Multiscale Dataflow Computing

(Building vertically in a horizontal world)

Oliver Pell

MuCoCoS 2013

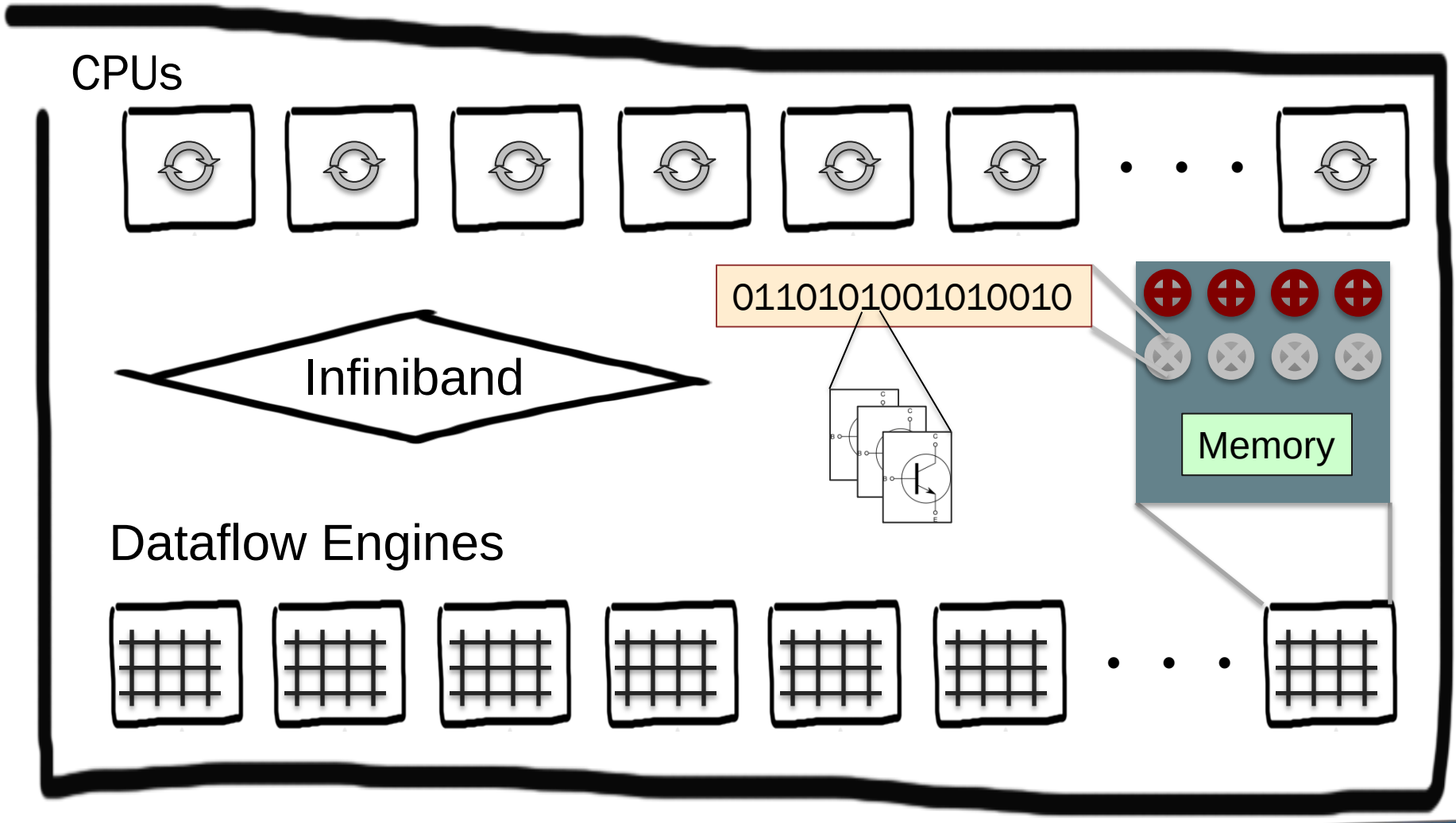
Multiscale dataflow computing

Definiton: “Multiscale”

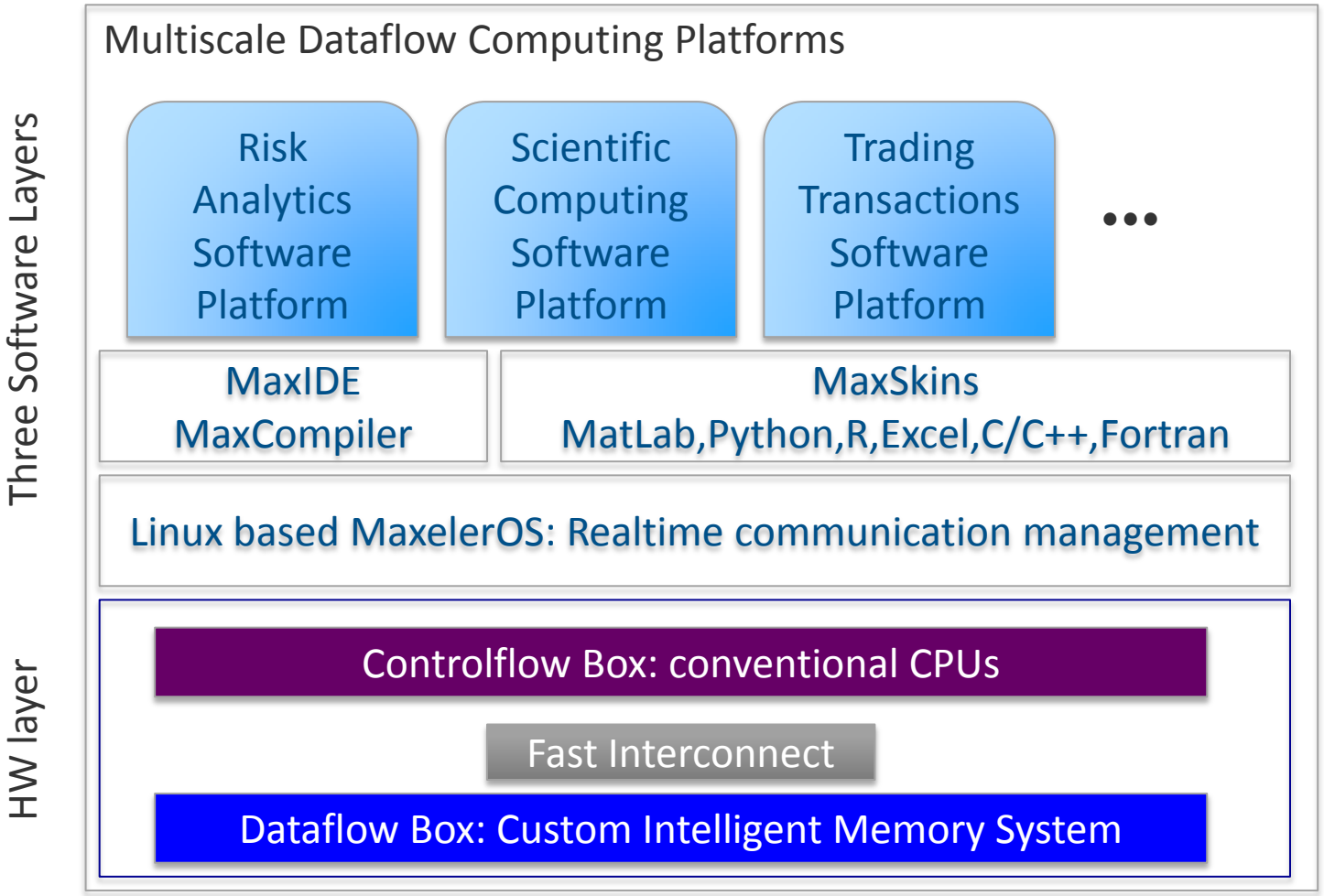
Problems which have important features at multiple scales

Multiple scales of computing	Important features for optimization
complete system level	⇒ balance compute, storage and IO
parallel node level	⇒ maximize utilization of compute and interconnect
microarchitecture level	⇒ minimize data movement
arithmetic level	⇒ tradeoff range, precision and accuracy = discretize in time, space and value
bit level	⇒ encode and add redundancy
transistor level	=> create the illusion of ‘0’ and ‘1’

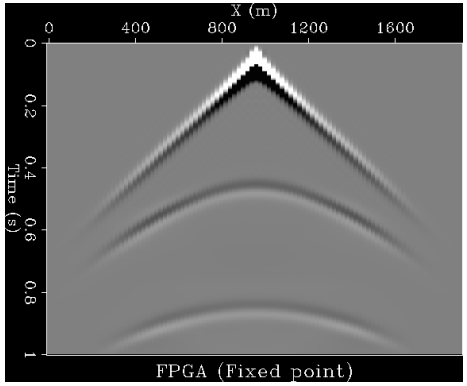
A heterogeneous system



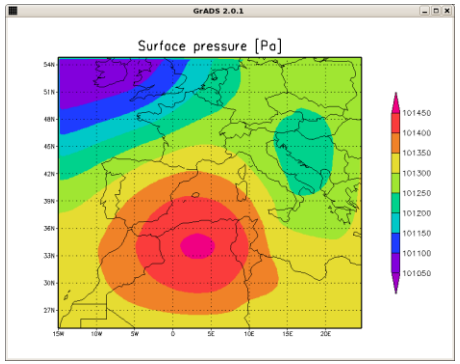
The Multiscale Dataflow Computer



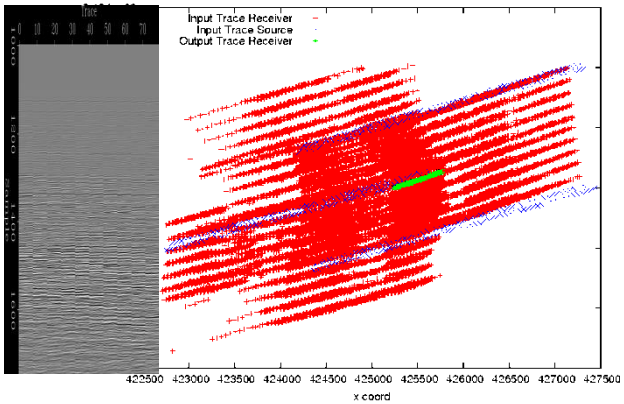
Multiscale Dataflow Advantage



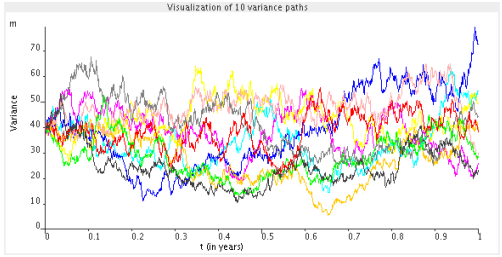
Acoustic Modelling 25x



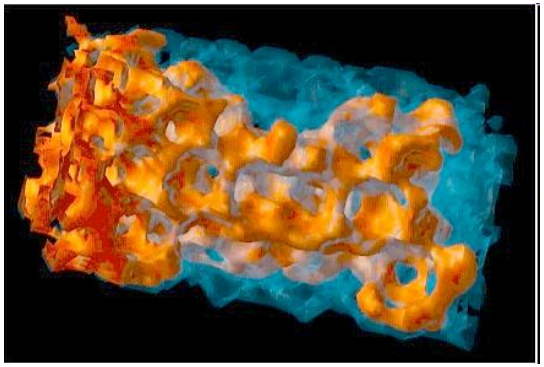
Weather Modeling 60x



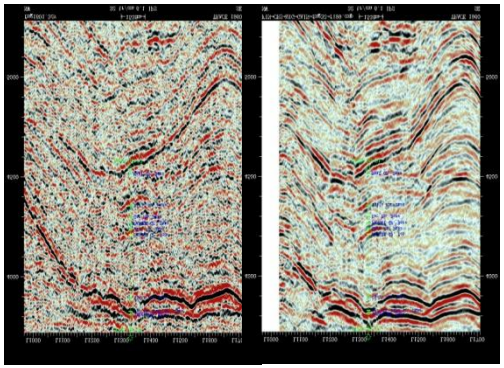
Trace Processing 22x



Financial Risk 32x



Fluid Flow 30x



Seismic Imaging 29x

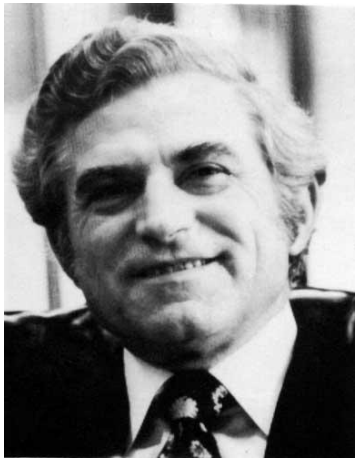
MAXELER Technologies

Amdahl's Laws

- First law: Serial processors always win; too much time spent in programming the parallel processor.
- Second law: fraction of serial code, s, limits speedup to:

  $Sp = T1 / (T1 (s) + T1 (1-s)/p)$ or

  $Sp = 1 / (s + (1-s)/p)$



Gene Amdahl

Slotnick's law (of effort)

"The parallel approach to computing does require that some original thinking be done about numerical analysis and data management in order to secure efficient use.

In an environment which has represented the absence of the need to think as the highest virtue this is a decided disadvantage."

