CO405H

Computing in Space with OpenSPL

Topic 4: DataFlow Engines (DFEs)

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

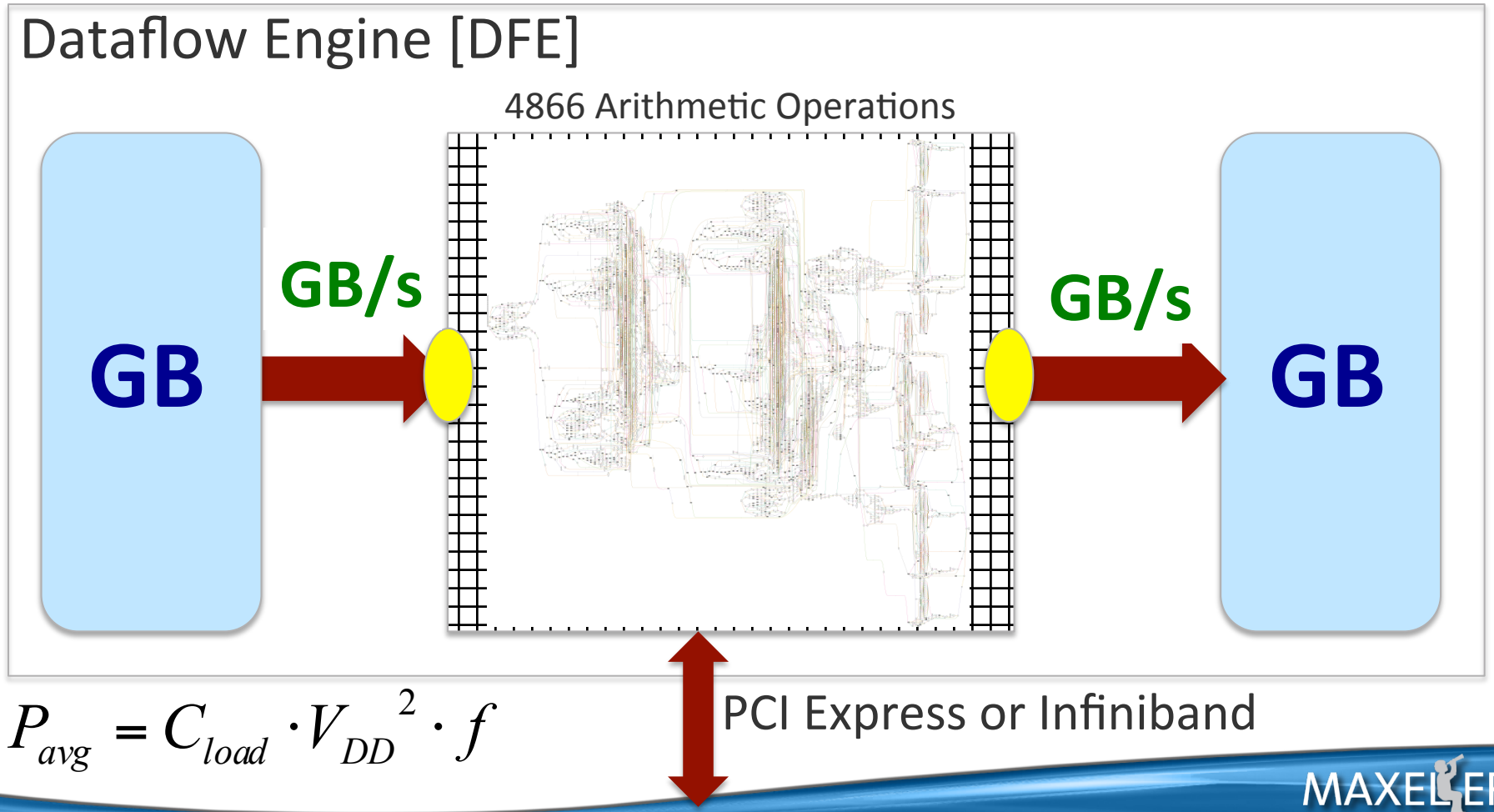
o.mencer@imperial.ac.uk g.gaydadjiev@imperial.ac.uk

Overview

- Dataflow Engines (DFEs) as concrete instantiation of computing in space
- DFE Architecture and Programming Model
- Runtime interface: Using DFEs from MatLab, Python, etc.

