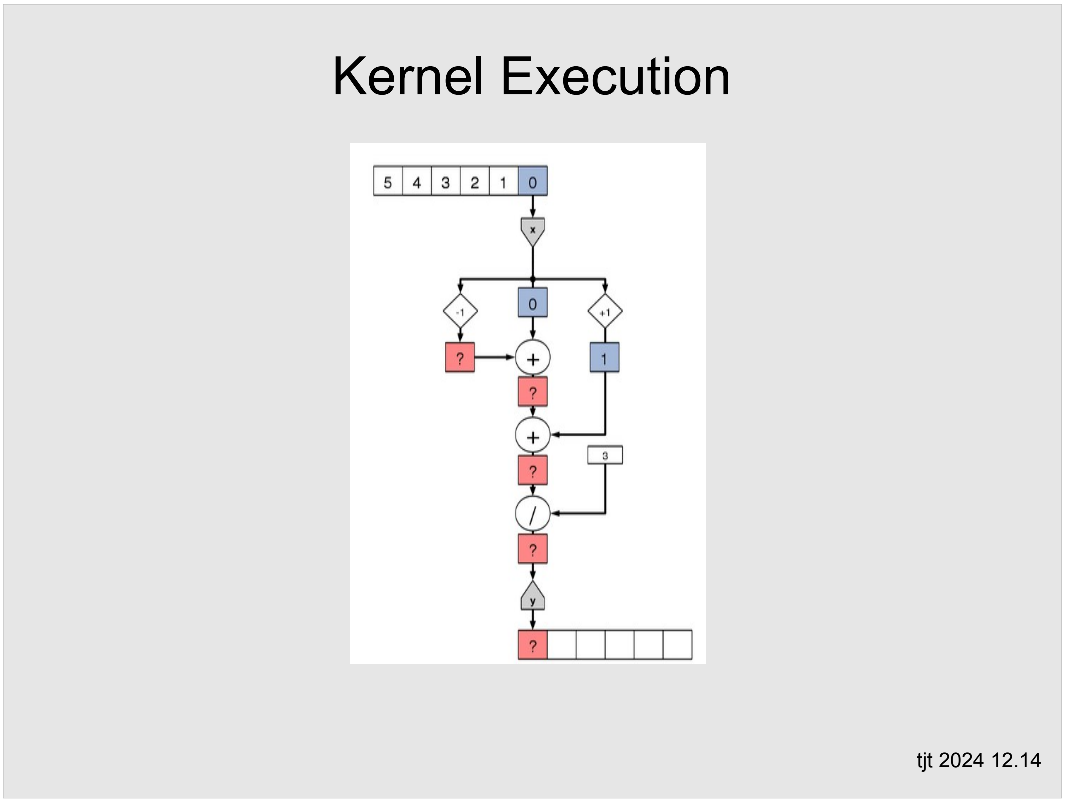
# Moving Average in MaxCompiler

MaxJ code

```
public class MovingAverageKernel extends Kernel {
      public MovingAverageKernel(
            KernelParameters parameters,
            int N) {
            super(parameters);
            // input
            DFEVar x=io.input(“x”, dfeFloat(8, 24));

            // data
            DFEVar prev=stream.offset(x, -1);

            DFEVar next=stream.offset(x, +1);

            DFEVar sum=prev+x+next;

            DFEVar result=sum/3;

            // output
            io.output(“y”, result, dfeFloat(8, 24));
      }
}
```

tjt 2024 12.13

Moving average – again, note each line corresponds to one part of the dataflow graph.



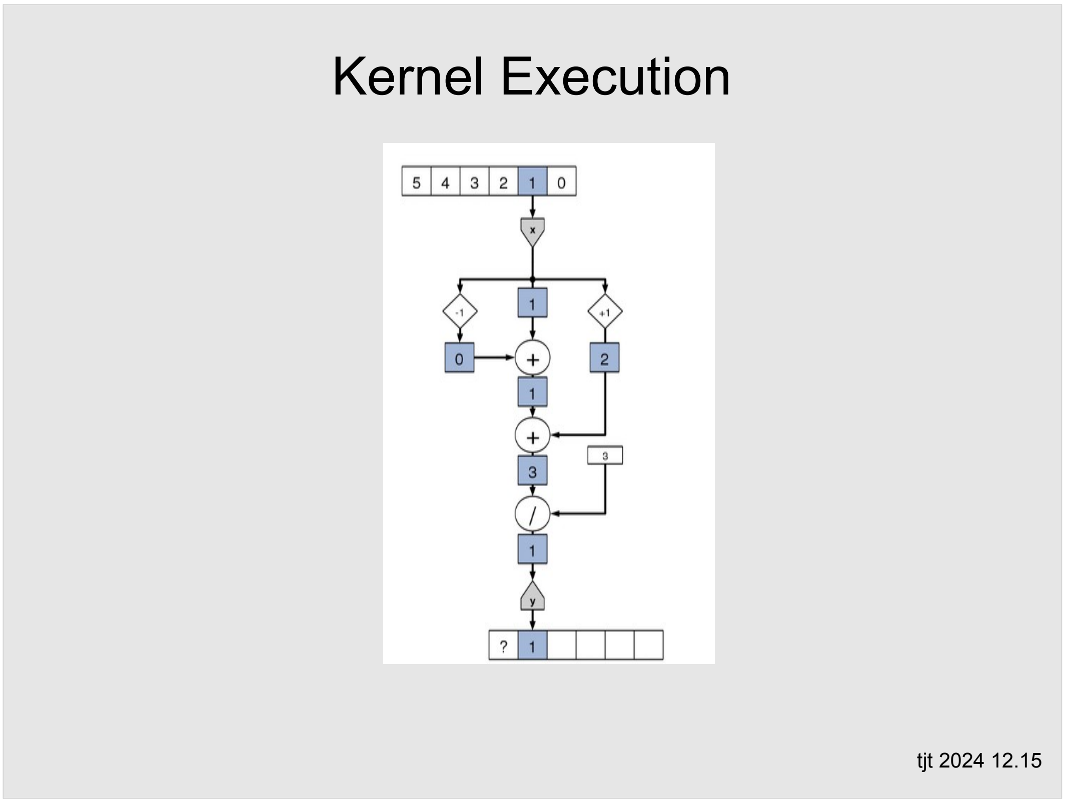
## Kernel Execution

tjt 2024 12.14

Let’s simulate the execution of the moving average with stream offsets.

This is cycle number one.

What happens when we try to look off the edges of the stream (negative stream offset at the start of the stream)? You get an undefined value, or “?” in the diagram.

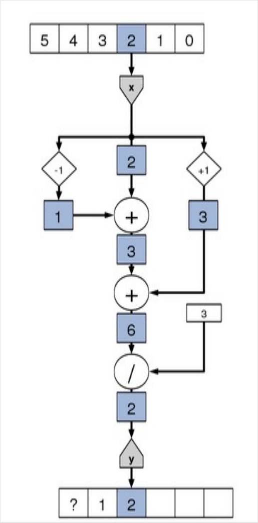


## Kernel Execution

tjt 2024 12.15

Cycle 2. Now the offsets are OK since we’re not looking outside the stream.