-Daniel Slotnick

The Vertical Perspective

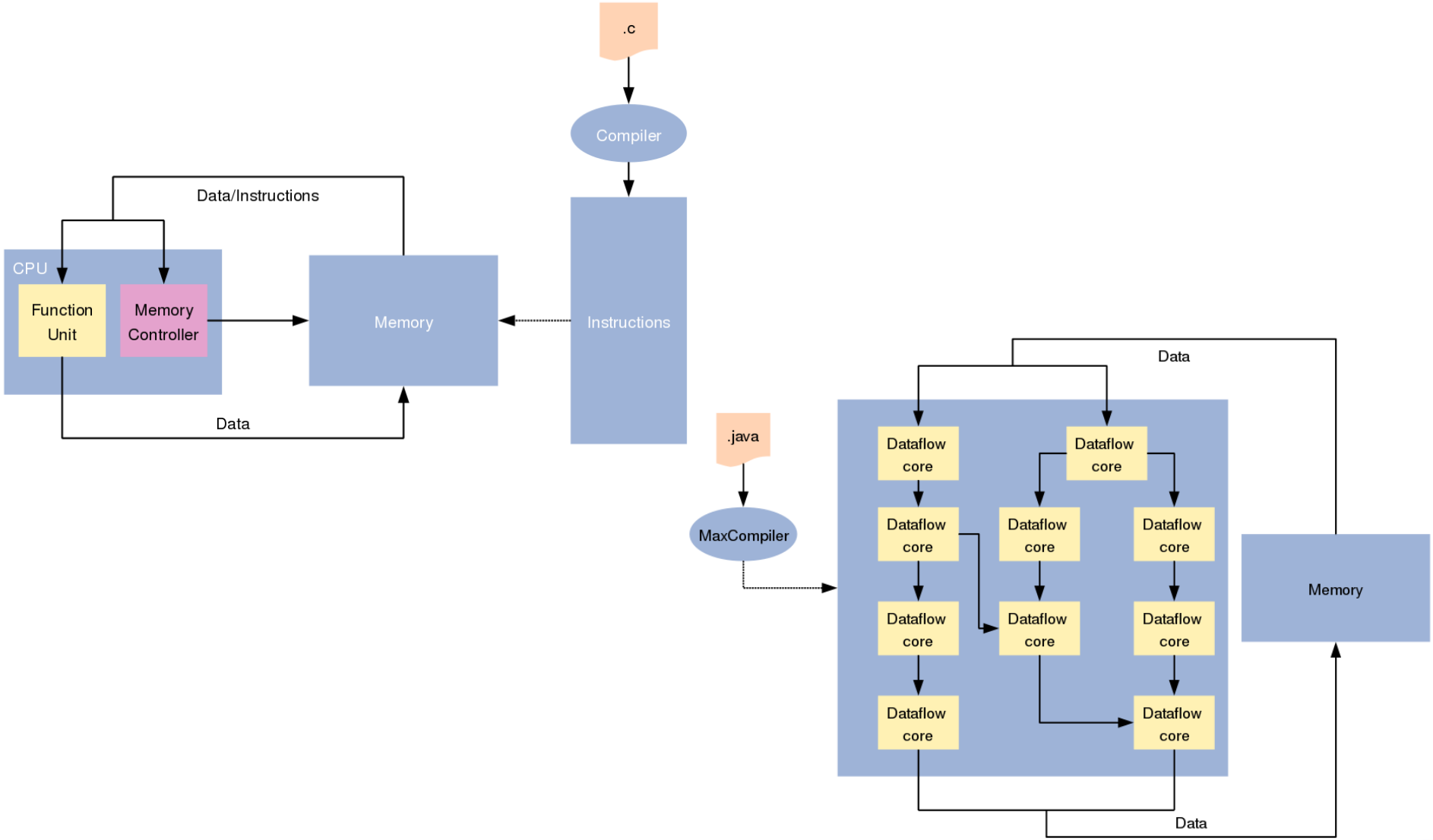
Dataflow Application Areas
Finance, Geophysics, Chemistry
Physics, Genetics, Astronomy

Application Programming Interface
MaxCompiler: Dataflow in Space and Time
Heterogeneous dataflow+controlflow optimization

Transactions Management
MaxelerOS manages dataflow transaction
MaxelerOS keeps Dataflow-Controlflow balance

Architecture: Static Dataflow
Static Dataflow microarchitecture, cards, boxes
Predictable execution time and efficiency

Control flow vs. Dataflow

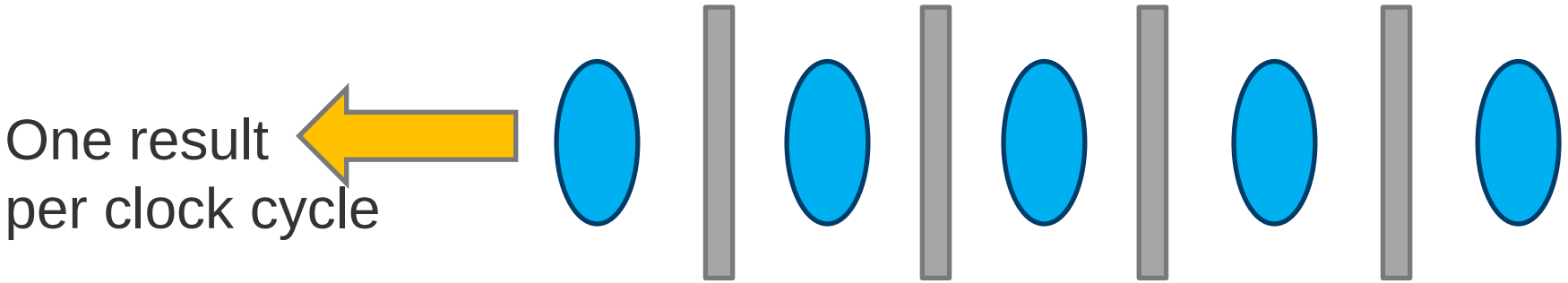


MAXELER Technologies

Static Dataflow

"Systolic Arrays" without nearest neighbour interconnect restrictions

Static ultradeep (>1000 stage) computing pipelines



~~The power challenge~~
The data movement challenge

	Today	2018-20
Double precision FLOP	100pj	10pj
Moving data on-chip: 1mm	6pj	
Moving data on-chip: 20mm	120pj	
Moving data to off-chip memory	5000pj	2000pj

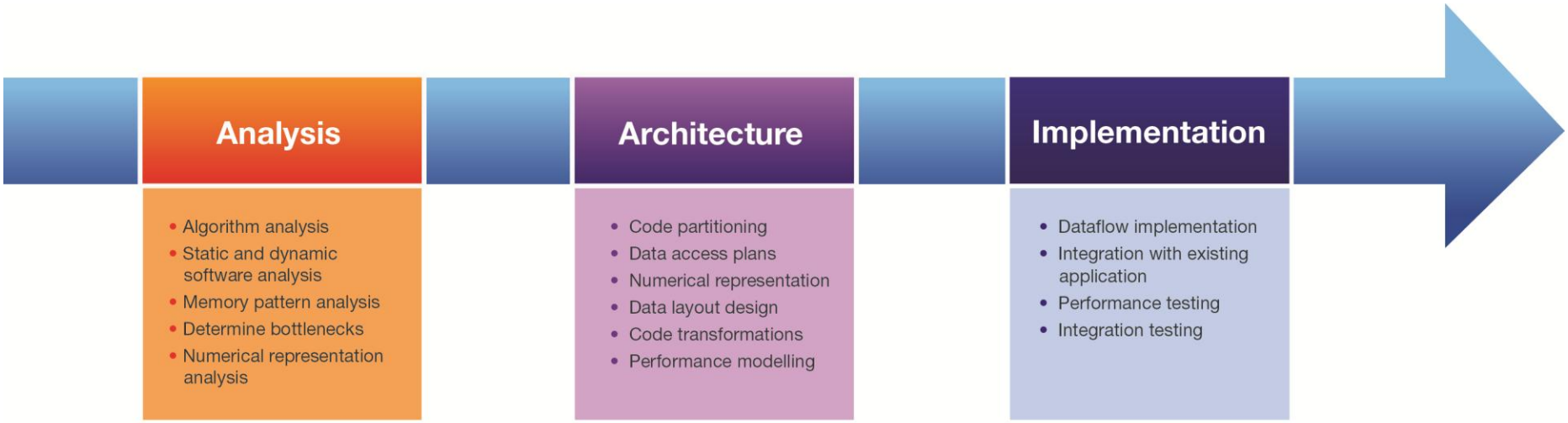
- Moving data on-chip will use as much energy as computing with it
- Moving data off-chip will use **200x more energy**!
  - And is much slower as well

The memory hierarchy (challenge)

John von Neumann, 1946:

“We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding, but which is less quickly accessible.”

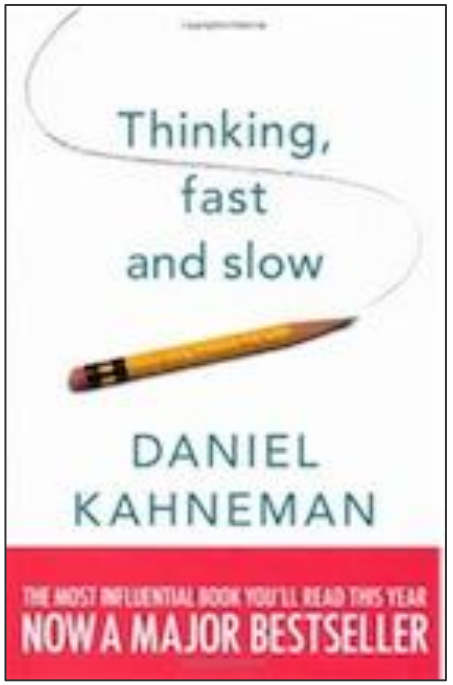
Vertical Co-design of applications



- Deploy domain expertise to **co-design** application, algorithm and computer architecture

$$17 \times 24 = ?$$

Thinking Fast and Slow



Daniel Kahneman

Nobel Prize in Economics, 2002

back to 17 × 24

Kahneman splits thinking into:

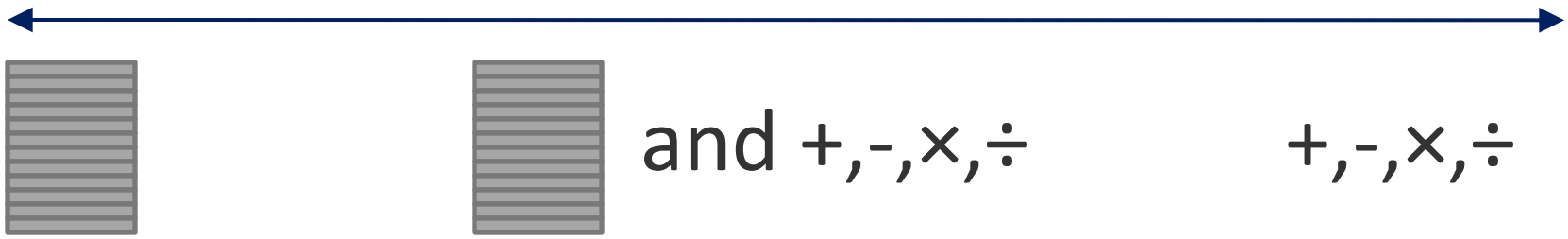
System 1: fast, hard to control ... 400

System 2: slow, easier to control ... 408

Putting it all together on the arithmetic level

Computing f(x) in the range [a,b] with $|E| \leq 2^{-n}$

Table *Table+Arithmetic* *Arithmetic*



Tradeoff: number of coefficients, number of bits per coefficient, range versus precision of result and maximal versus average error of result

Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk
Optimizing Hardware Function Evaluation
IEEE Transactions on Computers. vol. 54, no. 12, pp. 1520-1531. Dec, 2005.

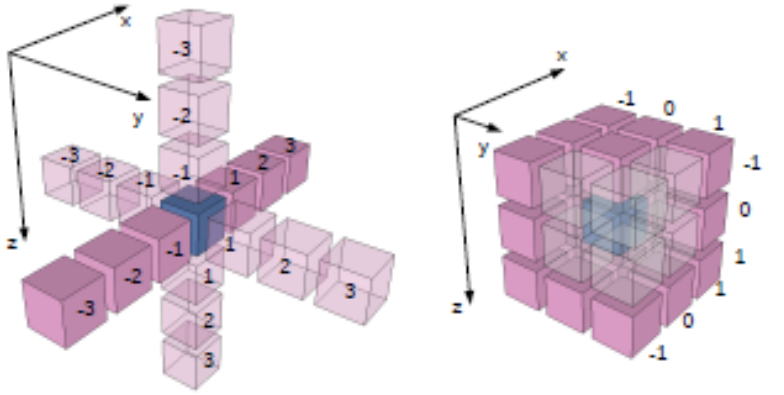
Given range and precision for the result, what is the optimal table+arithmetic solution?

Minimal Latency (Optimized for Latency)

Range [bits] \ Precision [bits]	4	8	12	16	20	24
24	sin: *tp2*, 24, 78 log: *po2*, 8, 30 sqr: *tp2*, 18, 912	sin: *tp2*, 28, 348 log: *tp2*, 12, 78 sqr: *tp2*, 24, 2400	sin: *tp2*, 32, 792 log: *tp2*, 15, 384 sqr: *tp2*, 26,10368	sin: *tp2*, 32, 3168 log: *tp2*, 21, 1056 sqr: *tp2*, 30,23808	sin: *tp2*, 40, 7872 log: *tp2*, 24, 4800 sqr: *tp3*, 34,17920	sin: *tp3*, 42, 5504 log: *tp2*, 28,11136 sqr: *tp4*, 39,12800
20	sin: *po2*, 19, 61 log: *po2*, 7, 27 sqr: *tp2*, 18, 456	sin: *tp2*, 24, 300 log: *tp2*, 12, 78 sqr: *tp2*, 20, 2016	sin: *tp2*, 28, 696 log: *tp2*, 15, 384 sqr: *tp2*, 24, 4800	sin: *tp2*, 32, 3168 log: *tp2*, 21, 1056 sqr: *tp2*, 32,12288	sin: *tp2*, 32, 6336 log: *tp2*, 24, 4800 sqr: *tp3*, 33, 8704	sin: *tp3*, 38, 4992 log: *tp2*, 27,10752 sqr: *tp3*, 37,19456
16	sin: *tp2*, 16, 54 log: *po2*, 7, 27 sqr: *tp2*, 17, 324	sin: *tp2*, 19, 240 log: *tp2*, 12, 78 sqr: *tp2*, 18, 912	sin: *tp2*, 24, 600 log: *tp2*, 15, 384 sqr: *tp2*, 24, 2400	sin: *tp2*, 28, 2784 log: *tp2*, 21, 1056 sqr: *tp2*, 26,10368	sin: *tp2*, 32, 6336 log: *tp2*, 24, 4800 sqr: *tp2*, 30,23808	sin: *tp3*, 32, 4224 log: *tp2*, 27,10752 sqr: *tp3*, 34,17920
12	sin: *po2*, 9, 31 log: *po2*, 7, 27 sqr: *tp2*, 12, 156	sin: *tp2*, 16, 204 log: *tp2*, 12, 78 sqr: *tp2*, 18, 456	sin: *tp2*, 20, 504 log: *tp2*, 15, 384 sqr: *tp2*, 20, 2016	sin: *tp2*, 24, 2400 log: *tp2*, 21, 1056 sqr: *tp2*, 24, 4800	sin: *tp2*, 28, 5568 log: *tp2*, 24, 4800 sqr: *tp2*, 31,12288	sin: *tp2*, 32,12672 log: *tp2*, 27,10752 sqr: *tp3*, 33, 8704
8	sin: *po2*, 6, 22 log: *po2*, 7, 27 sqr: *tp2*, 7, 27	sin: *tp2*, 10, 132 log: *tp2*, 11, 72 sqr: *tp2*, 17, 324	sin: *tp2*, 16, 408 log: *tp2*, 15, 384 sqr: *tp2*, 18, 912	sin: *tp2*, 19, 1920 log: *tp2*, 21, 1056 sqr: *tp2*, 24, 2400	sin: *tp2*, 24, 4800 log: *tp2*, 24, 4800 sqr: *tp2*, 26,10368	sin: *tp2*, 28,11136 log: *tp2*, 27,10752 sqr: *tp2*, 30,23808
4	sin: *po2*, 6, 22 log: *po2*, 7, 27 sqr: *po2*, 7, 25	sin: *tp2*, 10, 132 log: *tp2*, 11, 72 sqr: *tp2*, 12, 156	sin: *tp2*, 15, 384 log: *tp2*, 15, 384 sqr: *tp2*, 18, 456	sin: *tp2*, 19, 1920 log: *tp2*, 17, 1056 sqr: *tp2*, 20, 2016	sin: *tp2*, 23, 4608 log: *tp2*, 24, 4800 sqr: *tp2*, 24, 4800	sin: *tp2*, 28,11136 log: *tp2*, 27,10752 sqr: *tp2*, 31,12288