Maxeler Multiscale Dataflow Computing

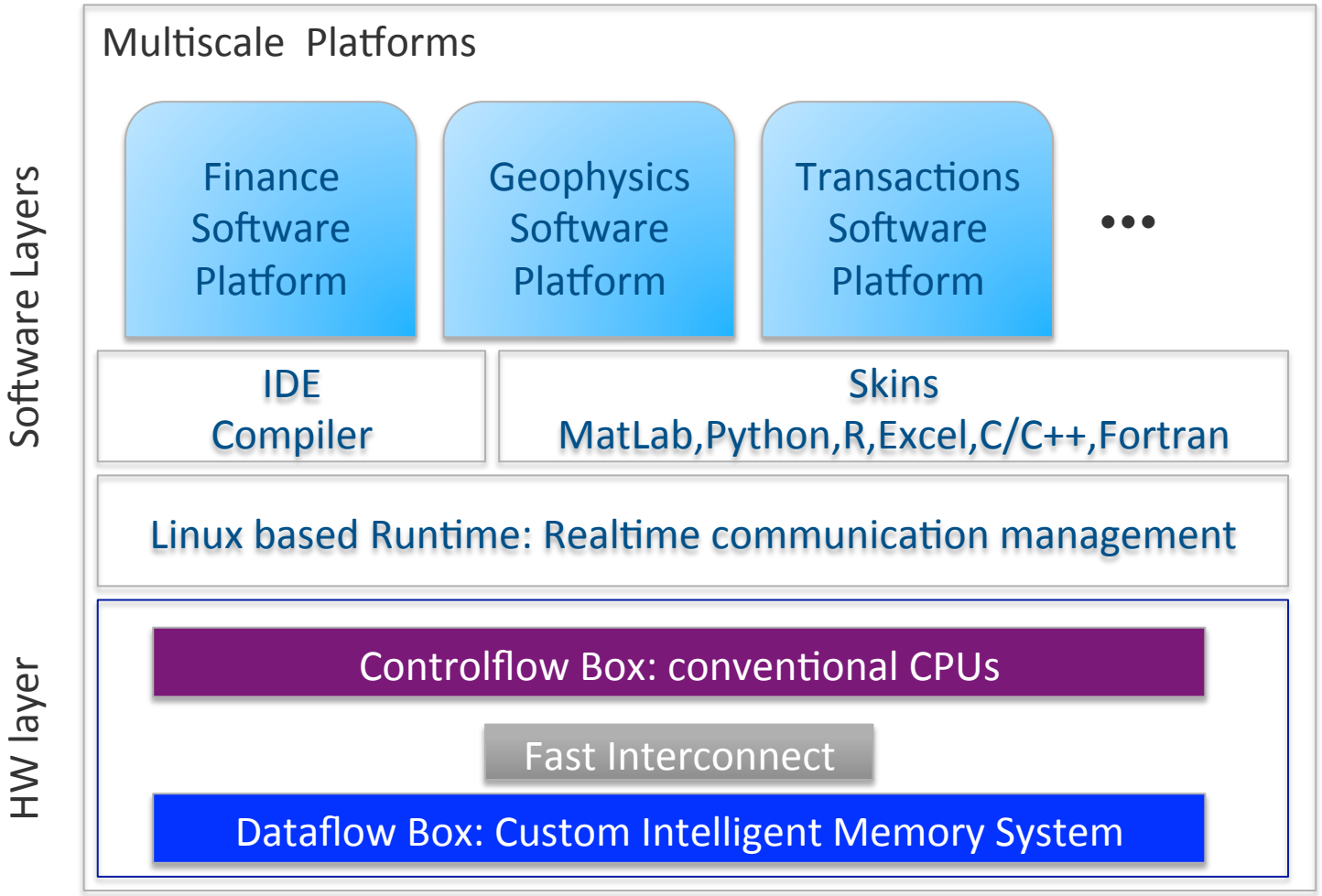
$T_{compute}$=**GB**/[**GB/s**] for up to 10K operations
computing on the stream within a window of 7 MB



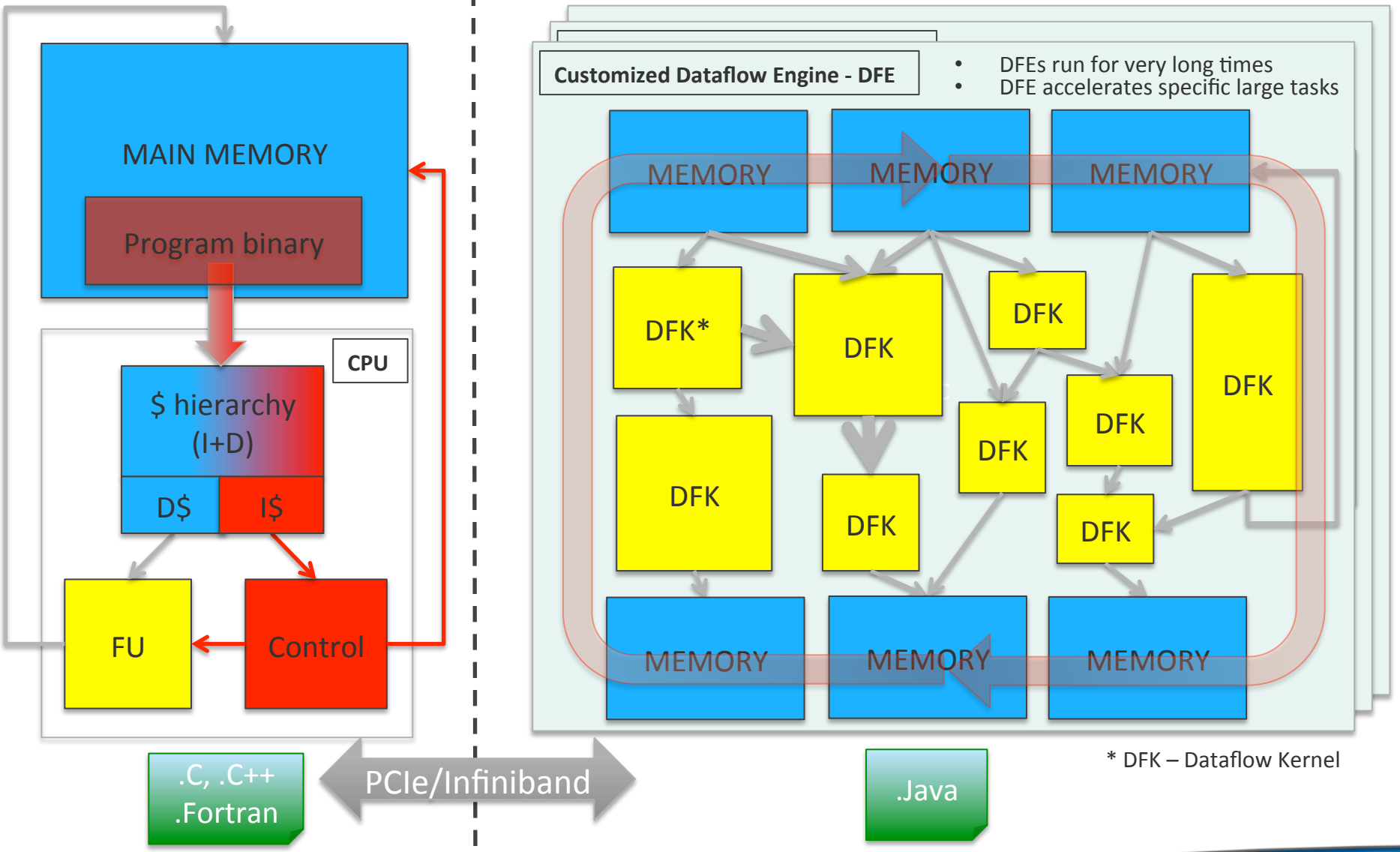
$$P_{avg} = C_{load} \cdot V_{DD}^{\ 2} \cdot f$$