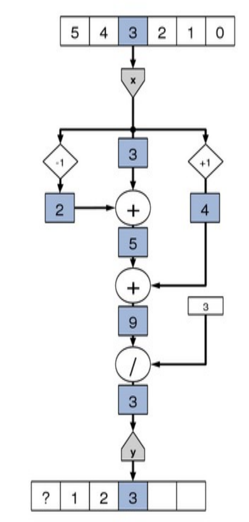
# Kernel Execution



tjt 2024 12.16

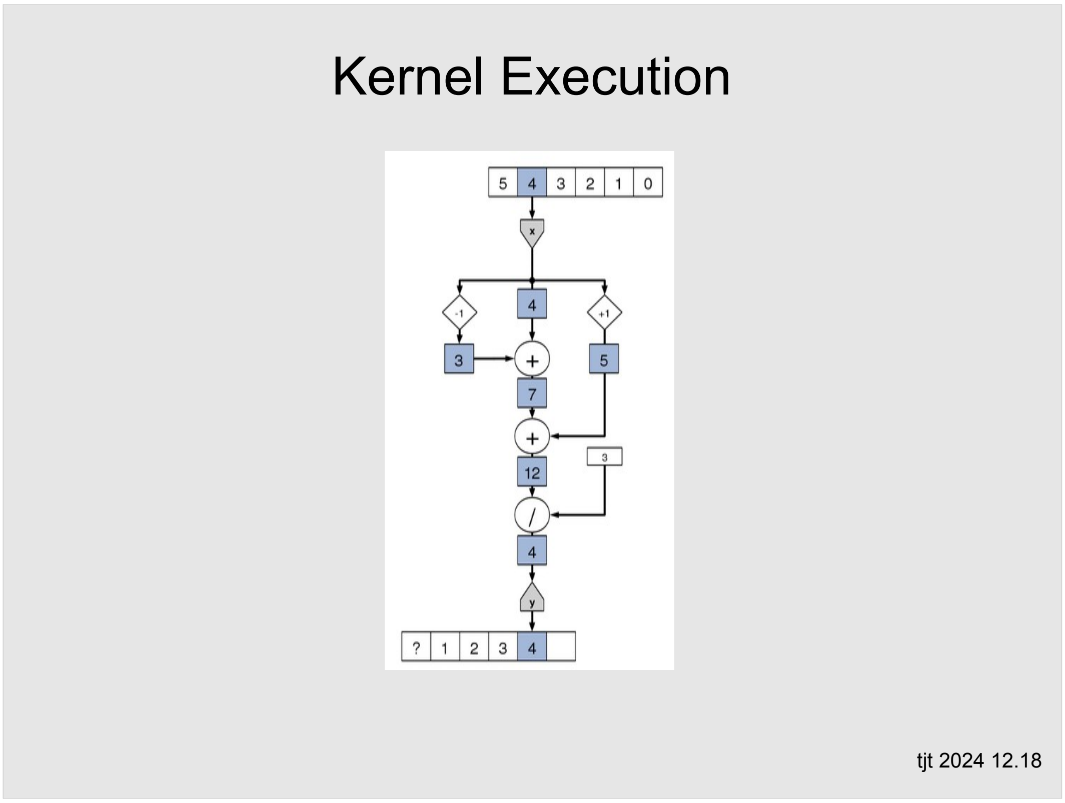
Cycle 3.

# Kernel Execution



tjt 2024 12.17

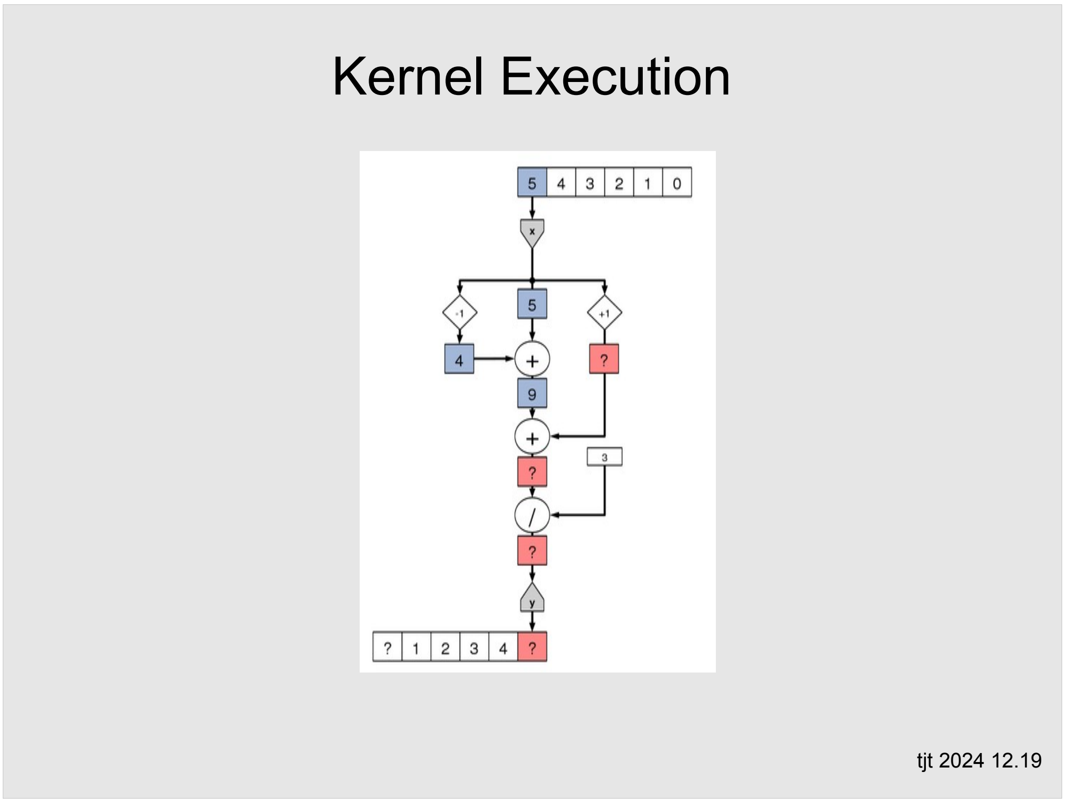
Cycle 4.



## Kernel Execution

tjt 2024 12.18

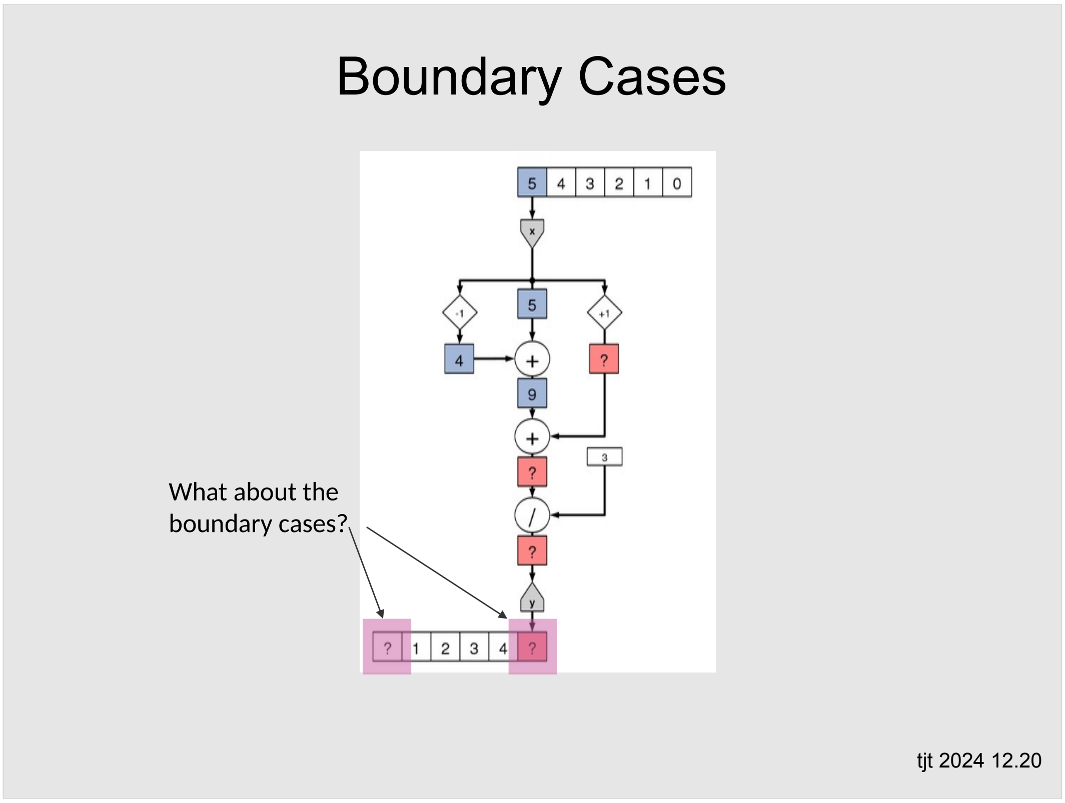
Cycle 5.