Architecture Level: Star versus Cube Stencil

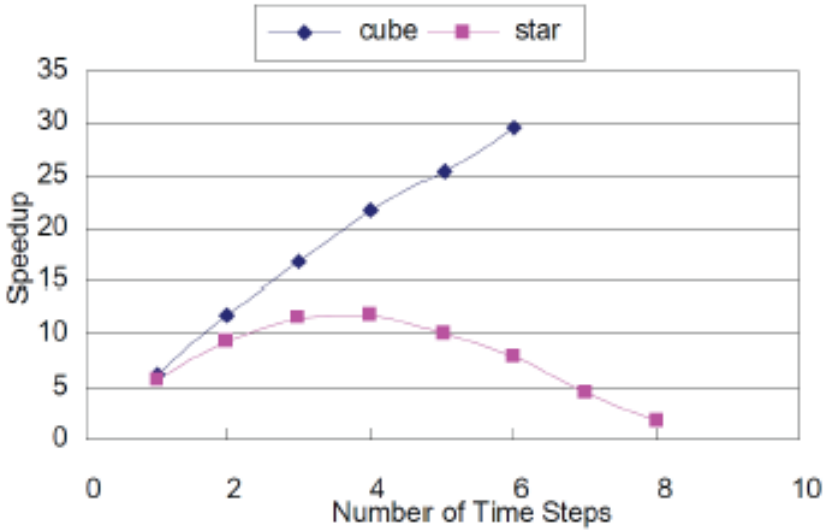
More Computation in Less Time?



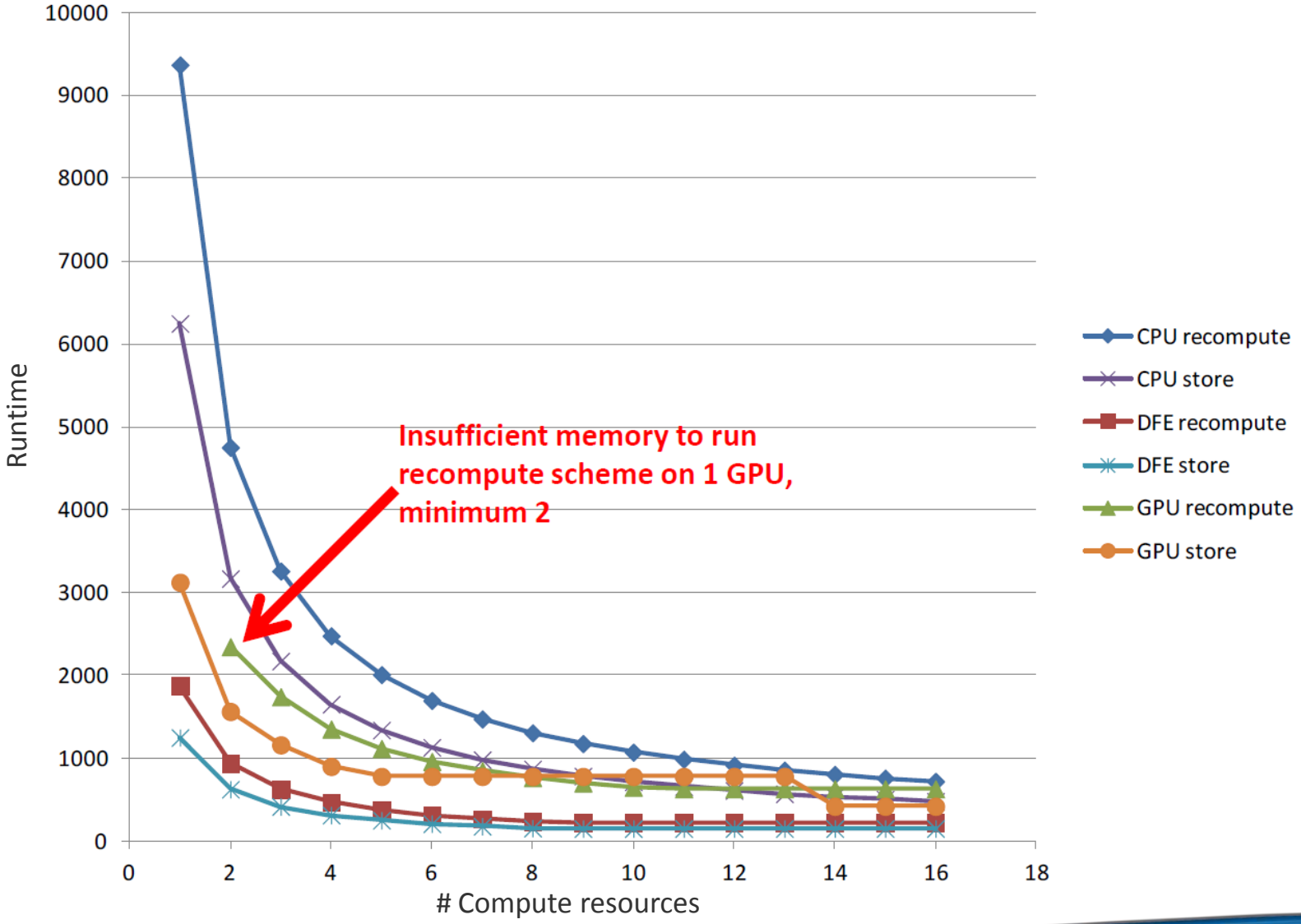
(a) 7-point star stencil (b) 3x3x3 cube stencil

Figure 1: 2 Alternative stencil choices.