The Platforms

decoupling the data plane and control plane



DFE Architecture Details

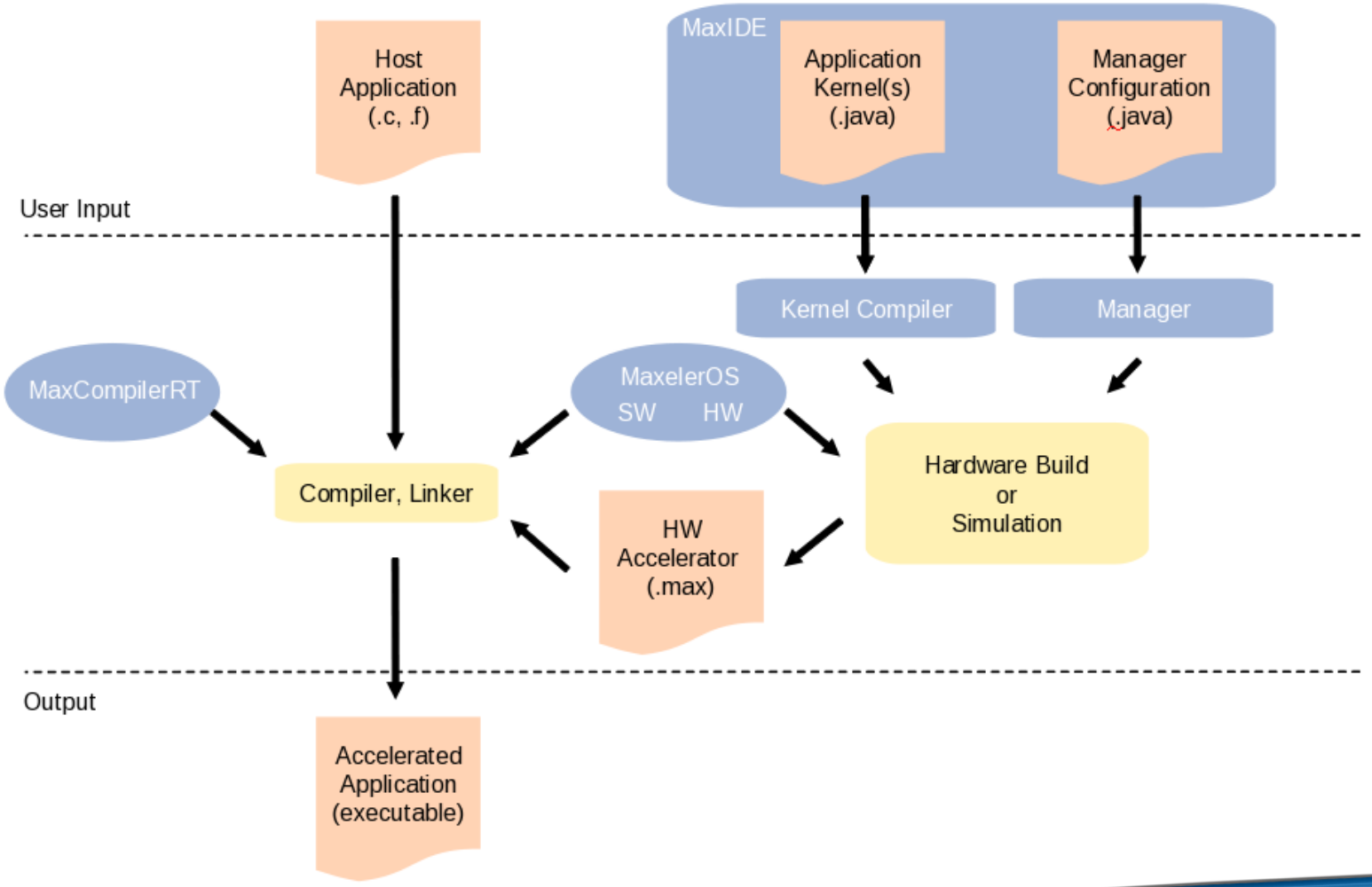


The Compiler

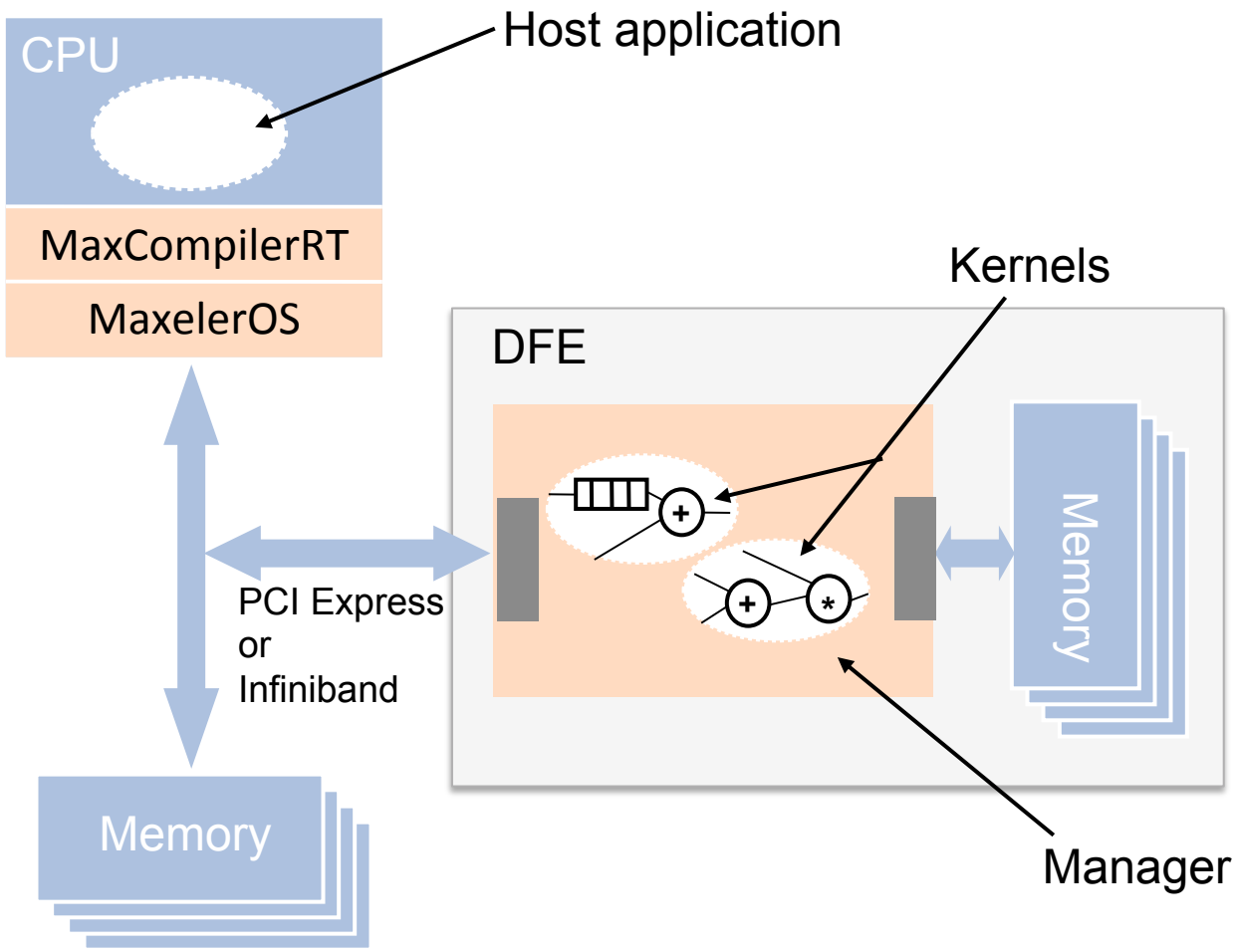
- Write *MaxJ* code to describe the dataflow accelerator
  - *MaxJ* is an extension of Java
  - *Execute* the Java → *generate* the accelerator
- C software on CPUs *uses* the accelerator

```
class MyAccelerator extends Kernel {
    public MyAccelerator(…) {
        DFEVar x = io.input("x", dfeFloat(8, 24));
        DFEVar y = io.input("y", dfeFloat(8, 24));