# Kernel Execution

tjt 2024 12.19

Cycle 6. Note that a positive stream offset that causes us to read outside the stream again results in an undefined value “?”.

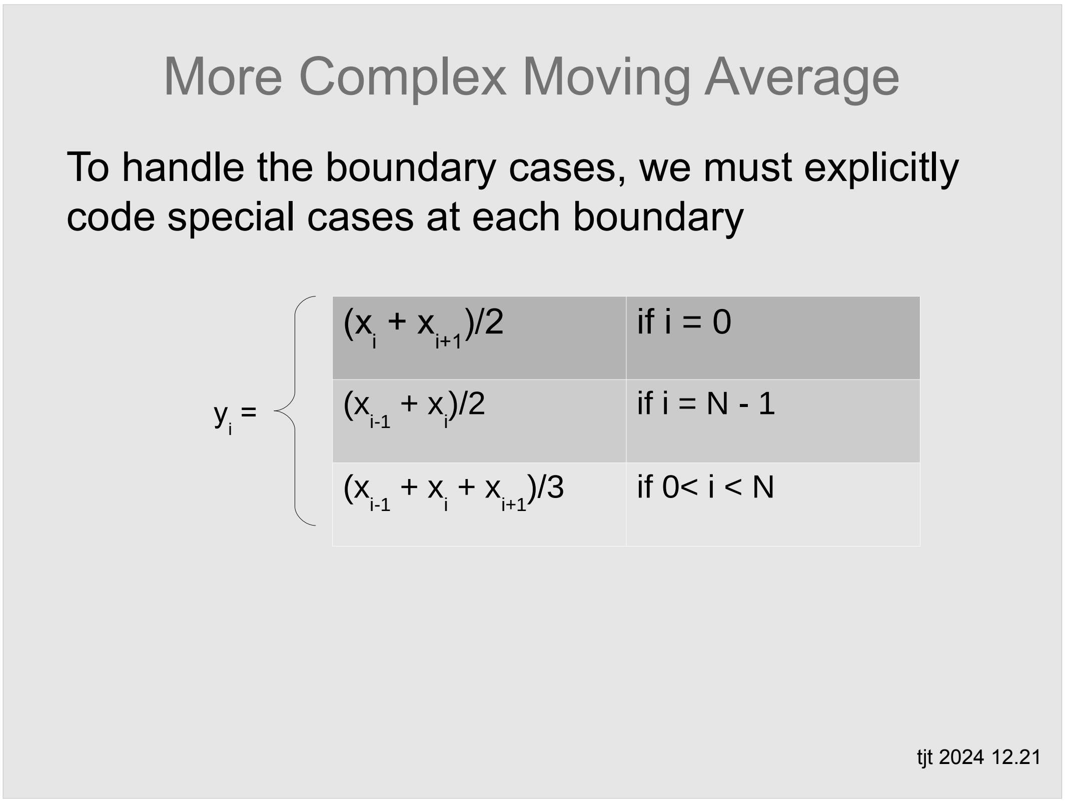


## Boundary Cases

What about the boundary cases?

tjt 2024 12.20

How to deal with these boundary cases and the undefined values?

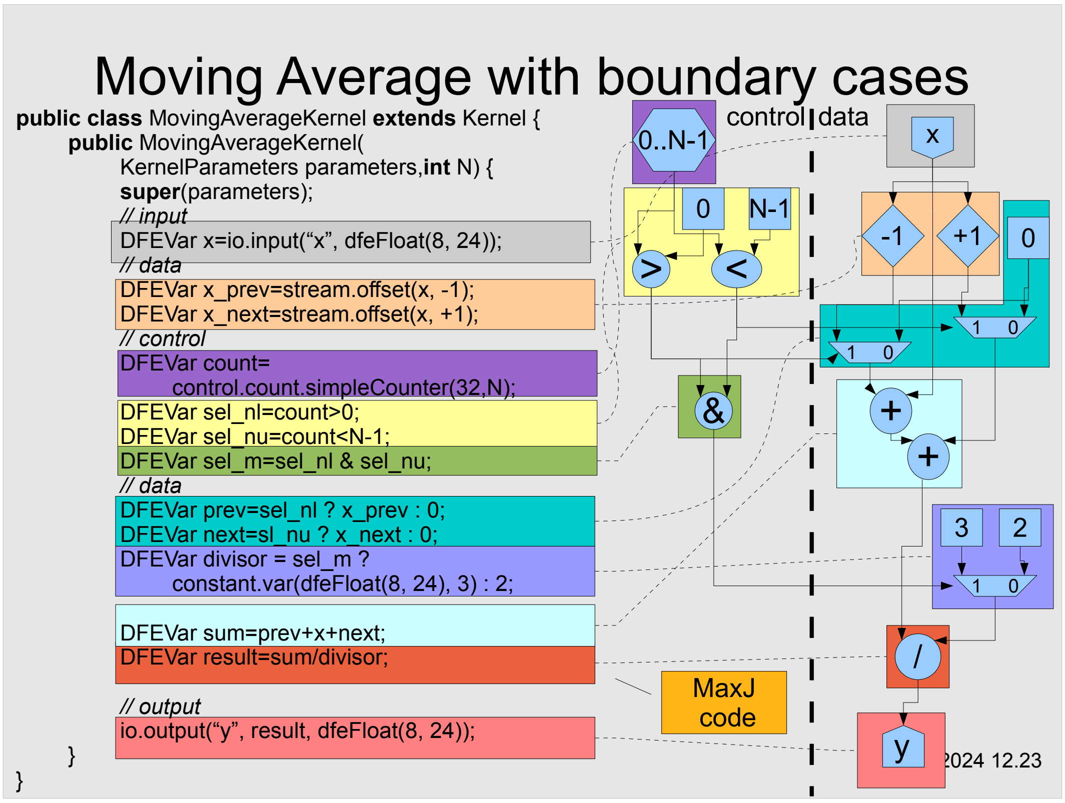


## More Complex Moving Average

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

$$y_i = \begin{cases} (x_i + x_{i+1})/2 & \text{if } i = 0 \\ (x_{i-1} + x_i)/2 & \text{if } i = N - 1 \\ (x_{i-1} + x_i + x_{i+1})/3 & \text{if } 0< i < N \end{cases}$$

One way is to alter the definition of moving average so you never read outside the stream.