Local temporal parallelism
=> Cascading timesteps

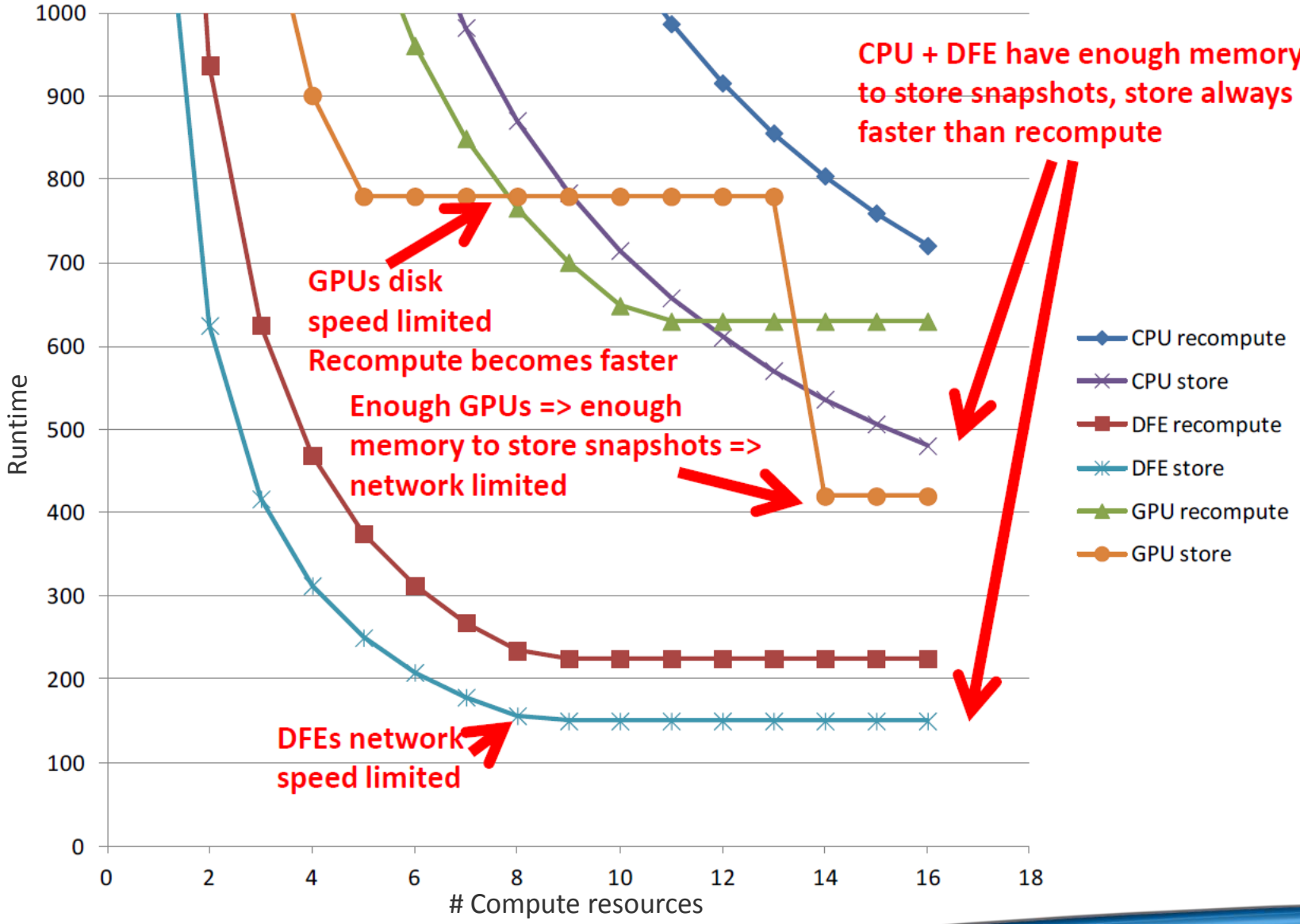


System level: algorithm vs. resource



MAXELER Technologies

System level: algorithm vs. resource



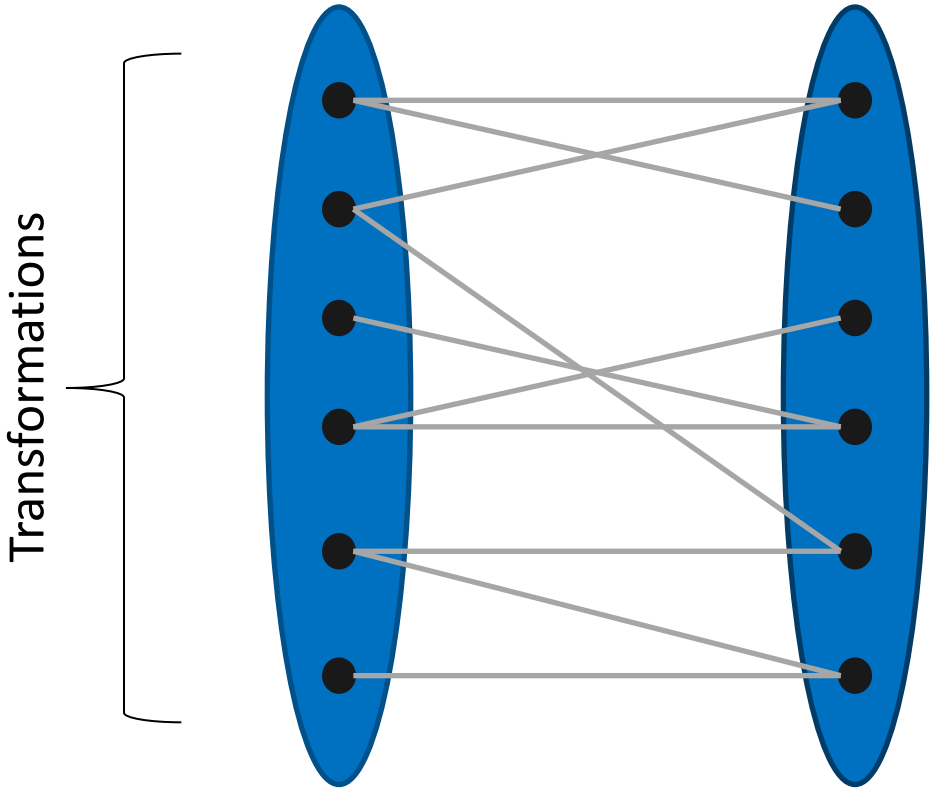
MAXELER Technologies

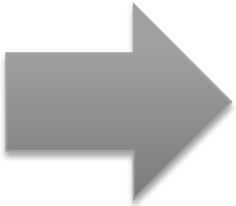
Identify and classify options

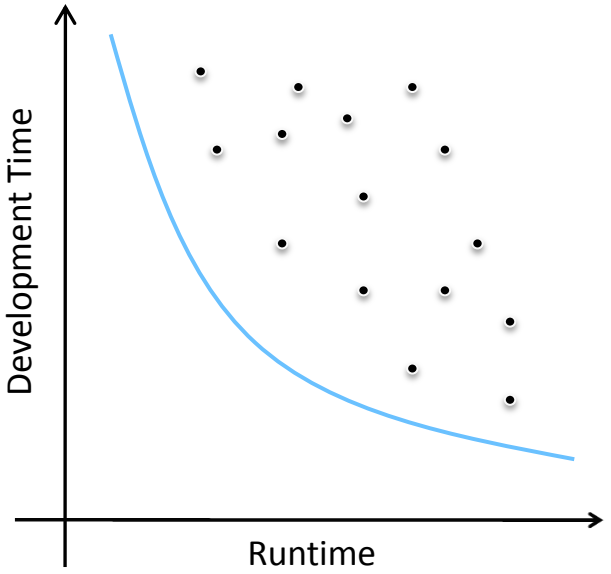
Data Access Plans

Code Partitioning

Pareto Optimal Options

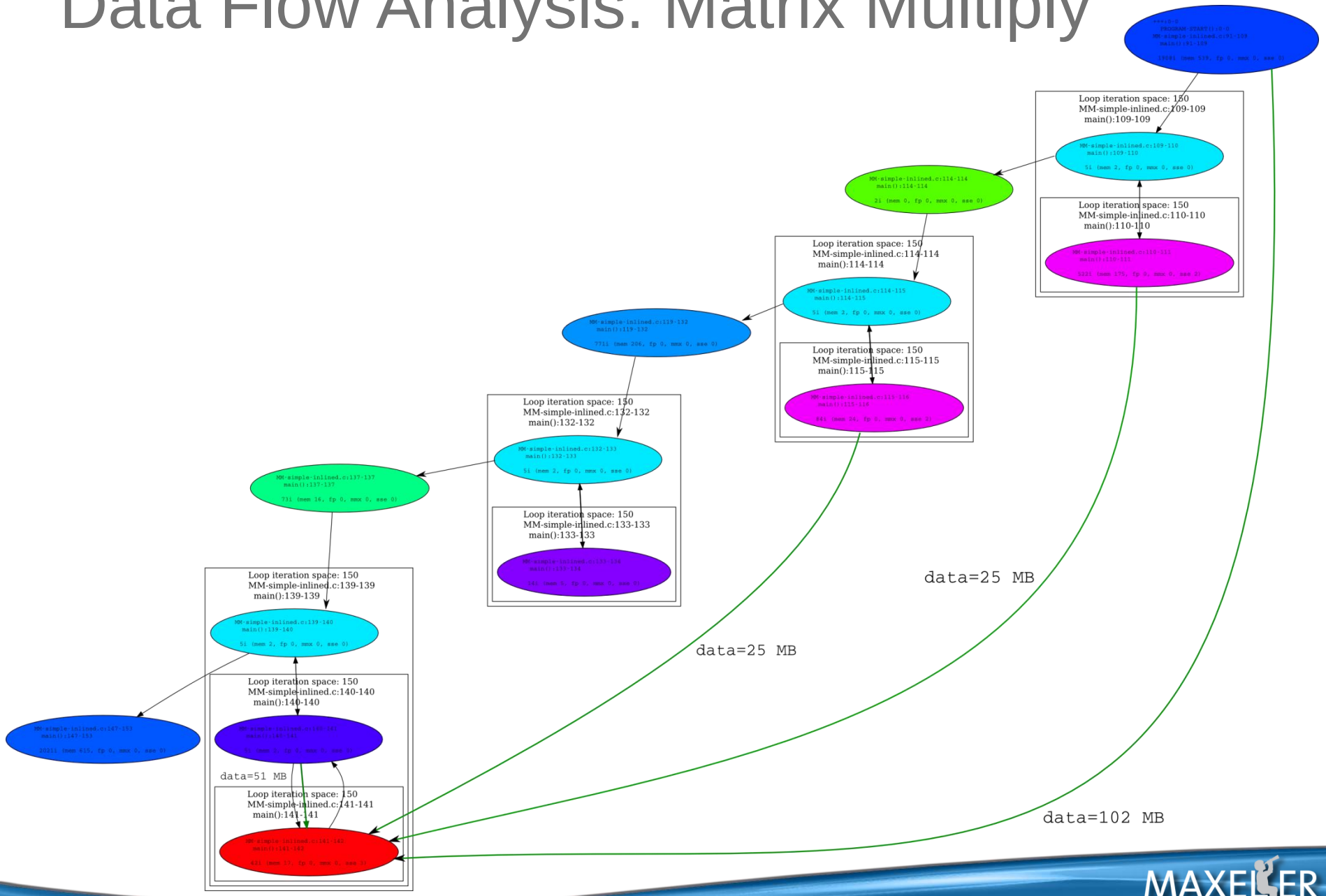






Try to minimise runtime and development time, while maximising flexibility and precision.

Data Flow Analysis: Matrix Multiply



MAXELER Technologies

Maxeler Dataflow Computers

MPC C Series

CPUs plus DFEs
Intel Xeon CPU cores and up to 4 DFEs with 192GB of RAM

MPC X Series

DFEs shared over Infiniband
Up to 8 DFEs with 384GB of RAM and dynamic allocation of DFEs to CPU servers

MPC N Series

Low latency connectivity
Intel Xeon CPUs and 1-4 DFEs with up to twelve 40Gbit Ethernet connections

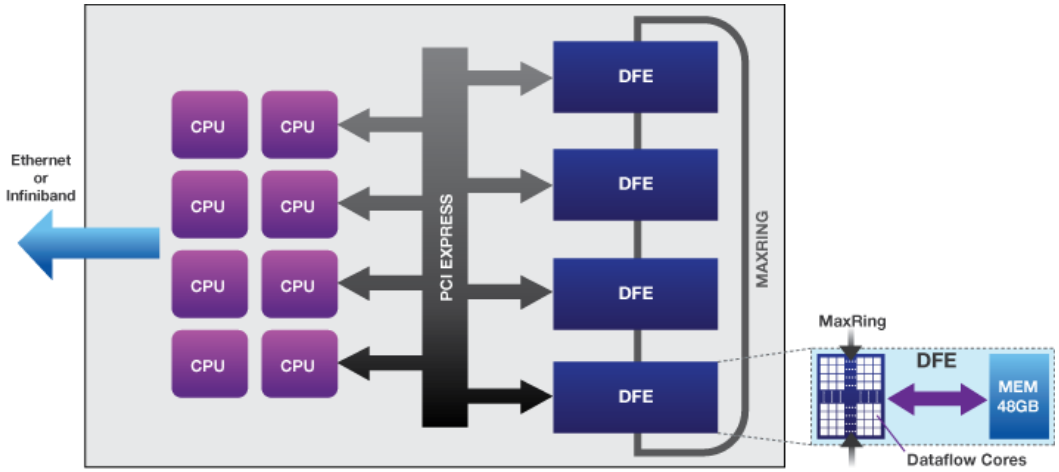
MaxWorkstation
Desktop development system

MaxCloud
On-demand scalable accelerated compute resource, hosted in London

MAXELER Technologies

MPC-C500

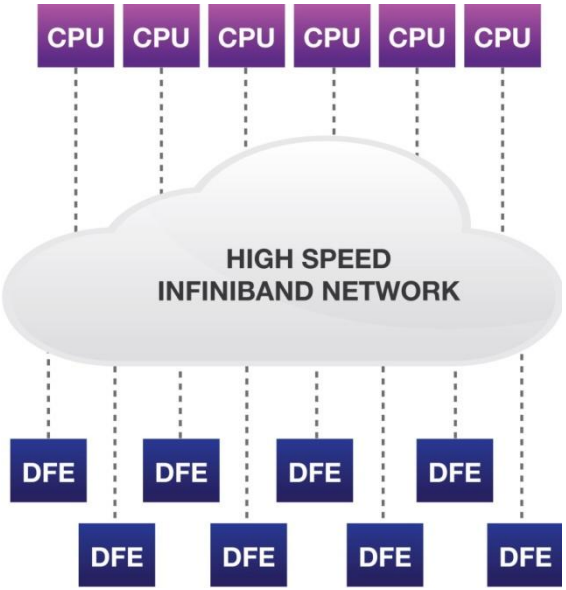
- 1U Form Factor
- 4x dataflow engines
- 12 Intel Xeon cores
- 96GB DFE RAM
- Up to 192GB CPU RAM
- *MaxRing* interconnect
- 3x 3.5” hard drives
- Infiniband/10GigE

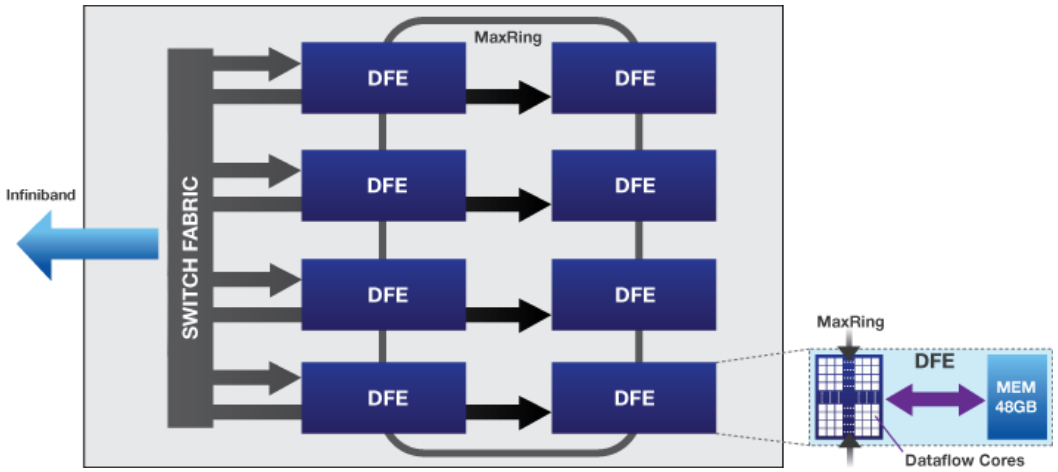


MAXELER Technologies

MPC-X1000

- 8 dataflow engines (384GB RAM)
- High-speed MaxRing
- Zero-copy RDMA between CPUs and DFEs over Infiniband
- Dynamic CPU/DFE balancing

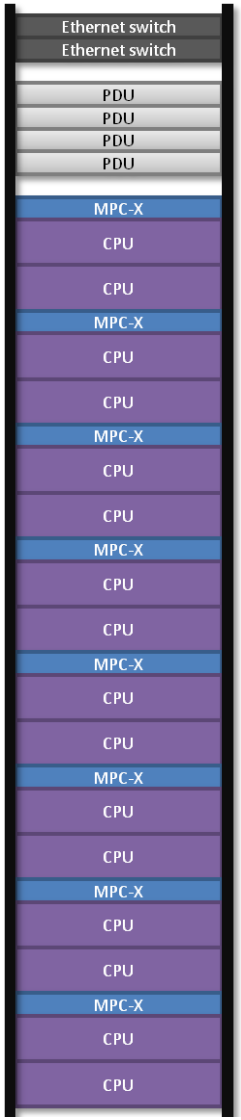




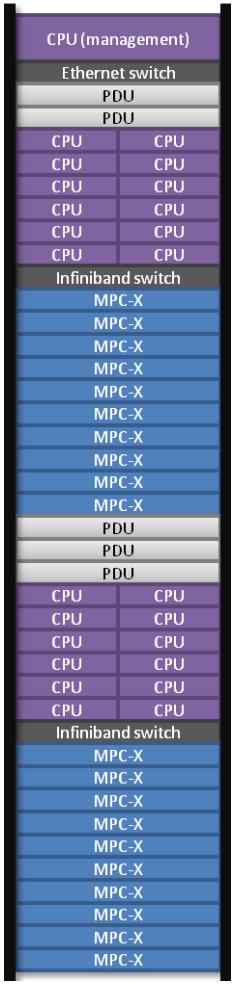
MAXELER Technologies

Dataflow clusters

- Optimized to balance resources for particular application challenges
- Flexible at design-time and at run-time