        DFEVar x2 = x * x;
        DFEVar y2 = y * y;
        DFEVar result = x2 + y2 + 30;

        io.output("z", result, dfeFloat(8, 24));
    }
}
```

Compilation Toolchain

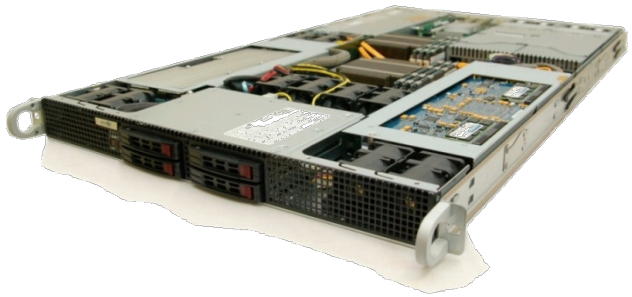


Application Components



Actual DFE Systems

High Density DFEs
Intel Xeon CPU cores and up to 6 DFEs with 288GB of RAM

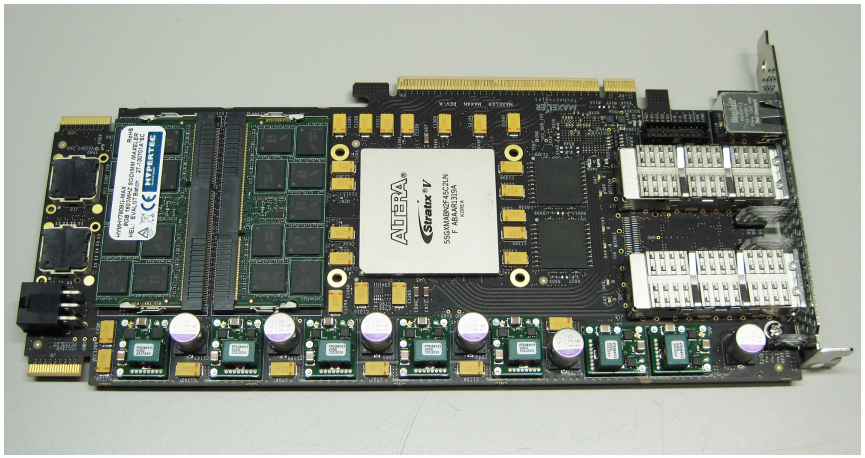


The Dataflow Appliance
Dense compute with 8 DFEs, 384GB of RAM and dynamic allocation of DFEs to CPU servers with zero-copy RDMA access