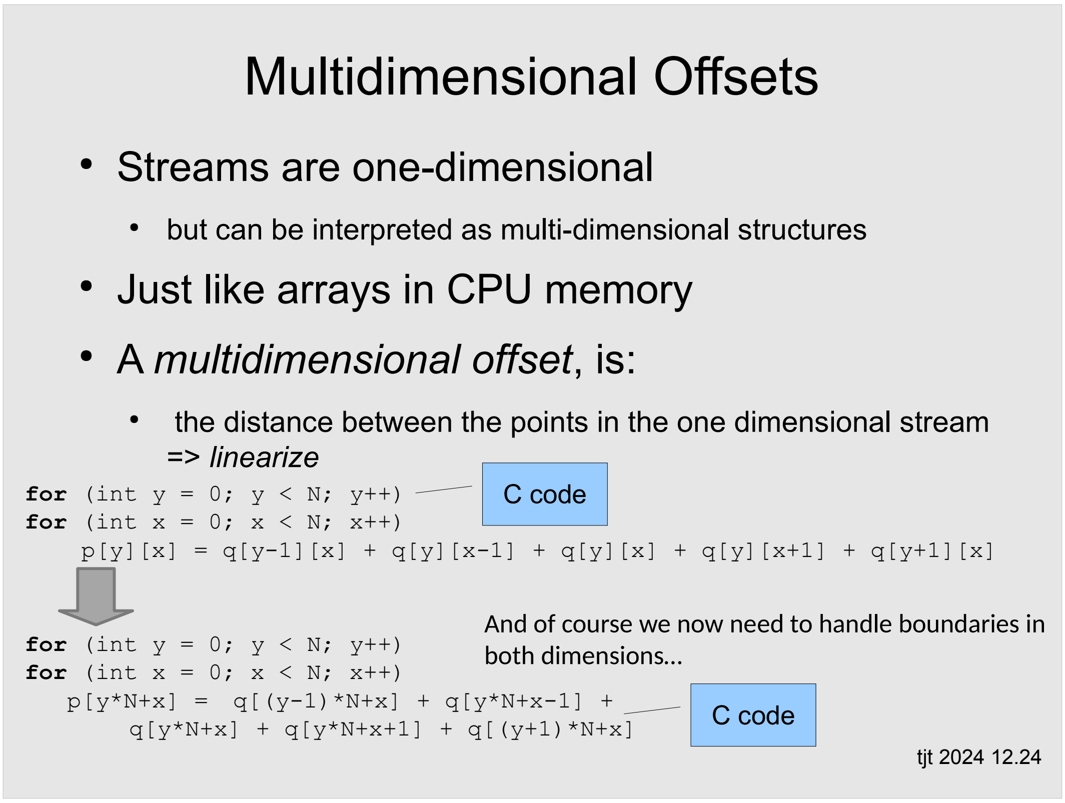
## Moving Average with boundary cases

```
public class MovingAverageKernel extends Kernel {
        public MovingAverageKernel(
                KernelParameters parameters,int N) {
                super(parameters);
                // input
                DFEVar x=io.input(“x”, dfeFloat(8, 24));
                // data
                DFEVar x_prev=stream.offset(x, -1);
                DFEVar x_next=stream.offset(x, +1);
                // control
                DFEVar count=
                        control.count.simpleCounter(32,N);
                DFEVar sel_nl=count>0;
                DFEVar sel_nu=count<N-1;
                DFEVar sel_m=sel_nl & sel_nu;
                // data
                DFEVar prev=sel_nl ? x_prev : 0;
                DFEVar next=sl_nu ? x_next : 0;
                DFEVar divisor = sel_m ?
                        constant.var(dfeFloat(8, 24), 3) : 2;

                DFEVar sum=prev+x+next;
                DFEVar result=sum/divisor;

                // output
                io.output(“y”, result, dfeFloat(8, 24));
        }
}
```

2024 12.23

To do this, use a counter to detect where you are in the stream, so you never compute with an undefined value.

Note the separation between the control part of the graph (on the left) and the data (on the right).



## 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]
```

C code

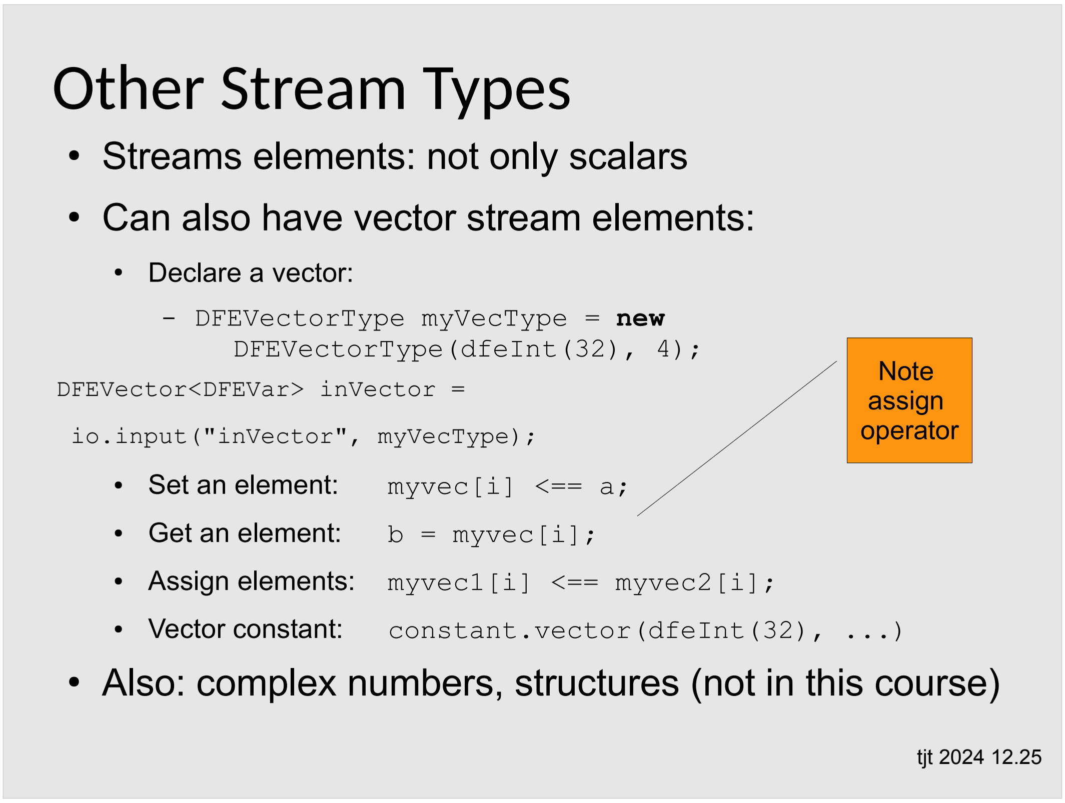
```
for (int y = 0; y < N; y++)
for (int x = 0; x < N; x++)
   p[y*N+x] =  q[(y-1)*N+x] + q[y*N+x-1] +
        q[y*N+x] + q[y*N+x+1] + q[(y+1)*N+x]
```

C code

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

tjt 2024 12.24

Offsets can also be multi-dimensional, but need to linearize the expressions.



# Other Stream Types

- Streams elements: not only scalars
- Can also have vector stream elements:
  - Declare a vector:

```
  - DFEVectorType myVecType = new
       DFEVectorType(dfeInt(32), 4);
DFEVector<DFEVar> inVector =