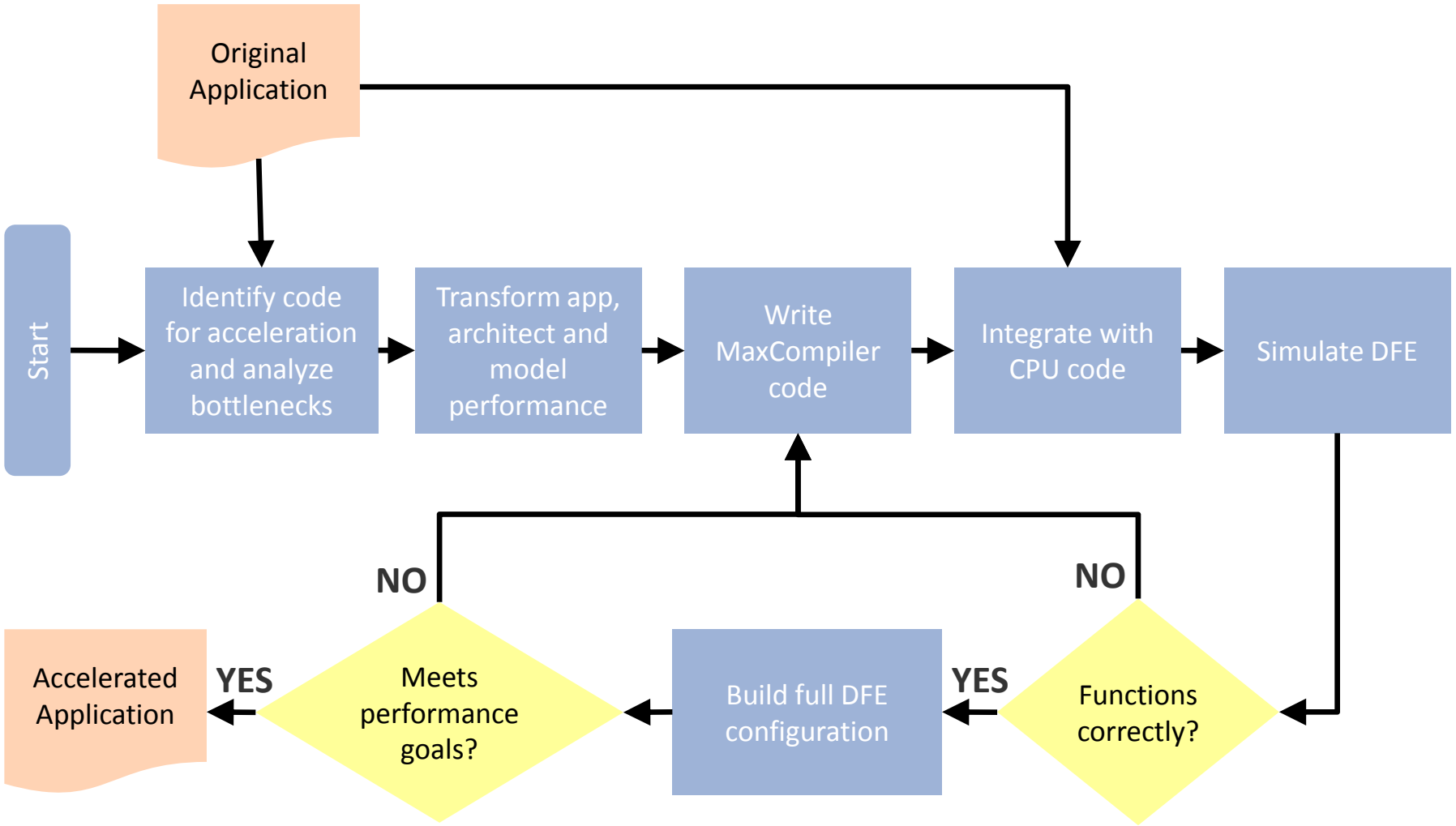
48U seismic imaging cluster



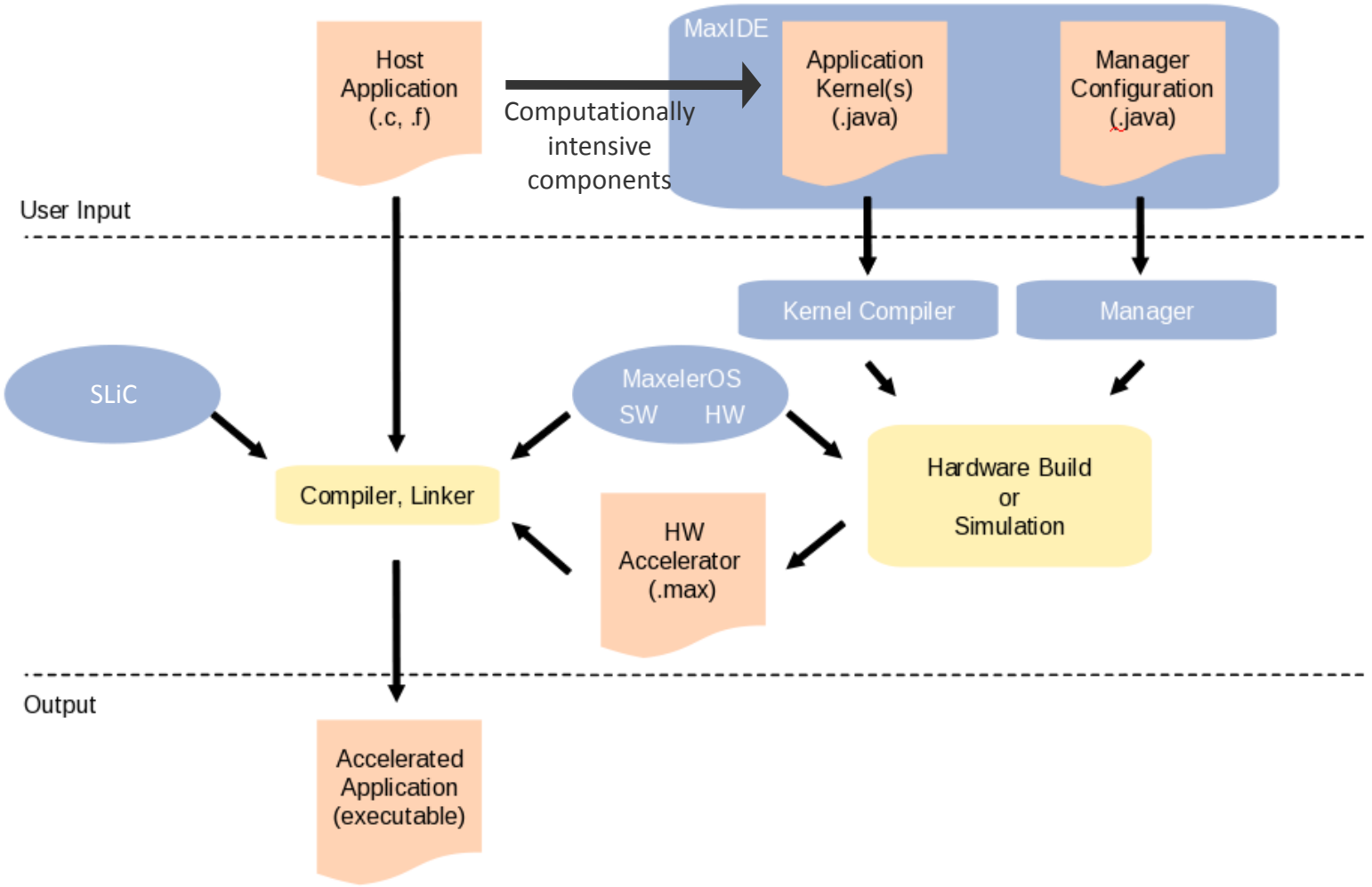
42U in-memory analytics cluster

MAXELER Technologies

Application Programming Process

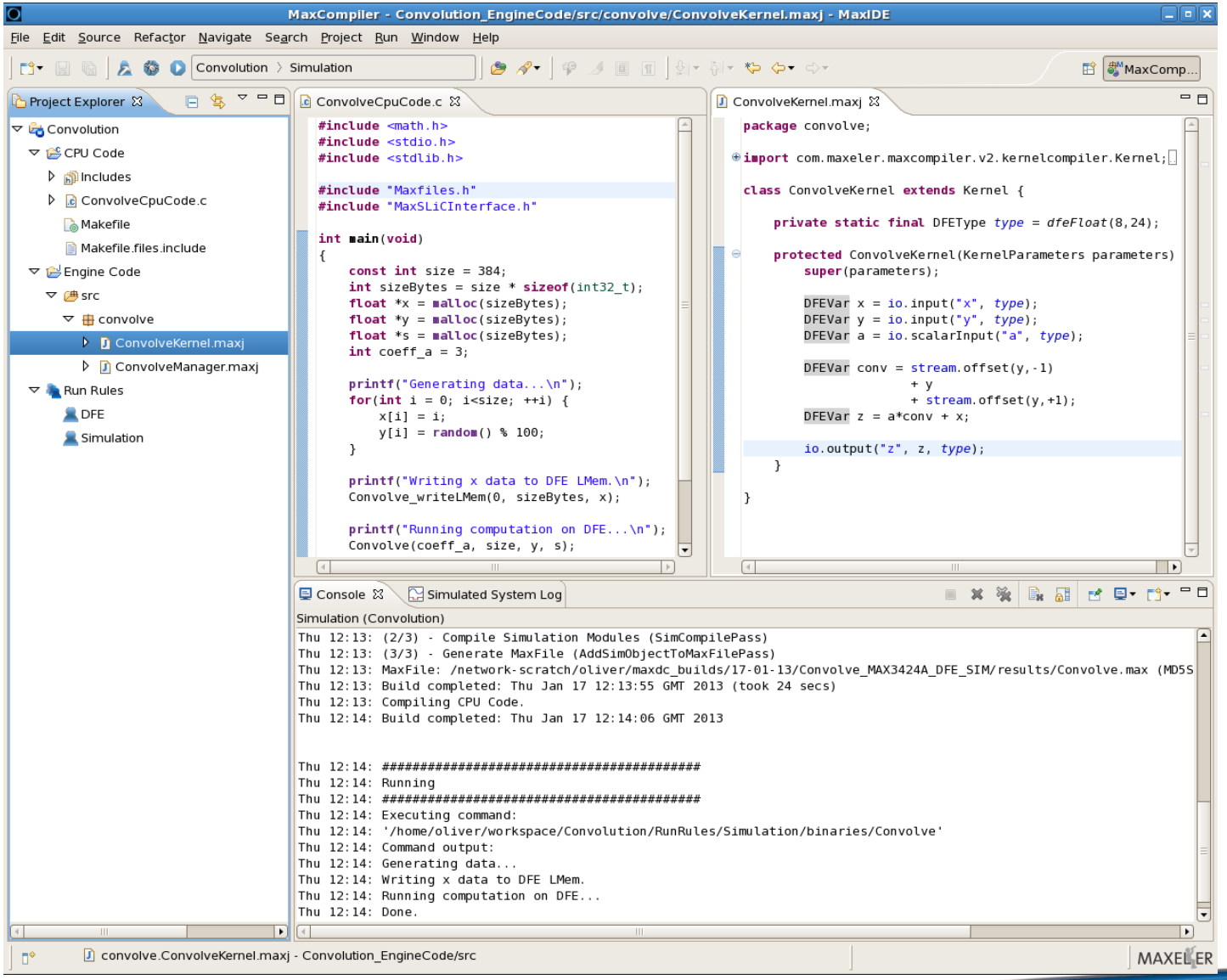


Programming with MaxCompiler



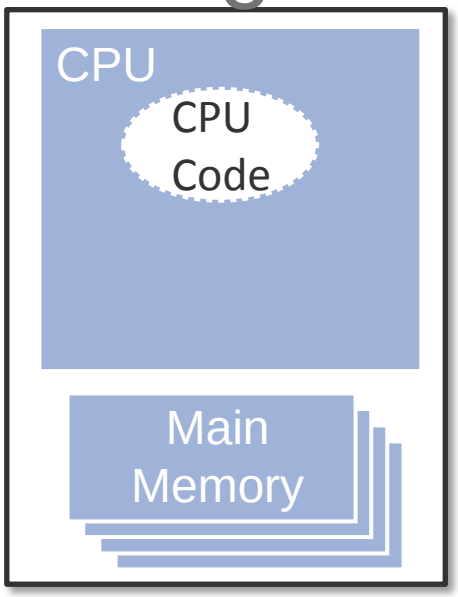
MAXELER Technologies

Programming with MaxCompiler



MAXELER Technologies

Programming with MaxCompiler

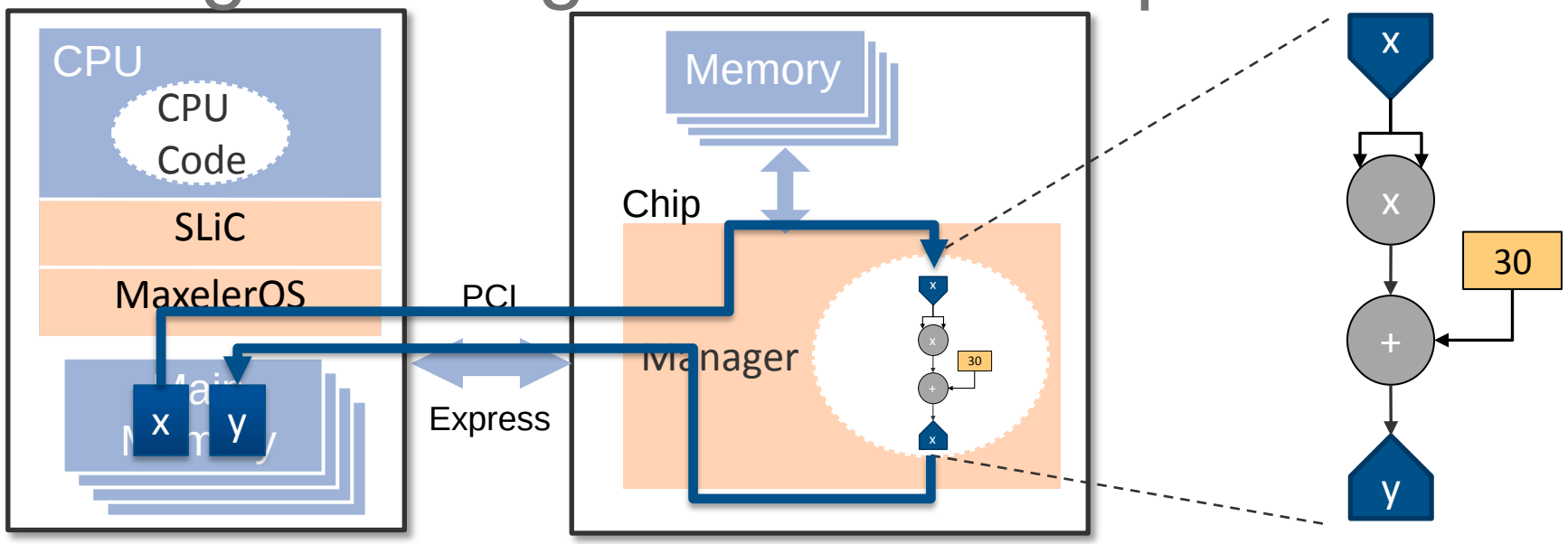


CPU Code (.c)

```
int *x, *y;
for (int i =0; i < DATA_SIZE; i++)
    y[i]= x[i] * x[i] + 30;
```

$$y_i = x_i \times x_i + 30$$

Programming with MaxCompiler



CPU Code (.c)

```
#include “MaxSLiCInterface.h”
#include “Calc.max”
int *x, *y;


Calc(x, y, DATA_SIZE)
```

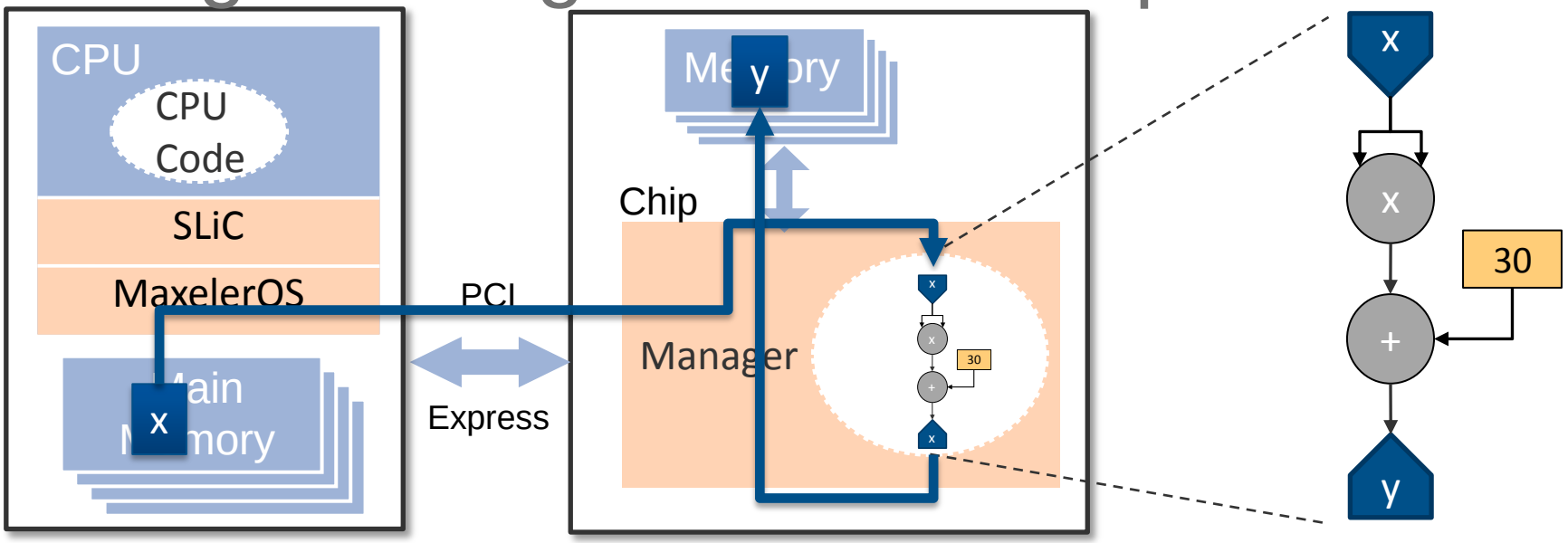
Manager (.java)

```
Manager m = new Manager(“Calc”);
Kernel k =
    new MyKernel();

m.setKernel(k);
m.setIO(
   link(“x", CPU),
   link(“y", CPU));
m.createSLiCInterface();
m.build();
```