The Low Latency Appliance
Intel Xeon CPUs and 4 DFEs with direct links to up to six 10Gbit Ethernet connections

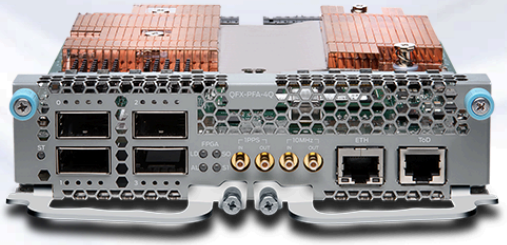
DFE Workstation
Desktop dataflow development system

DFEs in Action



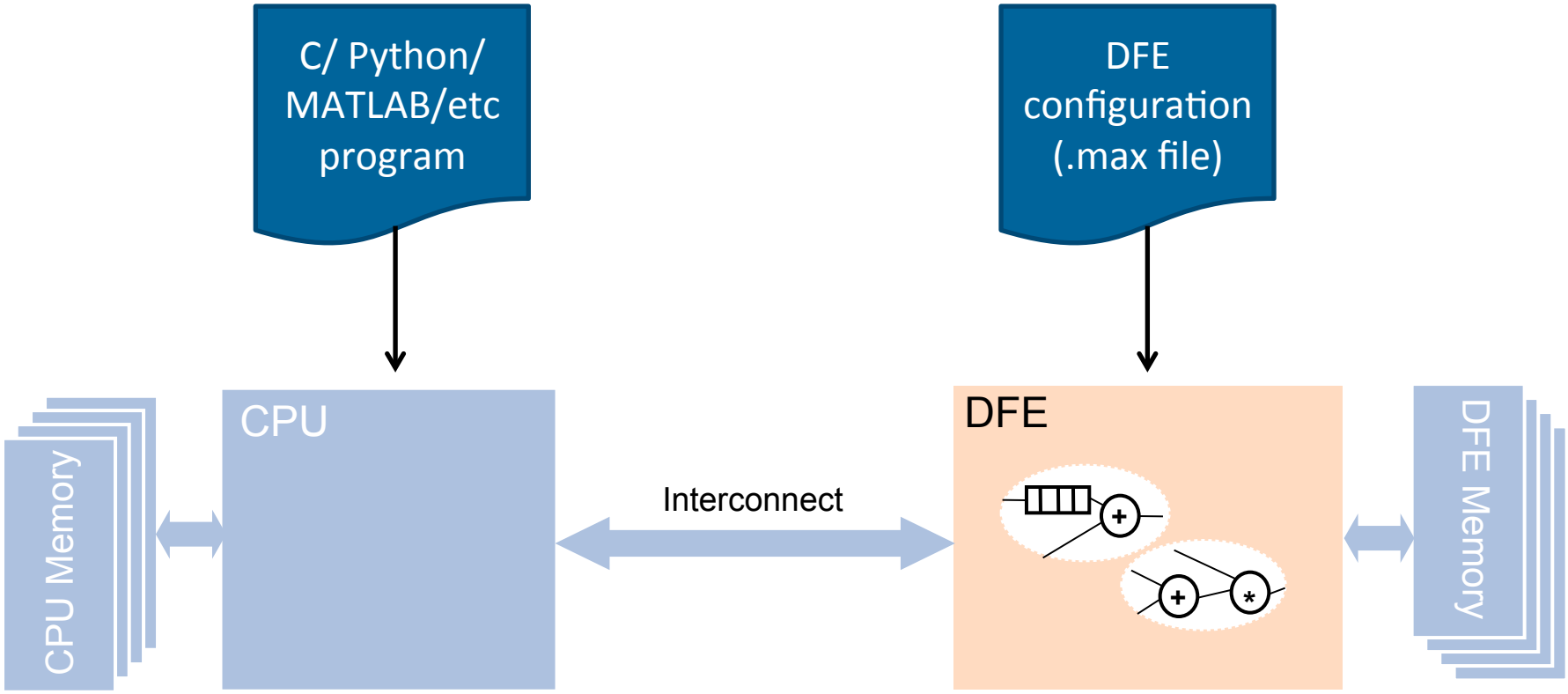
JDFE

Juniper QFX5100 with QFX-PFA-4Q

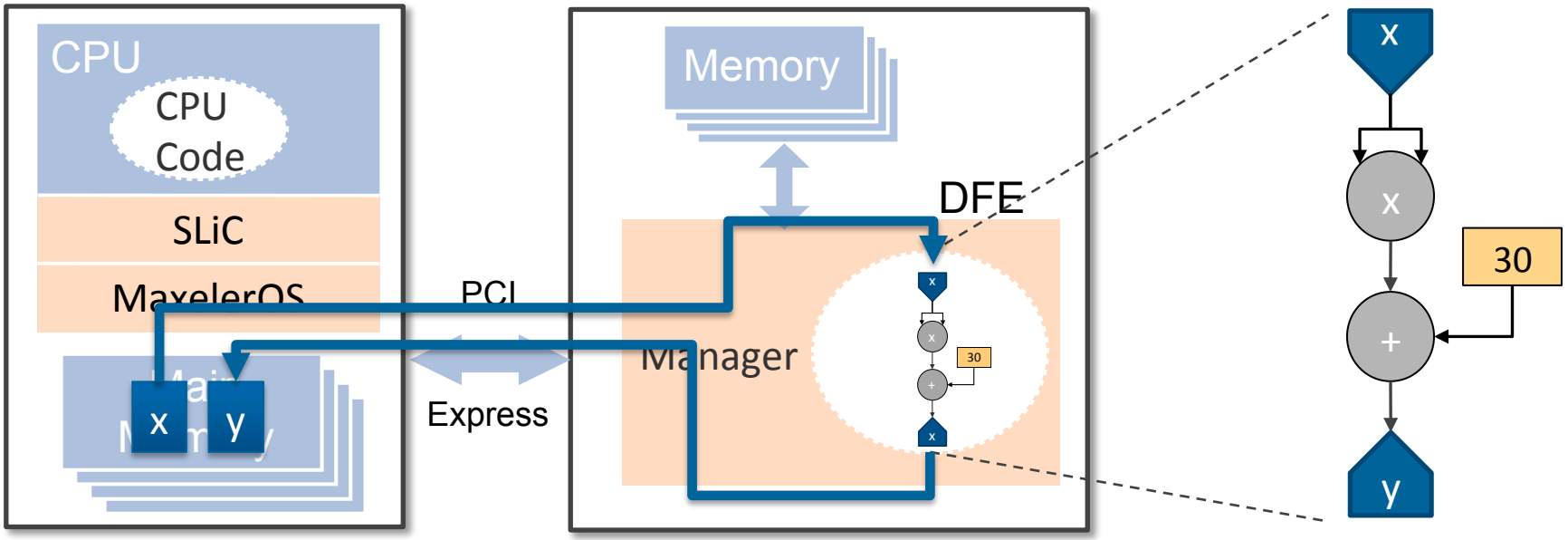


QFX-PFA-4Q based on Maxeler Dataflow Engines

Using DFEs


C/ Python/
MATLAB/etc
program
DFE
configuration
(.max file)
CPU Memory
CPU
Interconnect
DFE
DFE Memory