 io.input("inVector", myVecType);
```

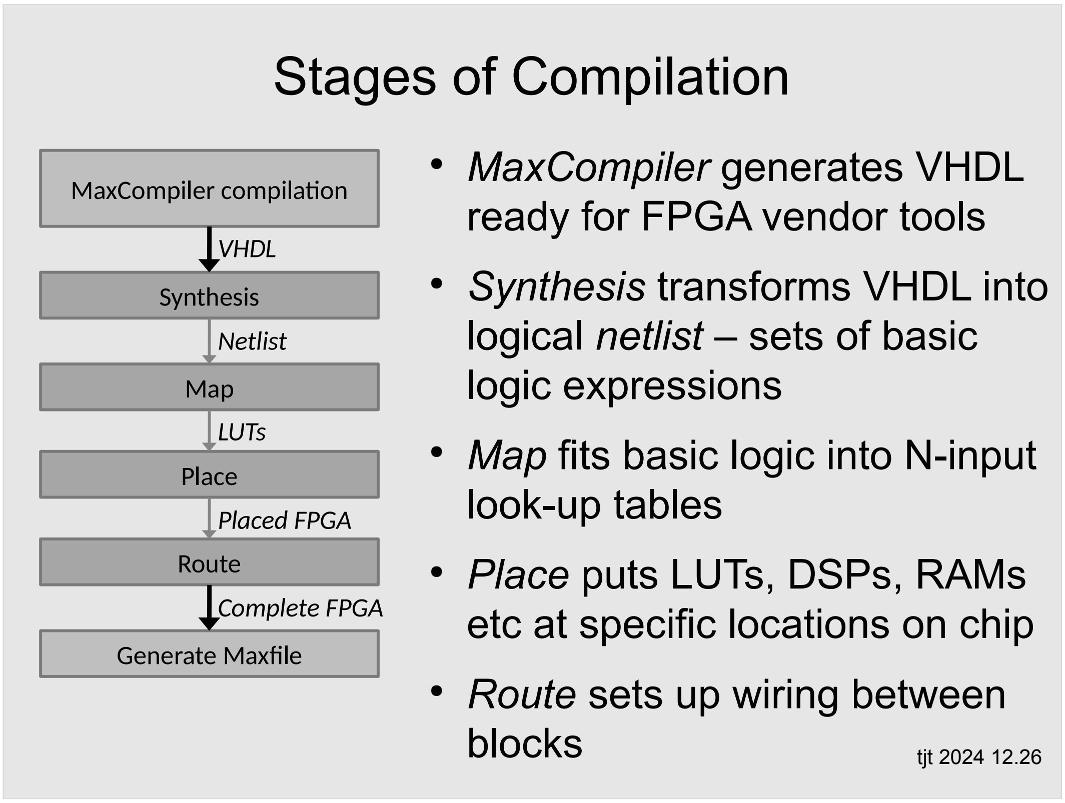
  - Set an element: `myvec[i] <== a;`
  - Get an element: `b = myvec[i];`
  - Assign elements: `myvec1[i] <== myvec2[i];`
  - Vector constant: `constant.vector(dfeInt(32), ...)`
- Also: complex numbers, structures (not in this course)

Note assign operator

tjt 2024 12.25

So far you've seen 3 stream types:
- unsigned integers (dfeUInt(N))
- signed integers (dfeInt(N))
- floating-point numbers (dfeFloat(E,M))

The stream element type can also be a vector. Complex numbers and structures are also possible.



## Stages of Compilation

MaxCompiler compilation

*VHDL*

Synthesis

*Netlist*

Map

*LUTs*

Place

*Placed FPGA*

Route

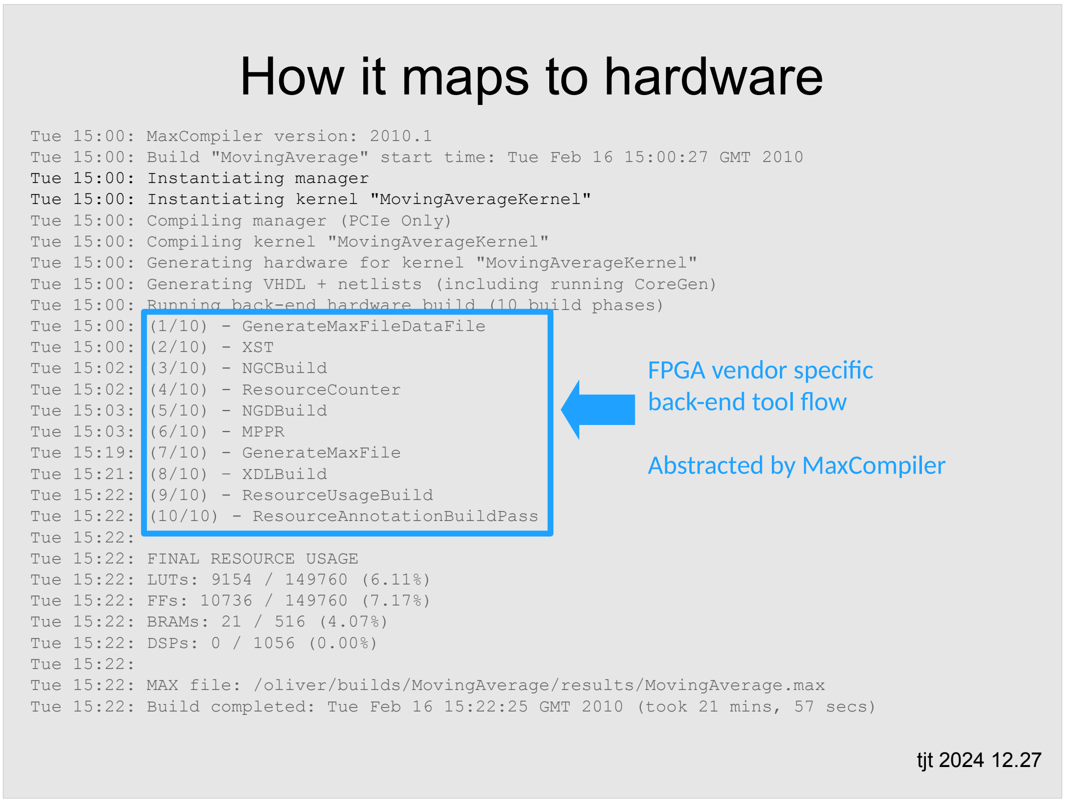
*Complete FPGA*

Generate Maxfile

- *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

tjt 2024 12.26

This shows how the graph is mapped into a circuit (contained in the Maxfile). The whole process can take hours to days for large designs.



## 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:
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:
Tue 15:22: MAX file: /oliver/builds/MovingAverage/results/MovingAverage.max
Tue 15:22: Build completed: Tue Feb 16 15:22:25 GMT 2010 (took 21 mins, 57 secs)
```

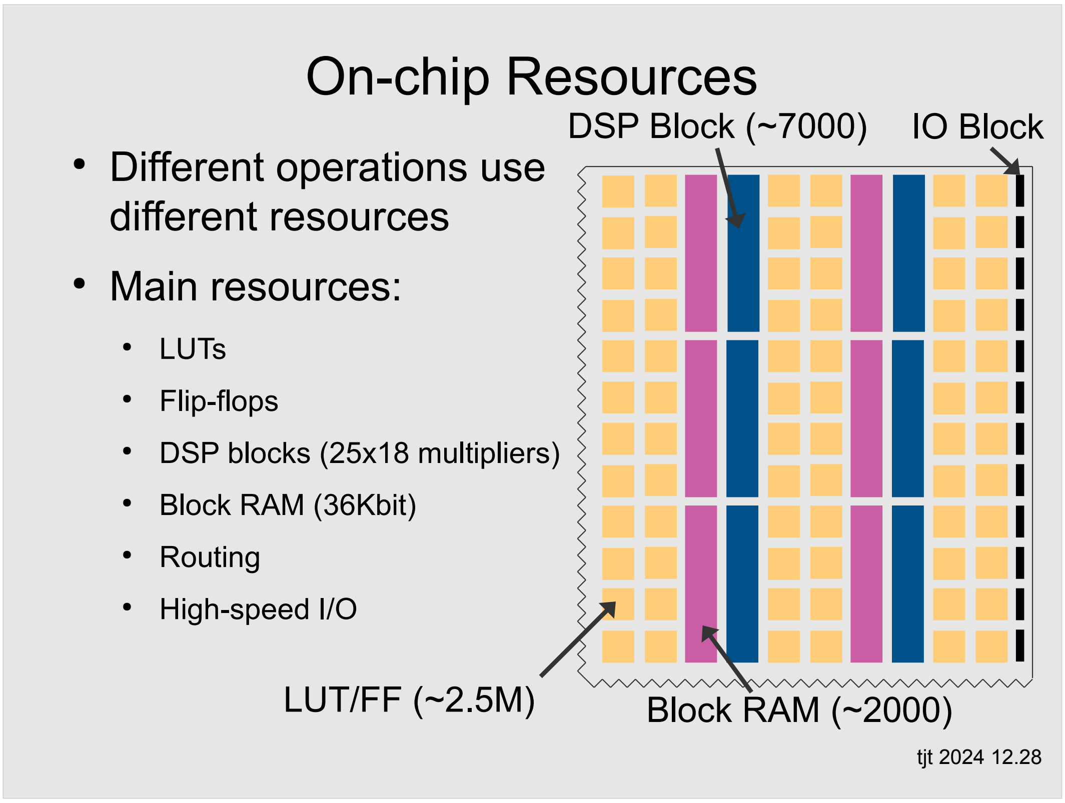
FPGA vendor specific
back-end tool flow

Abstracted by MaxCompiler

tjt 2024 12.27

The Maxeler tools control the process for you. In this example, a small design took about 22 minutes to build into a circuit. Note the final resource usage, in terms of:

- LUTs (look up tables, used for combinational logic);
- FFs (flip-flops, used for registers)
- BRAMs (Block RAMs, small on-chip memories);
- DSPs (digital signal processing units: fast, dedicated multipliers and adders).