MyKernel (.java)

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

DFEVar result = x * x + 30;

io.output("y", result, dfeInt(32));
```

Programming with MaxCompiler



CPUCode (.c)

```
#include “MaxSLiCInterface.h”
#include “Calc.max”
int *x, *y;


Calc(x, DATA_SIZE)
```

Manager (.java)

```
Manager m = new Manager(“Calc”);
Kernel k =
    new MyKernel();

m.setKernel(k);
m.setIO(
   link(“x", CPU),
   link(“y", LMEM_LINEAR1D));
m.createSLiCInterface();
m.build();
```

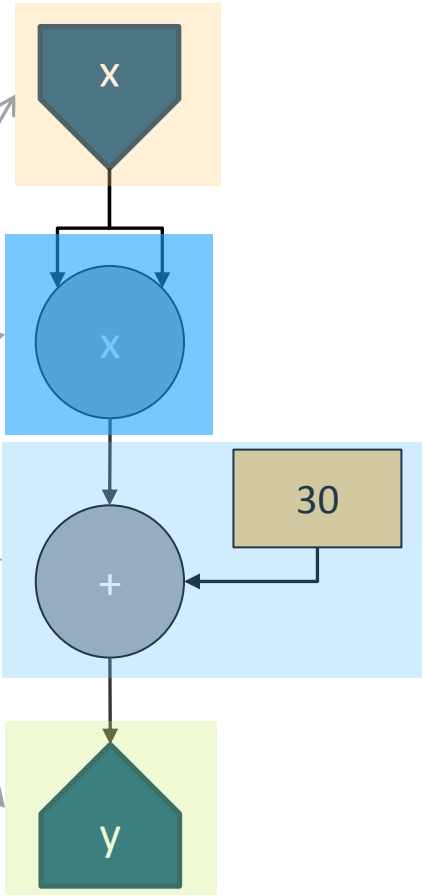
MyKernel (.java)

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

DFEVar result = x * x + 30;

io.output("y", result, dfeInt(32));
```

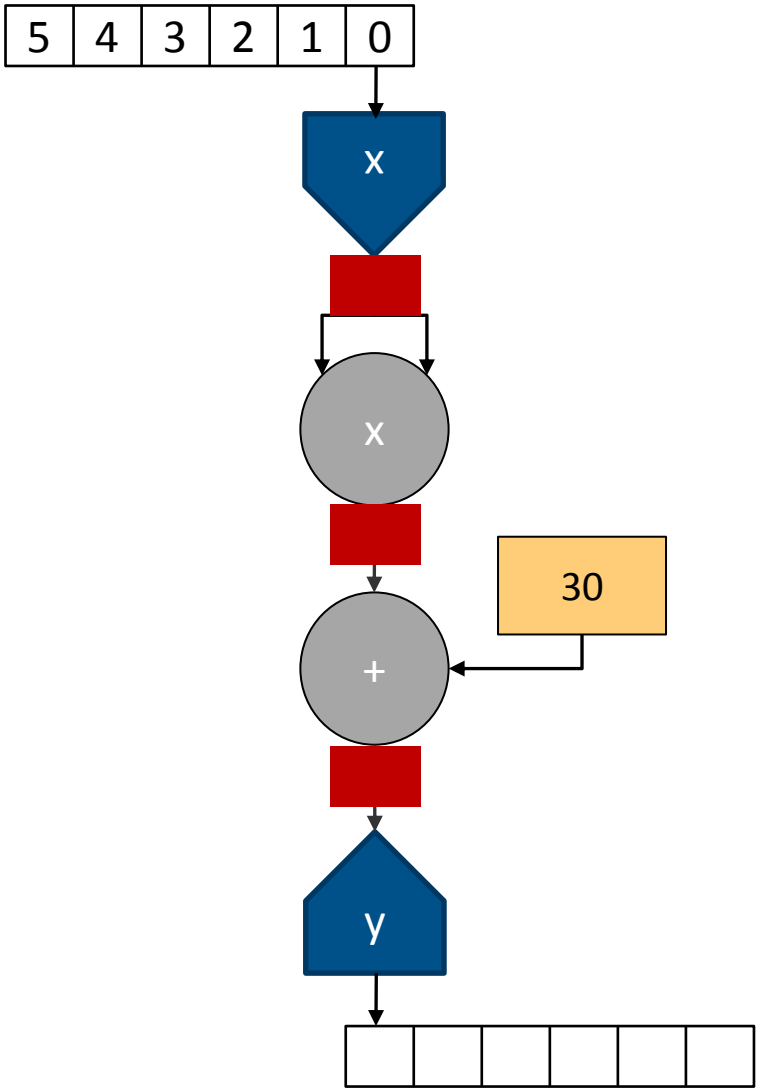

MAXELER Technologies

The Full Kernel

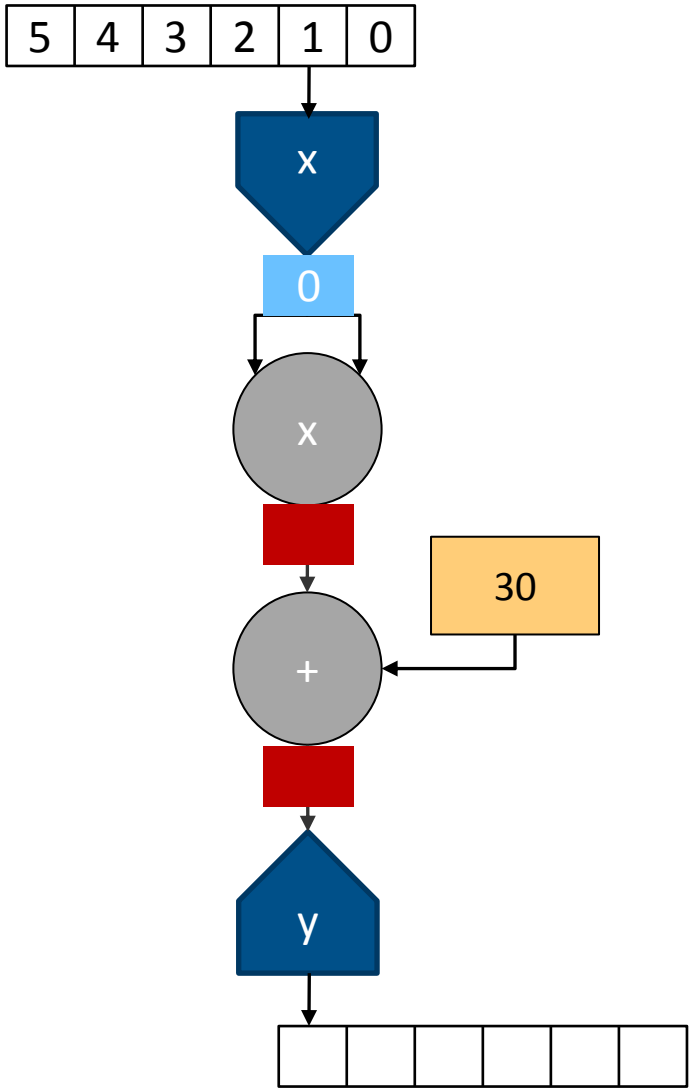
```
public class MyKernel extends Kernel {

    public MyKernel (KernelParameters parameters) {
        super(parameters);

        HWVar x = io.input("x", hwInt(32));

        HWVar result = x * x + 30;

        io.output("y", result, hwInt(32));
    }
}
```