Development Process



Host Code (.c)

```
int*x, *y;
for (int i =0; i < DATA_SIZE; i++)
   y[i]= x[i] * x[i] + 30;
                y, DATA_SIZE*4);
```

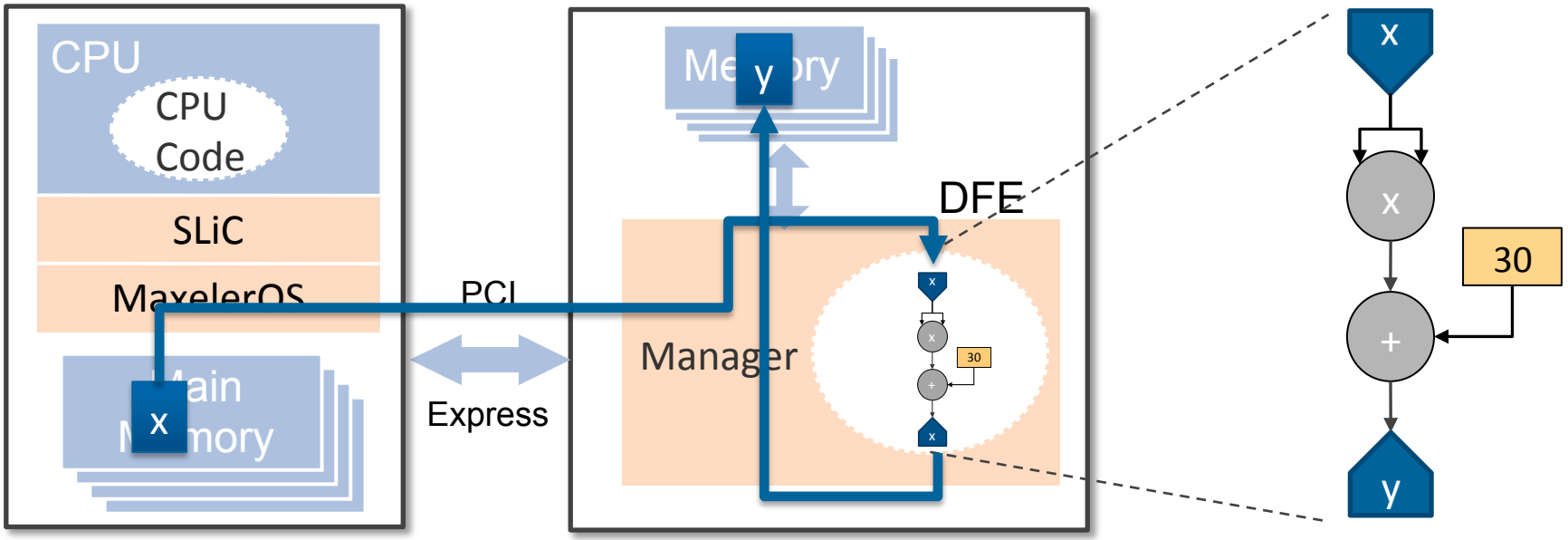
MyManager (.maxj)

```
Manager m = new Manager();
Kernel k =
    new MyKernel();

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

MyKernel (.maxj)

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

DFEVar result = x * x + 30;

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

Development Process



Host Code (.c)