## On-chip Resources

- Different operations use different resources
- Main resources:
  - LUTs
  - Flip-flops
  - DSP blocks (25x18 multipliers)
  - Block RAM (36Kbit)
  - Routing
  - High-speed I/O

DSP Block (~7000)

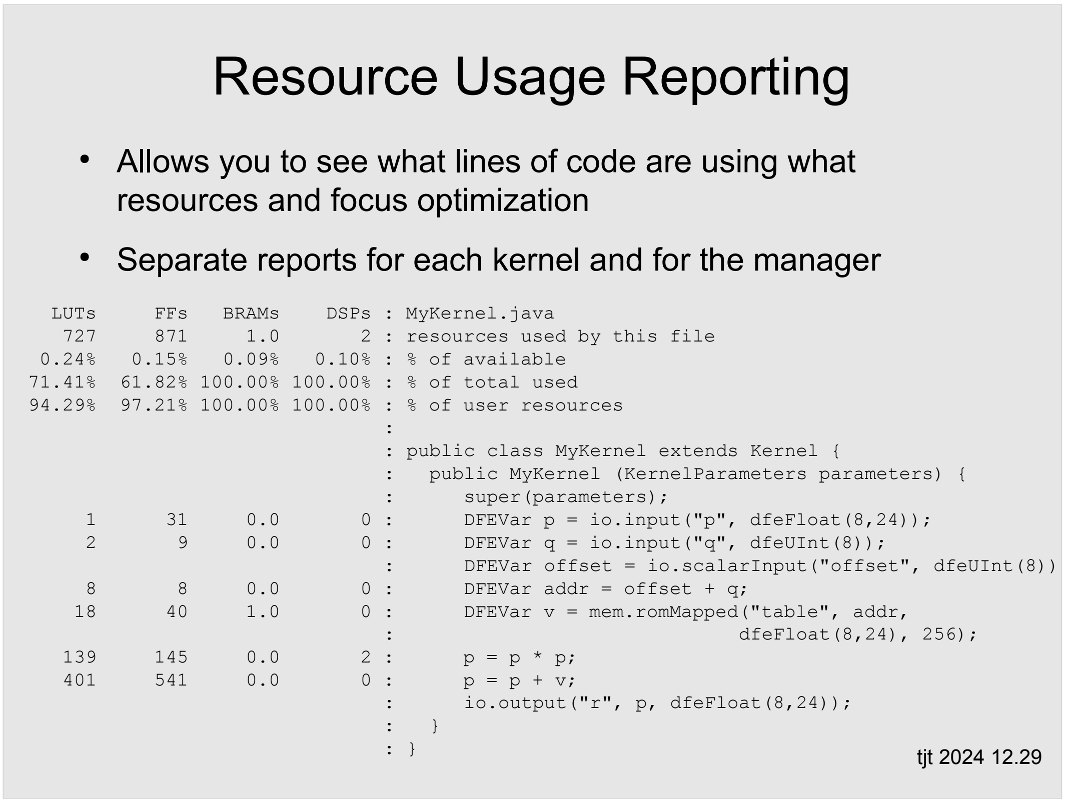
IO Block

LUT/FF (~2.5M)

Block RAM (~2000)

tjt 2024 12.28

The FPGA has multiple different resources.
The numbers on the slide are just an example.
Most of the area tends to be taken by the routing, which is the connections between the different blocks.
LUTs can implement any logical function of 4-6 inputs – the number depends on the FPGA architecture.
DSP elements implement dedicated multiply-add hardware – much faster than implementing the same function using LUTs.
Some recent FPGAs even include dedicated floating-point and neural network hardware.



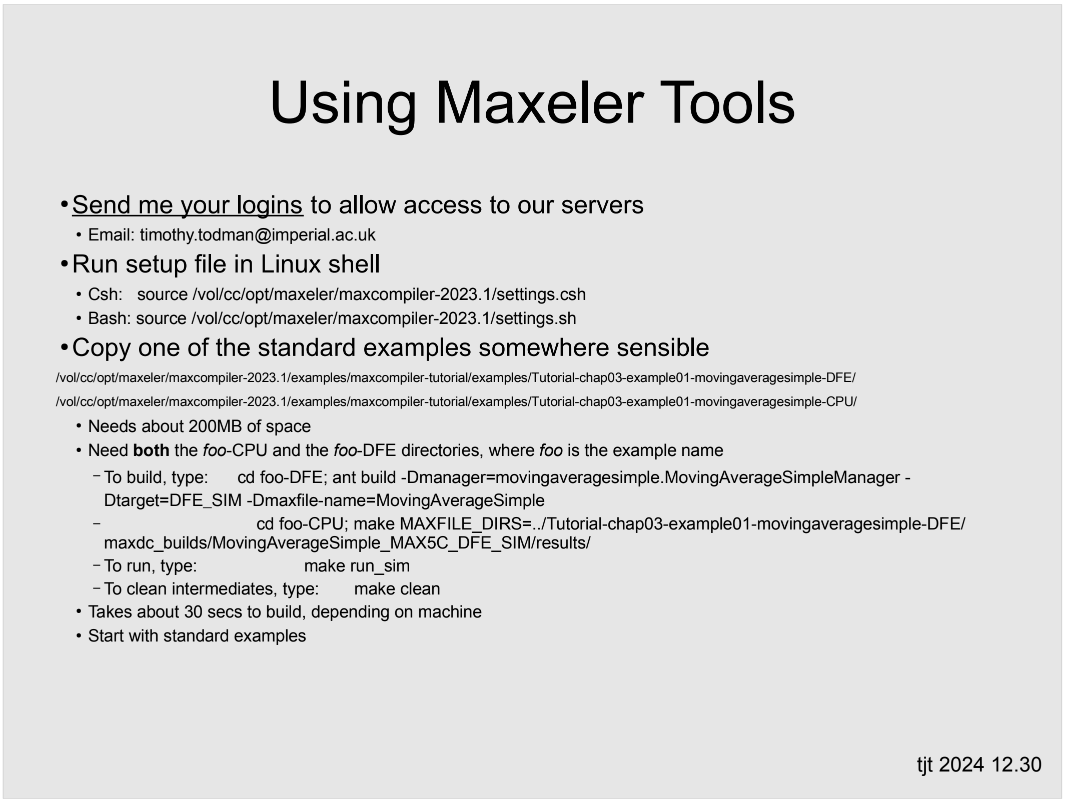
## Resource Usage Reporting

- 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));
                               :   }
                               : }
```

tjt 2024 12.29

You can also show what parts of your design use the on-chip resources.



# Using Maxeler Tools

- Send me your logins to allow access to our servers
  - Email: timothy.todman@imperial.ac.uk
- Run setup file in Linux shell
  - Csh: source /vol/cc/opt/maxeler/maxcompiler-2023.1/settings.csh
  - Bash: source /vol/cc/opt/maxeler/maxcompiler-2023.1/settings.sh
- Copy one of the standard examples somewhere sensible

/vol/cc/opt/maxeler/maxcompiler-2023.1/examples/maxcompiler-tutorial/examples/Tutorial-chap03-example01-movingaveragesimple-DFE/

/vol/cc/opt/maxeler/maxcompiler-2023.1/examples/maxcompiler-tutorial/examples/Tutorial-chap03-example01-movingaveragesimple-CPU/

  - Needs about 200MB of space
  - Need **both** the *foo*-CPU and the *foo*-DFE directories, where *foo* is the example name
    - To build, type: cd foo-DFE; ant build -Dmanager=movingaveragesimple.MovingAverageSimpleManager -Dtarget=DFE_SIM -Dmaxfile-name=MovingAverageSimple
    - cd foo-CPU; make MAXFILE_DIRS=../Tutorial-chap03-example01-movingaveragesimple-DFE/maxdc_builds/MovingAverageSimple_MAX5C_DFE_SIM/results/
    - To run, type: make run_sim
    - To clean intermediates, type: make clean
  - Takes about 30 secs to build, depending on machine
  - Start with standard examples

# Using Maxeler Tools: example

- Run the tools – first build the hardware:

Build command