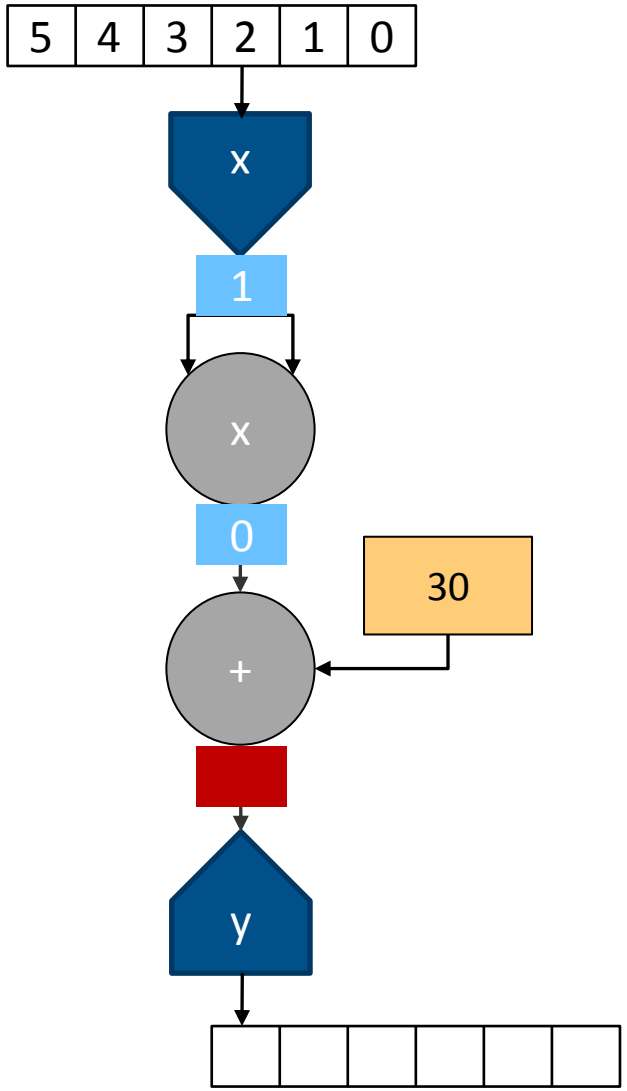
Kernel Streaming



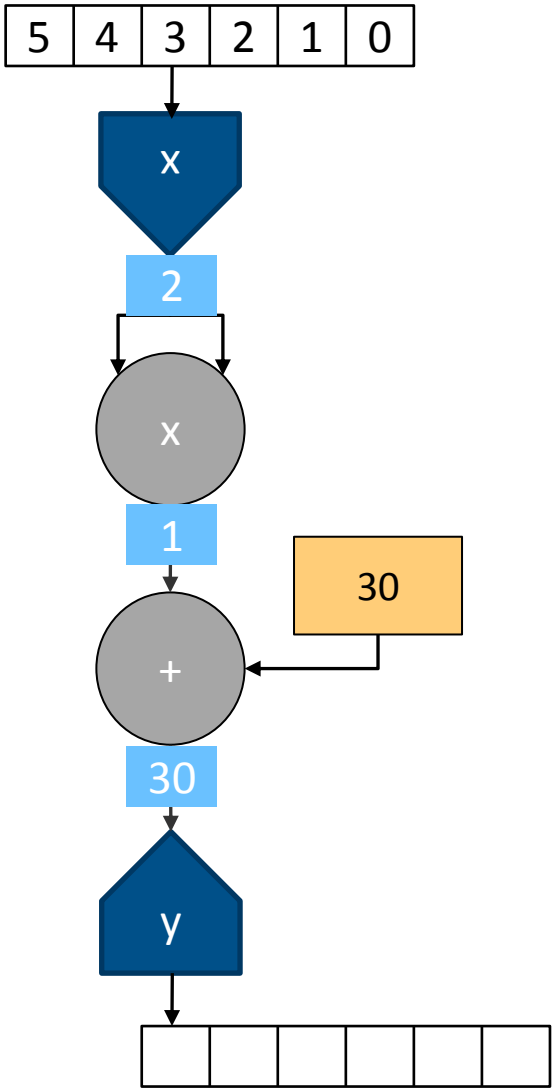
Kernel Streaming



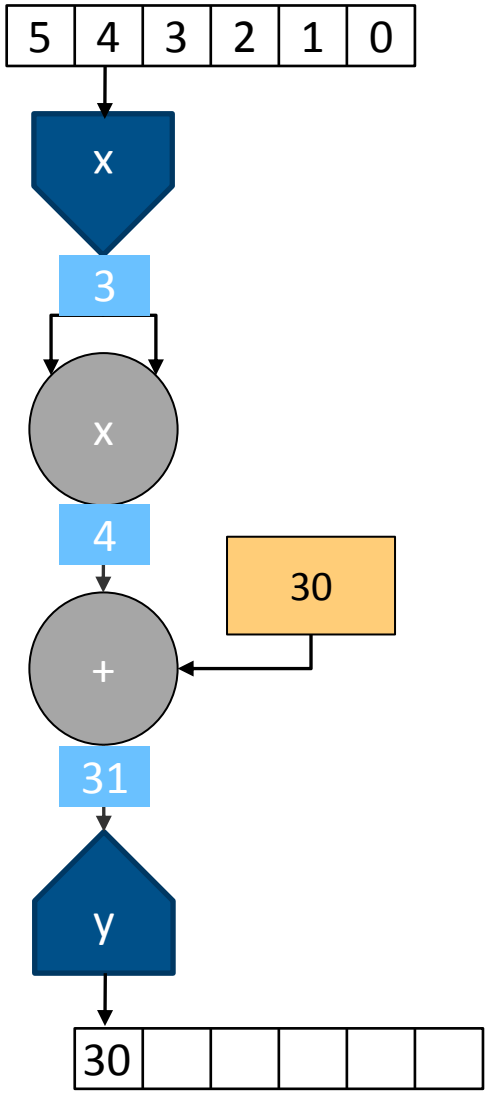
Kernel Streaming



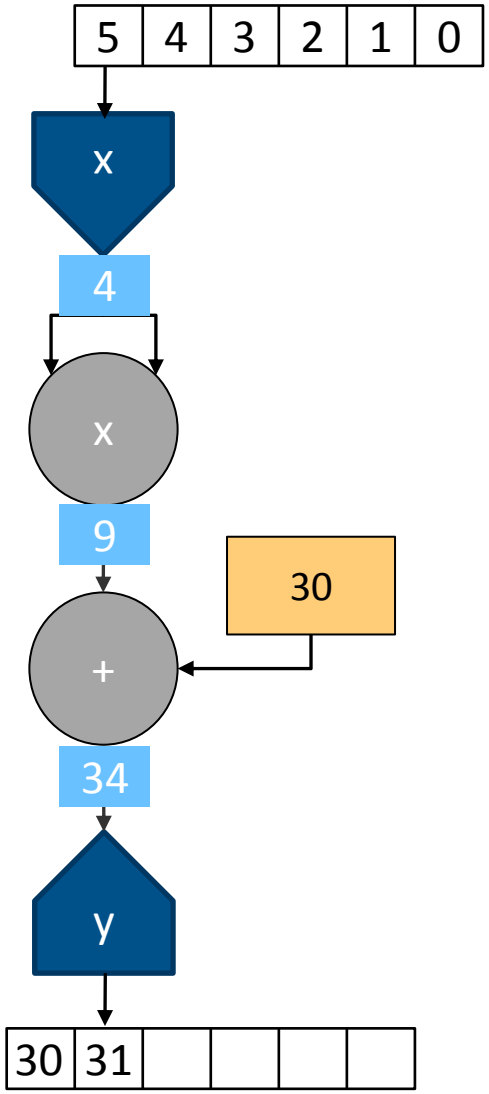
Kernel Streaming



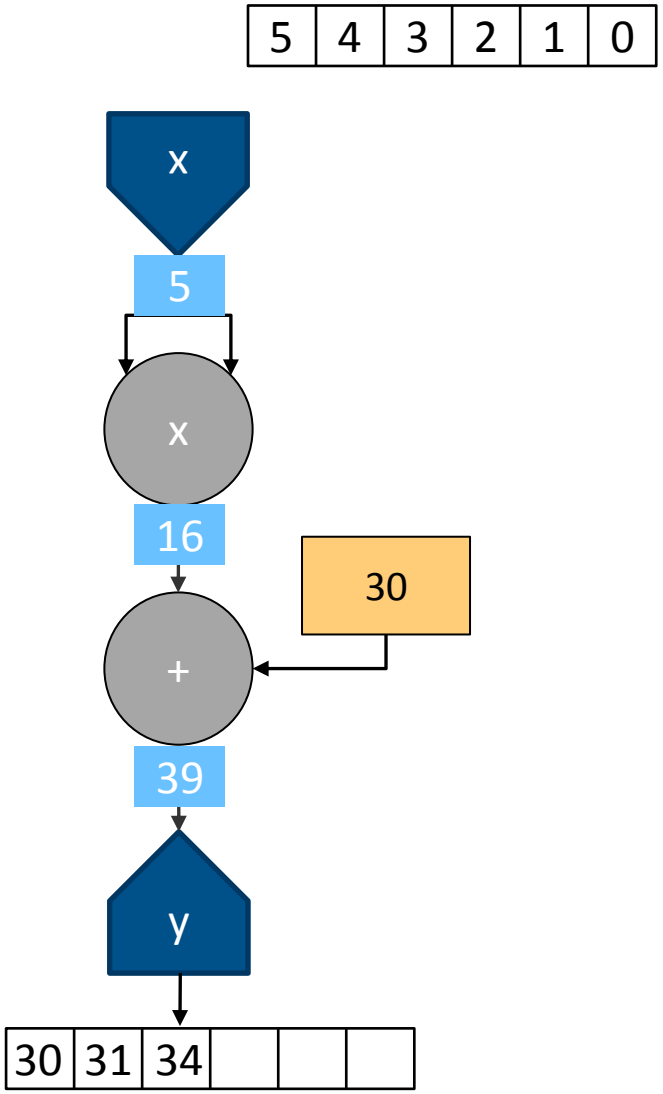
Kernel Streaming



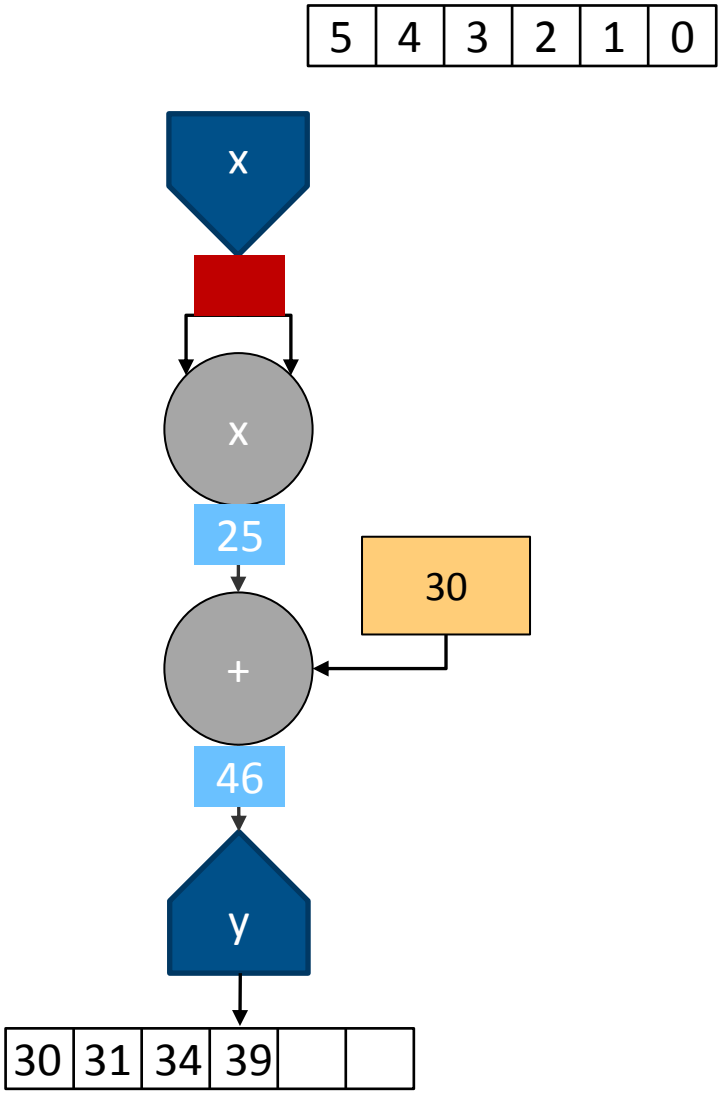
Kernel Streaming



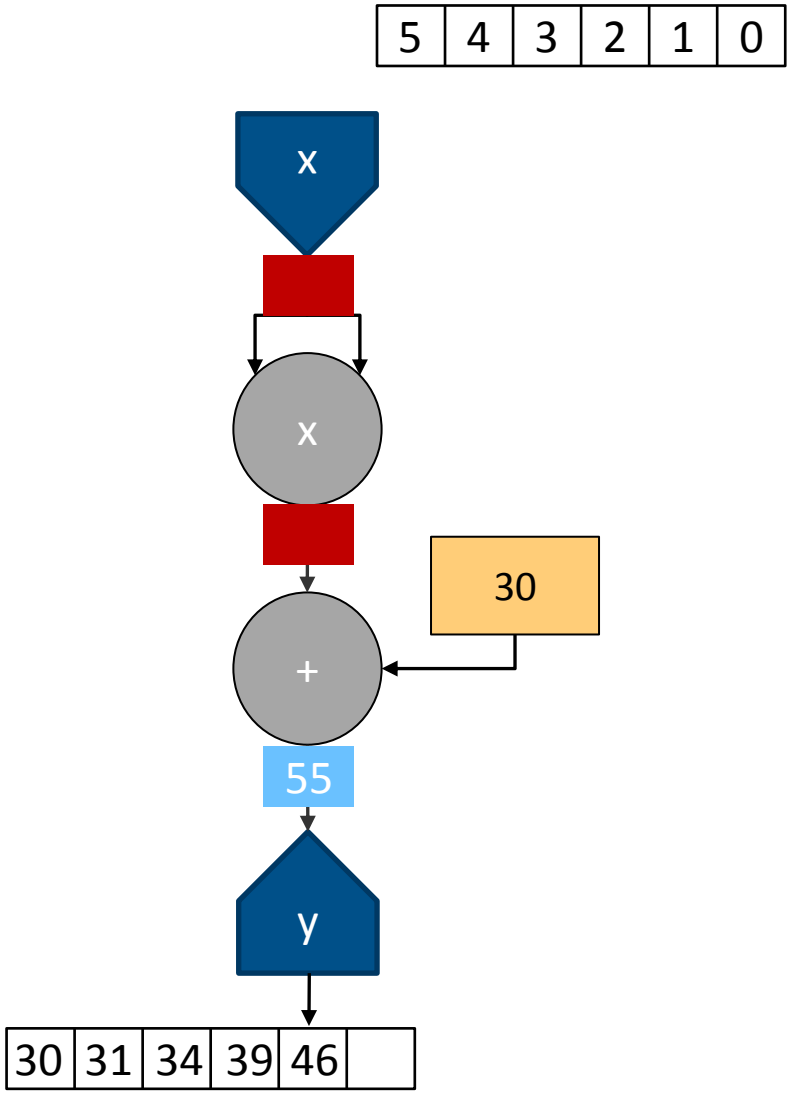
Kernel Streaming



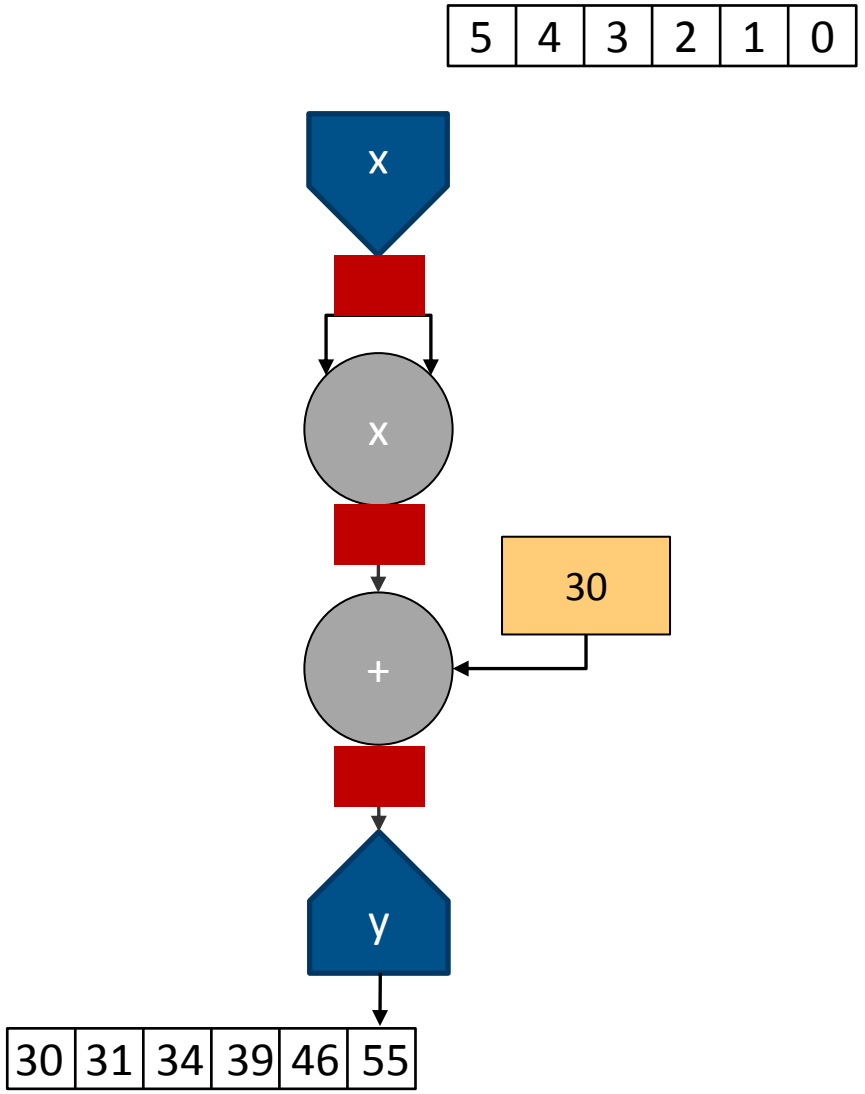
Kernel Streaming



Kernel Streaming

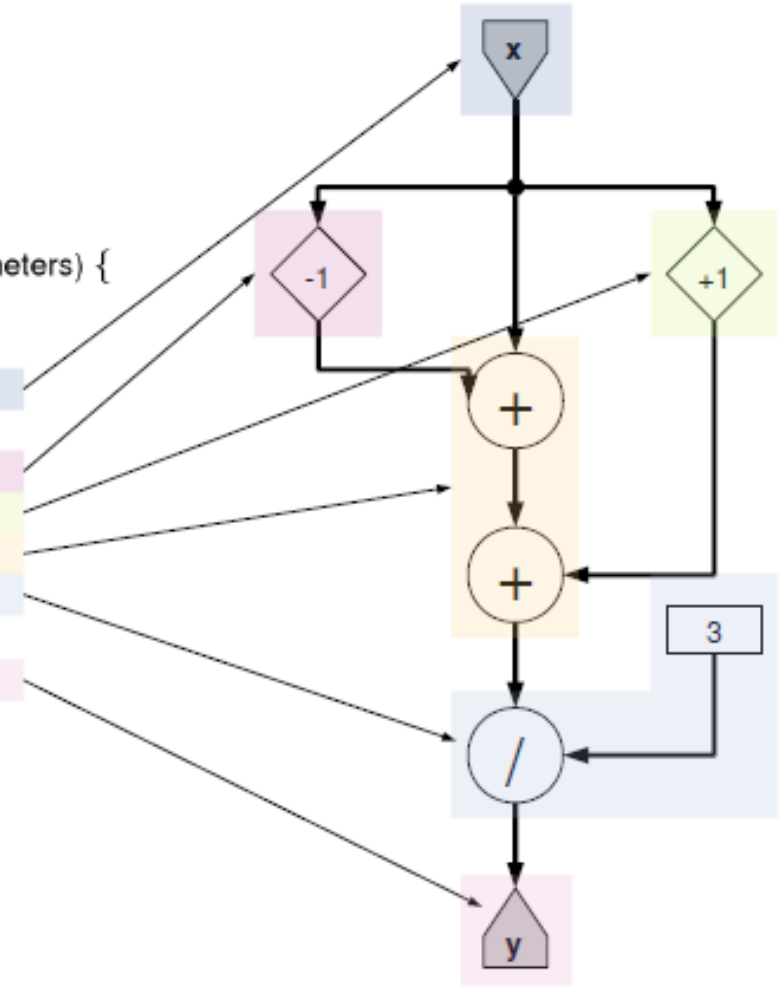


Kernel Streaming

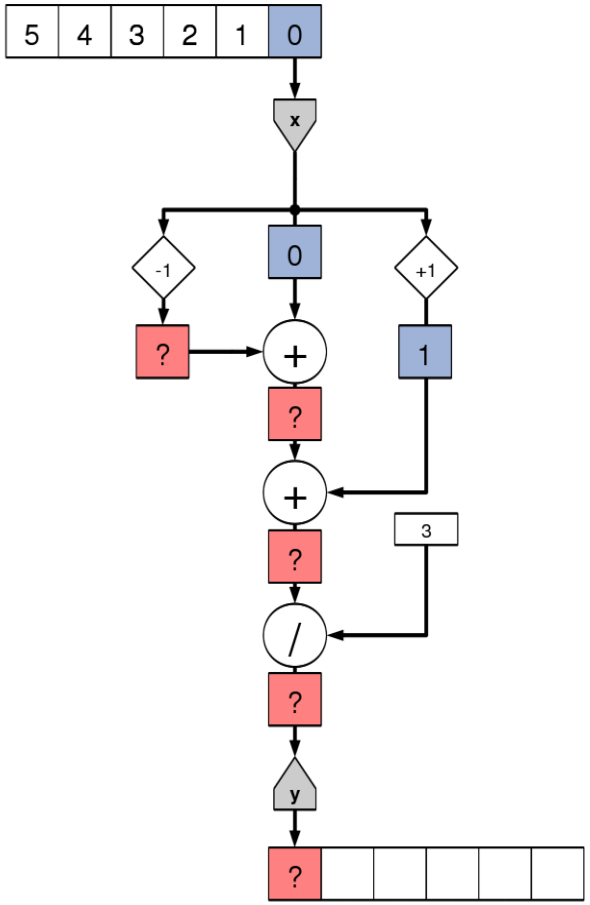


A (slightly) more complex kernel

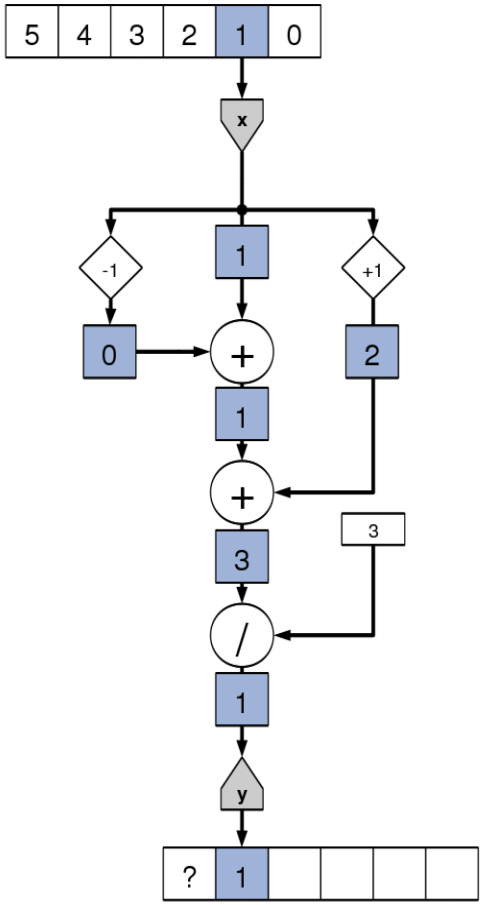
```
14  class MovingAverageSimpleKernel extends Kernel {
15
16      MovingAverageSimpleKernel(KernelParameters parameters) {
17          super(parameters);
18
19          DFEVar x = io.input("x", dfeFloat(8, 24));
20
21          DFEVar prev = stream.offset(x, −1);
22          DFEVar next = stream.offset(x, 1);
23          DFEVar sum = prev + x + next;