```
int*x, *y;
MyKernel( DATA_SIZE,
          x, DATA_SIZE*4);
```

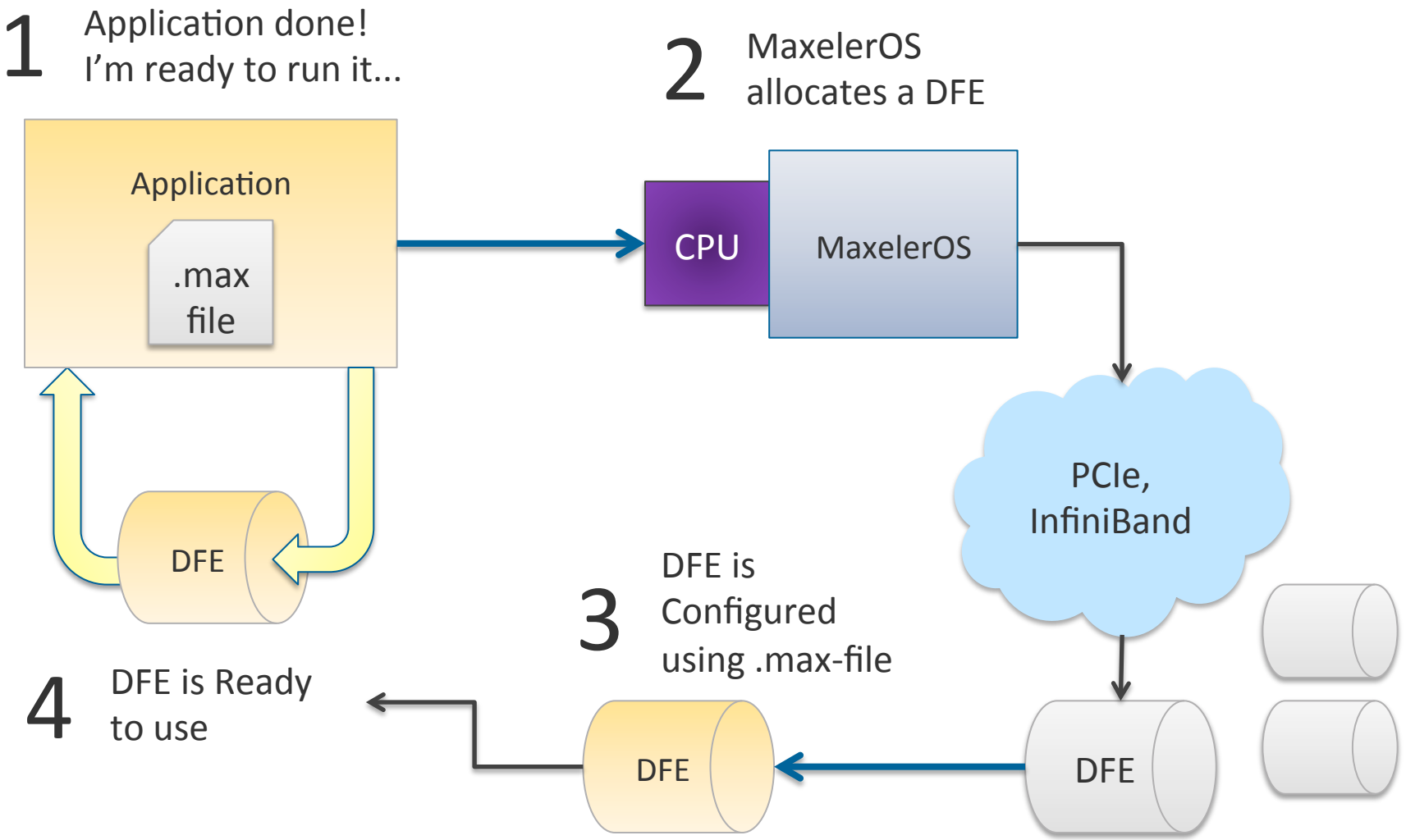
MyManager (.maxj)

```
Manager m = new Manager();
Kernel k =
    new MyKernel();