```
> ant build -Dmanager=movingaveragesimple.MovingAverageSimpleManager -Dtarget=DFE_SIM -Dmaxfile-name=MovingAverageSimple
Buildfile: /vol/cc/tjt97/projects/max/stdexamples/2021.1/maxcompiler-tutorial/examples/Tutorial-chap03-example01-movingaveragesimple-
(snip lots of output)

    [java] Wed 00:42: Running back-end simulation build (3 phases)
    [java] Wed 00:42: (1/3) - Prepare MaxFile Data (GenerateMaxFileDataFile)
    [java] Wed 00:42: (2/3) - Compile Simulation Modules (SimCompilePass)
    [java] Wed 00:42: (3/3) - Generate MaxFile (AddSimObjectToMaxFilePass)
    [java] Wed 00:42: MaxFile: /vol/cc/tjt97/projects/max/stdexamples/2021.1/maxcompiler-tutorial/examples/Tutorial-chap03-example01-m
    [java] Wed 00:42: Build completed: Wed Feb 16 00:42:20 GMT 2022 (took 5 secs)

build:

BUILD SUCCESSFUL
Total time: 6 seconds

> make MAXFILE_DIRS=../Tutorial-chap03-example01-movingaveragesimple-DFE/maxdc_builds/MovingAverageSimple_MAX5C_DFE_S
(snip lots of output)
```

Tool output:
building the hardware

Tool output:
Building the C host code

tjt 2024 12.31

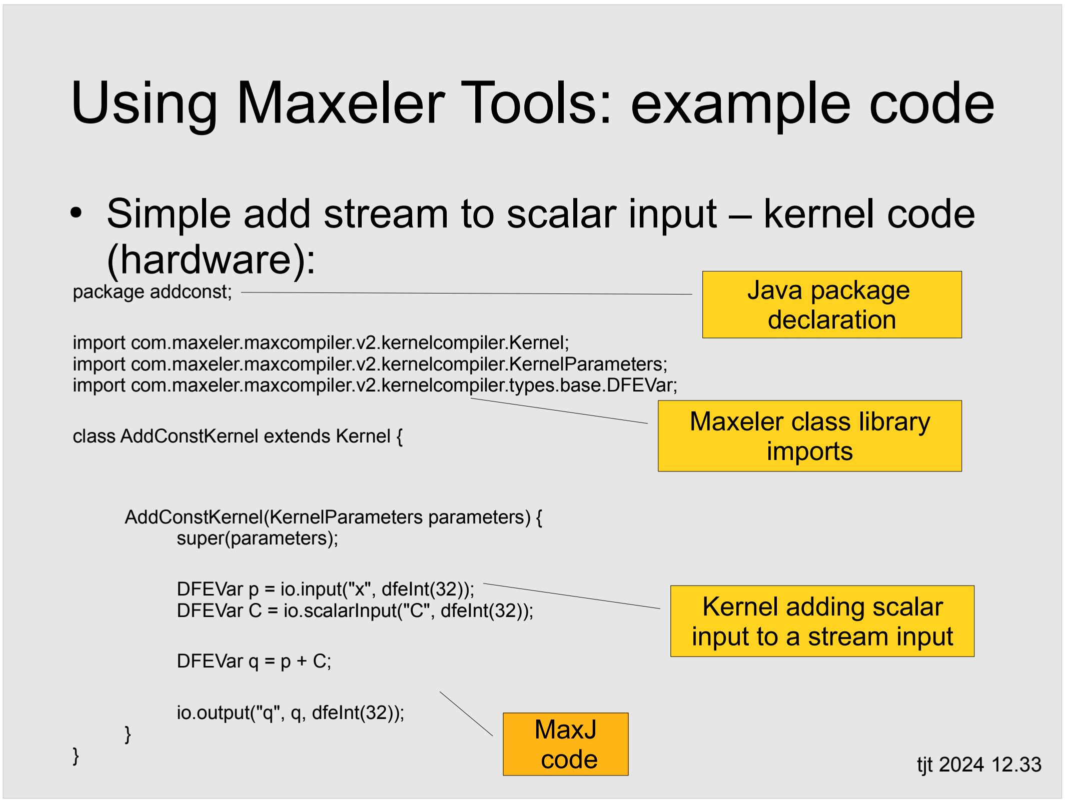
# Using Maxeler Tools: example

- Second, run the hardware: make runSIM

```
> make run_sim
maxcompilersim -n tjt97_movingaveragesimple_sim -c MAX5C restart

Simulated system 'tjt97_movingaveragesimple_sim' started:
    Board:                  MAX5C (default: 48GB RAM)
    RAM size for simulation: 51539607552 bytes.
    Temporary RAM file in    /tmp/. (Use option -k to preserve it.)
    Simulation log:          /homes/tjt97/.maxcompilersim/tjt97_movingaveragesimple_sim-cccad5.log
    Daemon log:              /homes/tjt97/.maxcompilersim/tjt97_movingaveragesimple_sim-cccad5_daemon.log
(snip some output)
Running dist/release/bin/movingaveragesimple

Running DFE
dataOut[1] = 1.000000
dataOut[2] = 0.666667
dataOut[3] = 2.000000
dataOut[4] = 1.666667
dataOut[5] = 4.333333
dataOut[6] = 4.000000

dist/release/bin/movingaveragesimple Terminated successfully.
```

Run command

Output from
your host program



# Using Maxeler Tools: example code

- Simple add stream to scalar input – kernel code (hardware):

```
package addconst;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class AddConstKernel extends Kernel {


        AddConstKernel(KernelParameters parameters) {
                super(parameters);

                DFEVar p = io.input("x", dfeInt(32));
                DFEVar C = io.scalarInput("C", dfeInt(32));

                DFEVar q = p + C;

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

Java package declaration

Maxeler class library imports

Kernel adding scalar input to a stream input

MaxJ code

# Using Maxeler Tools: example code

- Simple add stream to scalar input – host C code (software):

```
// CPU code adding scalar to stream input
// Max includes
#include "Maxfiles.h"                    // Includes .max files
#include <MaxSLiCInterface.h>            // Simple Live CPU interface
// no. elems
#define N       8
// constant to add
#define C       2

int dataIn[N];
int dataOut[N];

int main()
{
        // init data
        for (int i = 0; i < N; i++)
        {
                dataIn[i] = i;
                dataOut[i] = -1;
        }
        // run
        printf("Running DFE\n");
```

Include Maxeler headers

Define constants

C code

Initialise data for streams

(continues on next slide)