24          DFEVar result = sum / 3;
25
26          io.output("y", result, dfeFloat(8, 24));
27      }
28  }
```



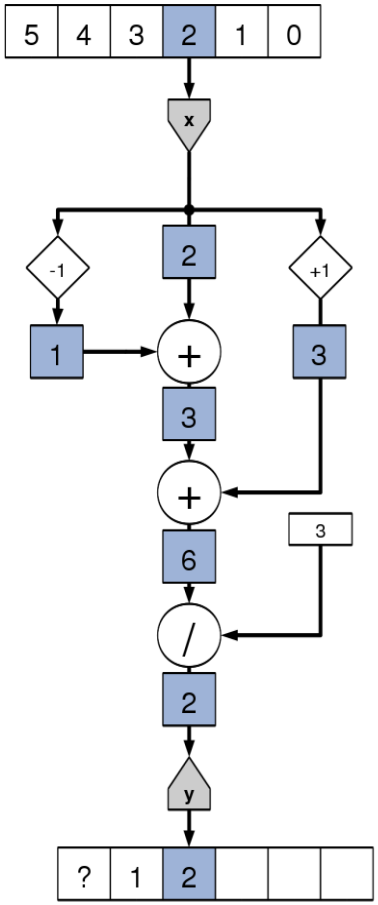
Kernel Execution



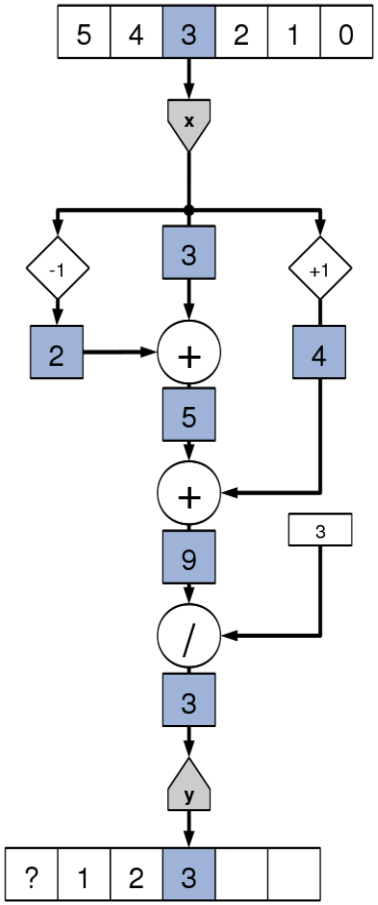
Kernel Execution



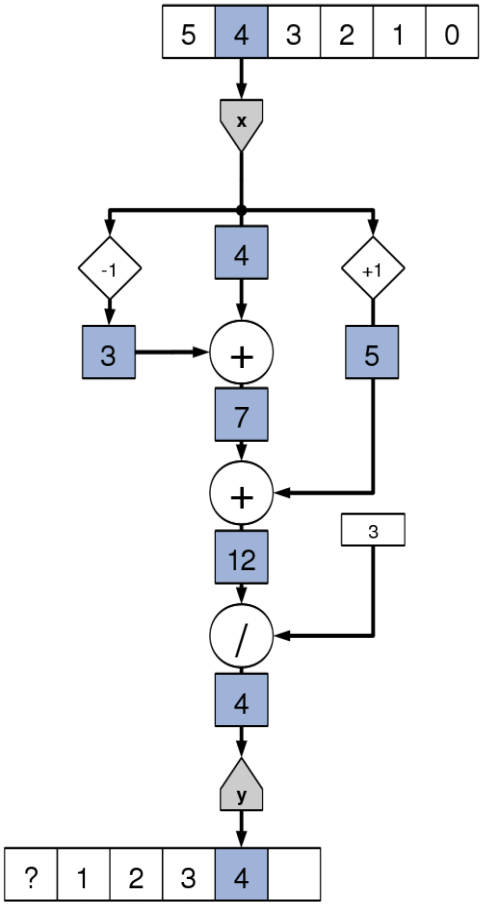
Kernel Execution



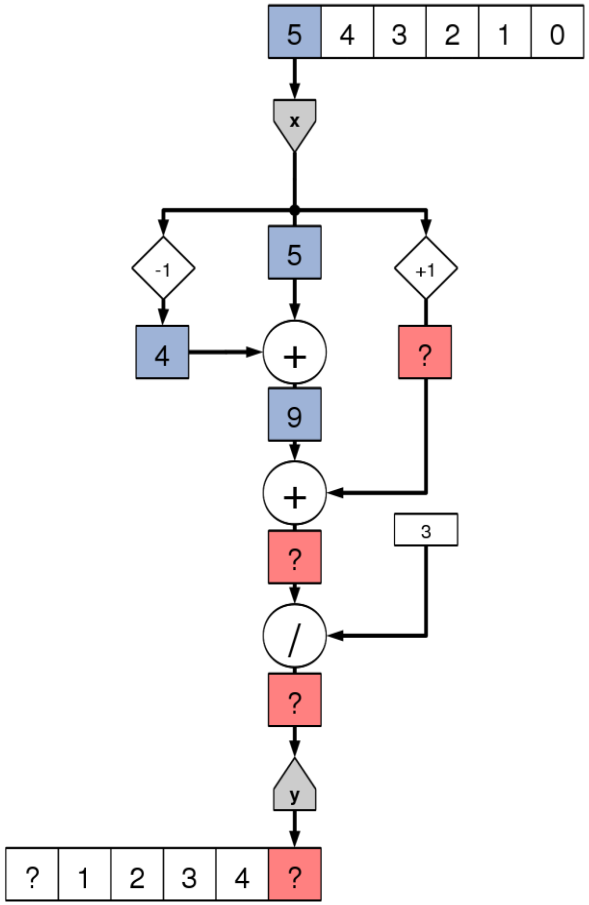
Kernel Execution



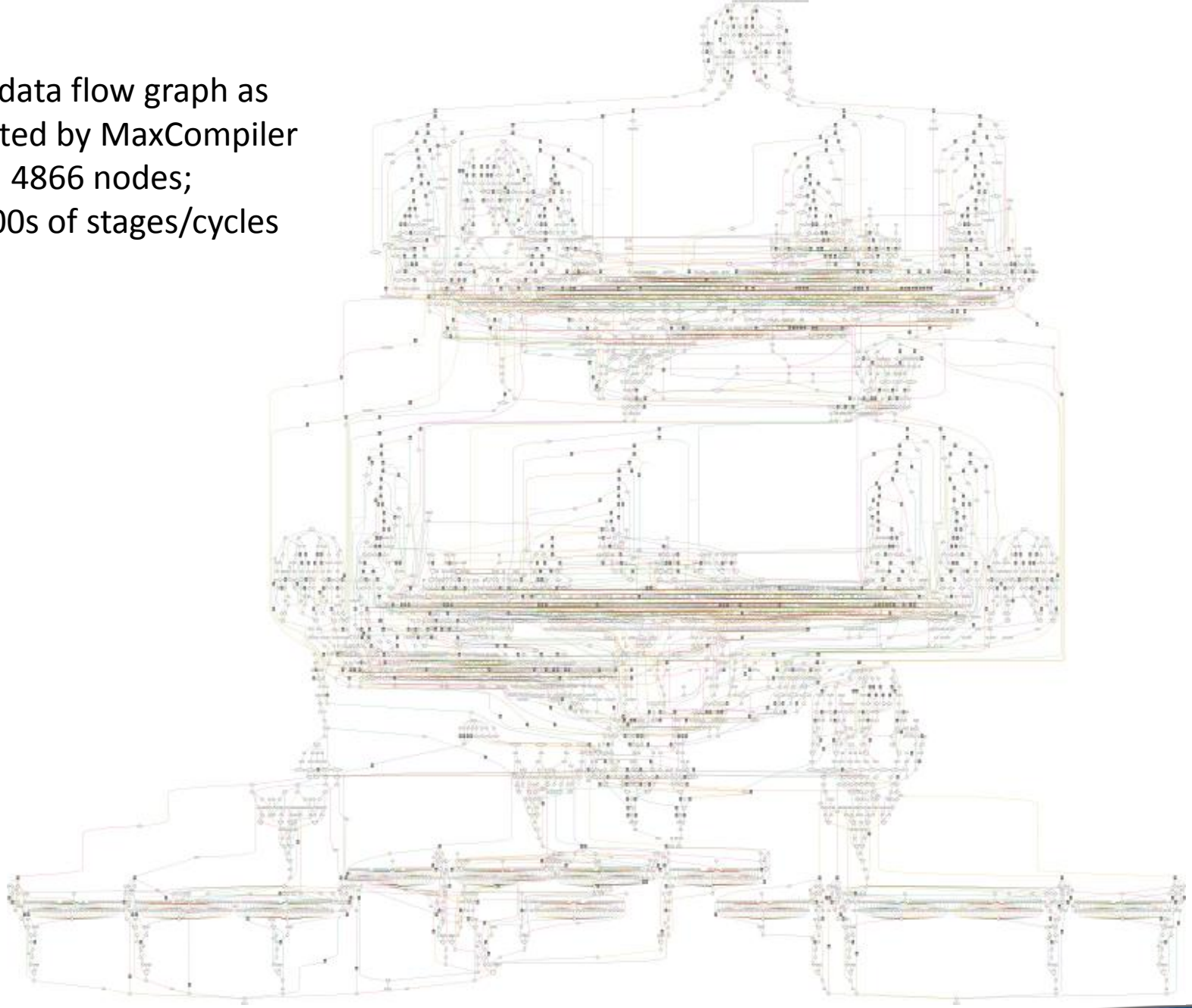
Kernel Execution



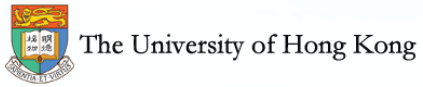
Kernel Execution



Real data flow graph as generated by MaxCompiler
4866 nodes;
10,000s of stages/cycles



Maxeler University Program



Summary & Conclusions

- Tackling a vertical problem at multiple scales can allow you to make major jumps in capability
- Dataflow computing achieves high performance through:
  - Explicitly putting data movement at the heart of the program
  - Employing massive parallelism at low clock frequencies
  - Embodying application co-design
- Many scientific applications can benefit from this approach