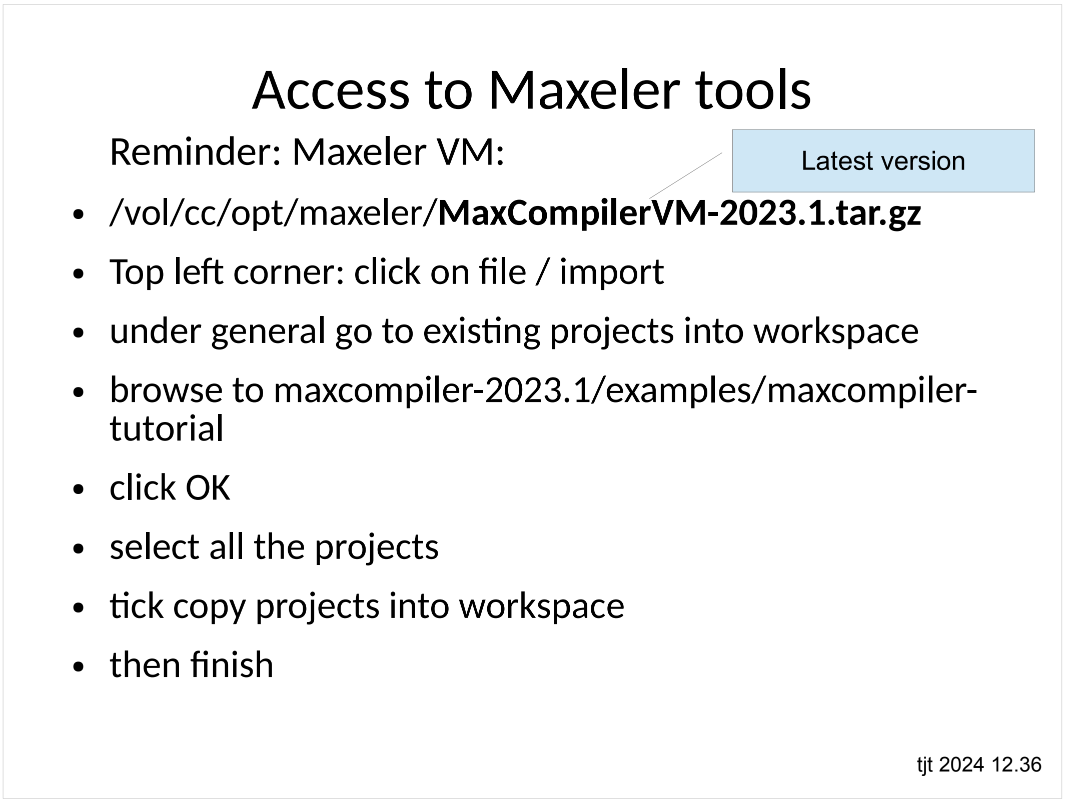
# Using Maxeler Tools: example code

- Host code (software) continued:

```
        // call the function generated by Maxeler tools to run the kernel
        // on the inputs, getting the outputs in response:
        AddConst(
                        // number of stream elements:
                        N,
                        // scalar input:
                        C,
                        // stream input:
                        dataIn,
                        // stream output:
                        dataOut);

        // print the inputs and outputs
        for (int i = 0; i < N; i++)
        {
                printf("%d: %d", i, dataIn[i]);
                printf(" -> %d\n", dataOut[i]);
        }

        printf("ALL OK!\n");

        return 0;
}
```

Call function generated by Maxeler tools from the kernel.
Each time you call this function, it runs the hardware (or simulator) using the given arrays as input or output streams.

Print input and output streams



# Access to Maxeler tools

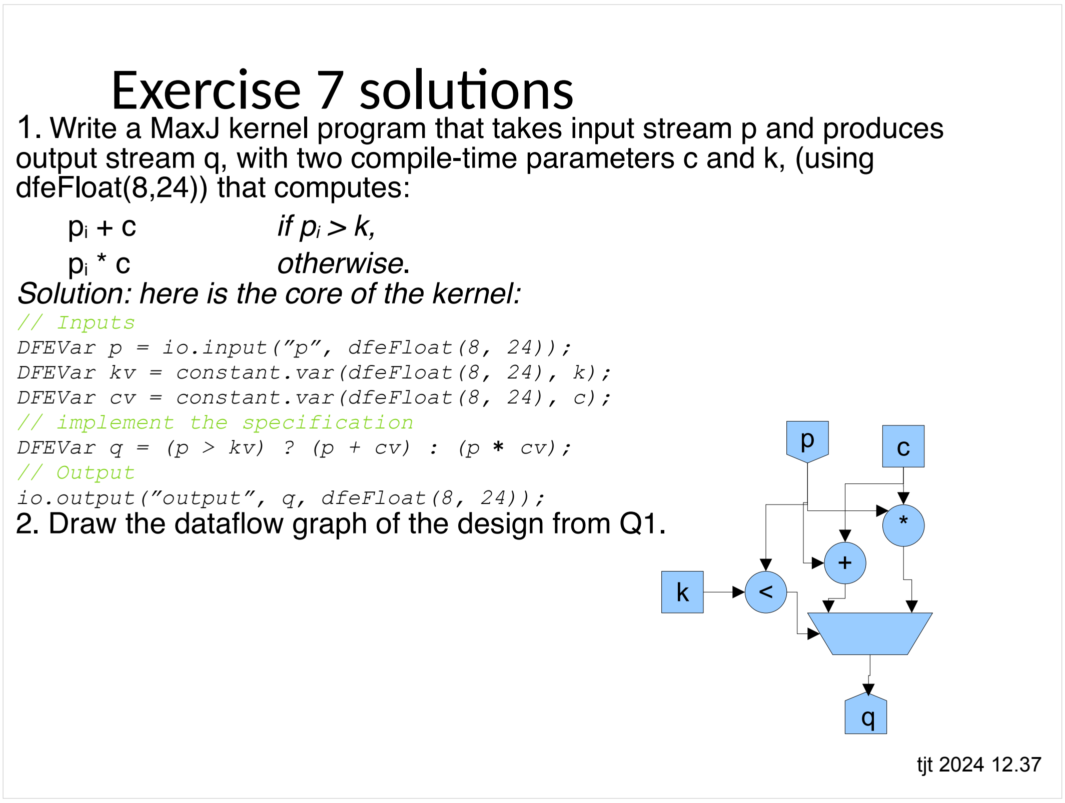
Reminder: Maxeler VM:

Latest version

- /vol/cc/opt/maxeler/**MaxCompilerVM-2023.1.tar.gz**
- Top left corner: click on file / import
- under general go to existing projects into workspace
- browse to maxcompiler-2023.1/examples/maxcompiler-tutorial
- click OK
- select all the projects
- tick copy projects into workspace
- then finish

tjt 2024 12.36

Note the compiler version.



## Exercise 7 solutions

1. Write a MaxJ kernel program that takes input stream p and produces output stream q, with two compile-time parameters c and k, (using dfeFloat(8,24)) that computes:

$p_i + c$ *if $p_i > k$,*

$p_i * c$ *otherwise.*

*Solution: here is the core of the kernel:*

```
// Inputs
DFEVar p = io.input(”p”, dfeFloat(8, 24));
DFEVar kv = constant.var(dfeFloat(8, 24), k);
DFEVar cv = constant.var(dfeFloat(8, 24), c);
// implement the specification
DFEVar q = (p > kv) ? (p + cv) : (p * cv);
// Output
io.output(”output”, q, dfeFloat(8, 24));
```

2. Draw the dataflow graph of the design from Q1.

tjt 2024 12.37

Exercise: simple MaxJ kernels.

## Exercise 7 Solutions

3. Write a MaxJ kernel program that takes three input streams, x, y and z (all of type dfeInt(32)), and computes an output stream p such that:

$p_i = (x_i + y_i) + 2$ *if $z_i > x_i$*

$(x_i + z_i) * 2$ *if $z_i < x_i$*

$(x_i * y_i * z_i)$ *otherwise*

*Solution:*

```
// Input
DFEVar x = io.input("x", dfeInt(32));
DFEVar y = io.input("y", dfeInt(32));
DFEVar z = io.input("z", dfeInt(32));
// implement the specification
// Note we must use conditional expressions
DFEVar output =
     (z > x)
       ? ((x + y) + 2)
       : ((z < x)
         ? ((x + z) * 2)
         : (x * y * z))
         ;
// Output
io.output("output", output, dfeInt(32));
```

# Exercise 8

Draw the dataflow graph generated by the following MaxJ kernel program:

```
for (int i = 0; i < 6; i++) {
  DFEVar x = io.input("x"+i, dfeInt(32));
  DFEVar y = x;
  if (i % 3 != 0) {
    for (int j = 0; j < 3; j++) {
      y = y + x * j;
    }
  } else {
    y = y * y;
  }
  io.output("y"+i, y, dfeInt(32));
}
```