m.setKernel(k);
m.setIO(
   link("x", CPU),
   link("y", DRAM_LINEAR1D));
m.build();
```

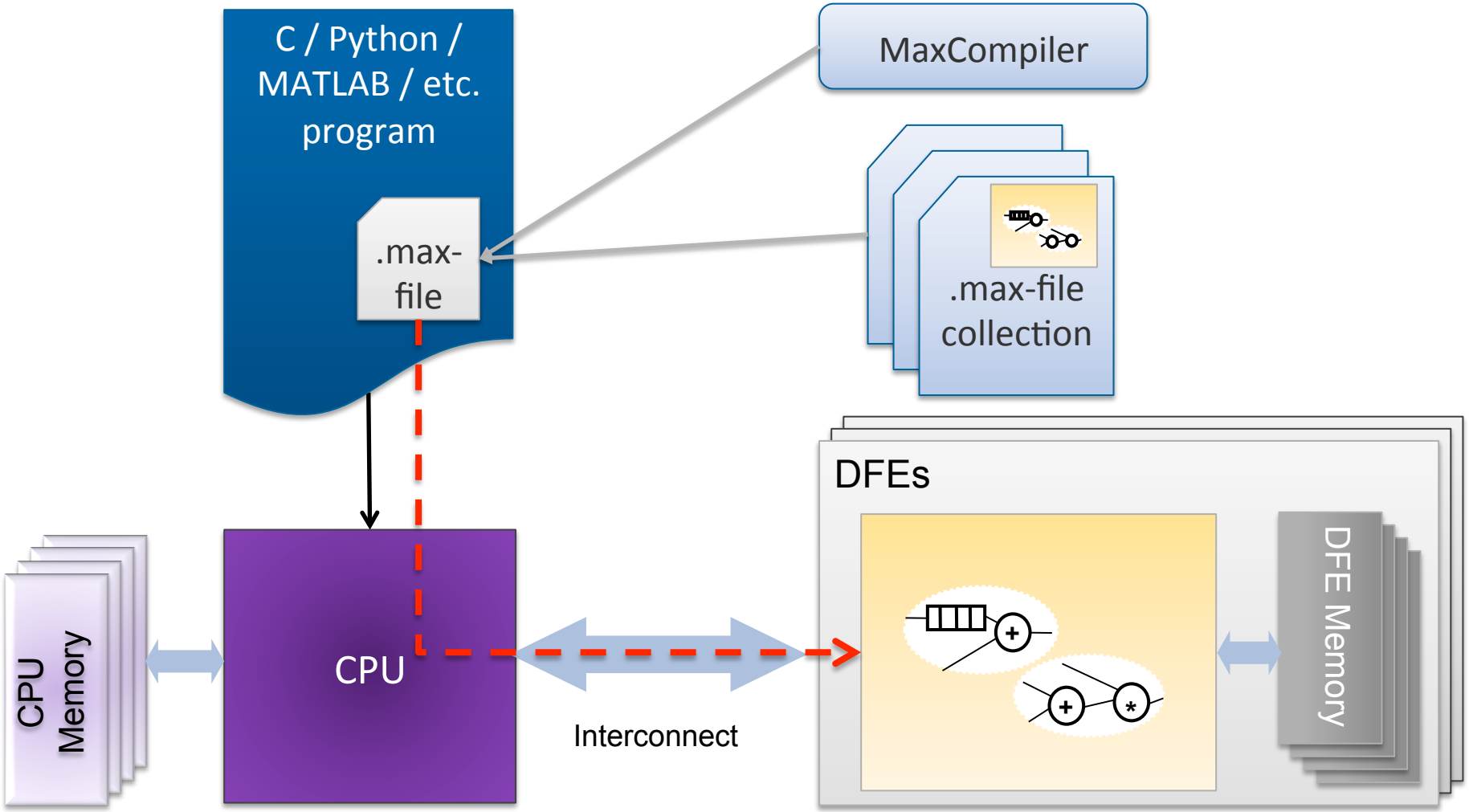
MyKernel (.maxj)

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

DFEVar result = x * x + 30;

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

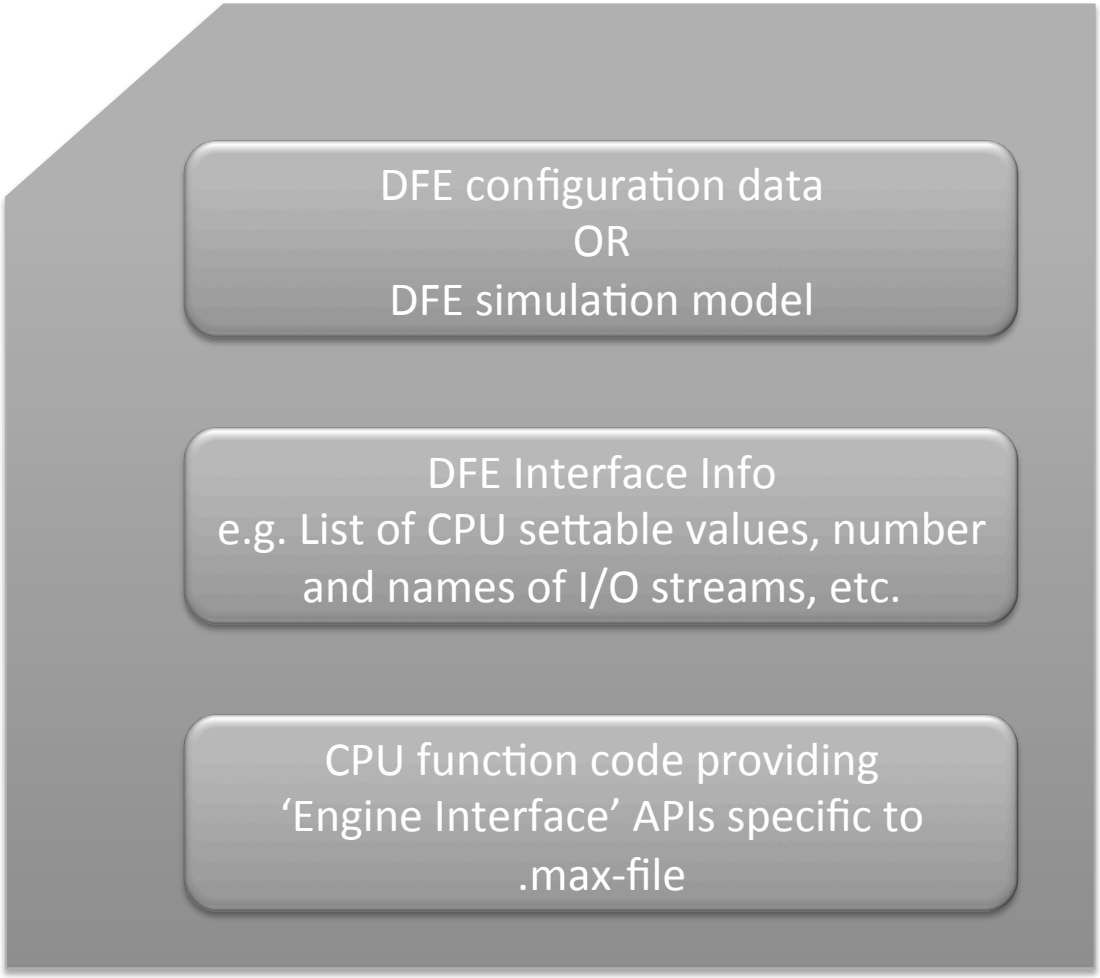
Application Execution



Create DFE Configurations



.max-files


DFE configuration data
OR
DFE simulation model
DFE Interface Info
e.g. List of CPU settable values, number and names of I/O streams, etc.
CPU function code providing 'Engine Interface' APIs specific to .max-file

SLiC Interface: CPU <-> DFE API

- Runtime Interface: **S**imple **Li**ve **C**PU Interface:

  Combination of fixed software function calls and

  MaxCompiler-generated code for interacting with DFEs

- Runtime Interface (SLiC) has a layered interface:
  - **Basic Static** – Single function-call to execute and complete a compute action on any appropriate DFE available
  - **Advanced Static** – Can be more specific about which DFE to use and enables use of multiple DFEs at once
  - **Dynamic** – Remove dependency on generated code from MaxCompiler in .max-file for maximum flexibility

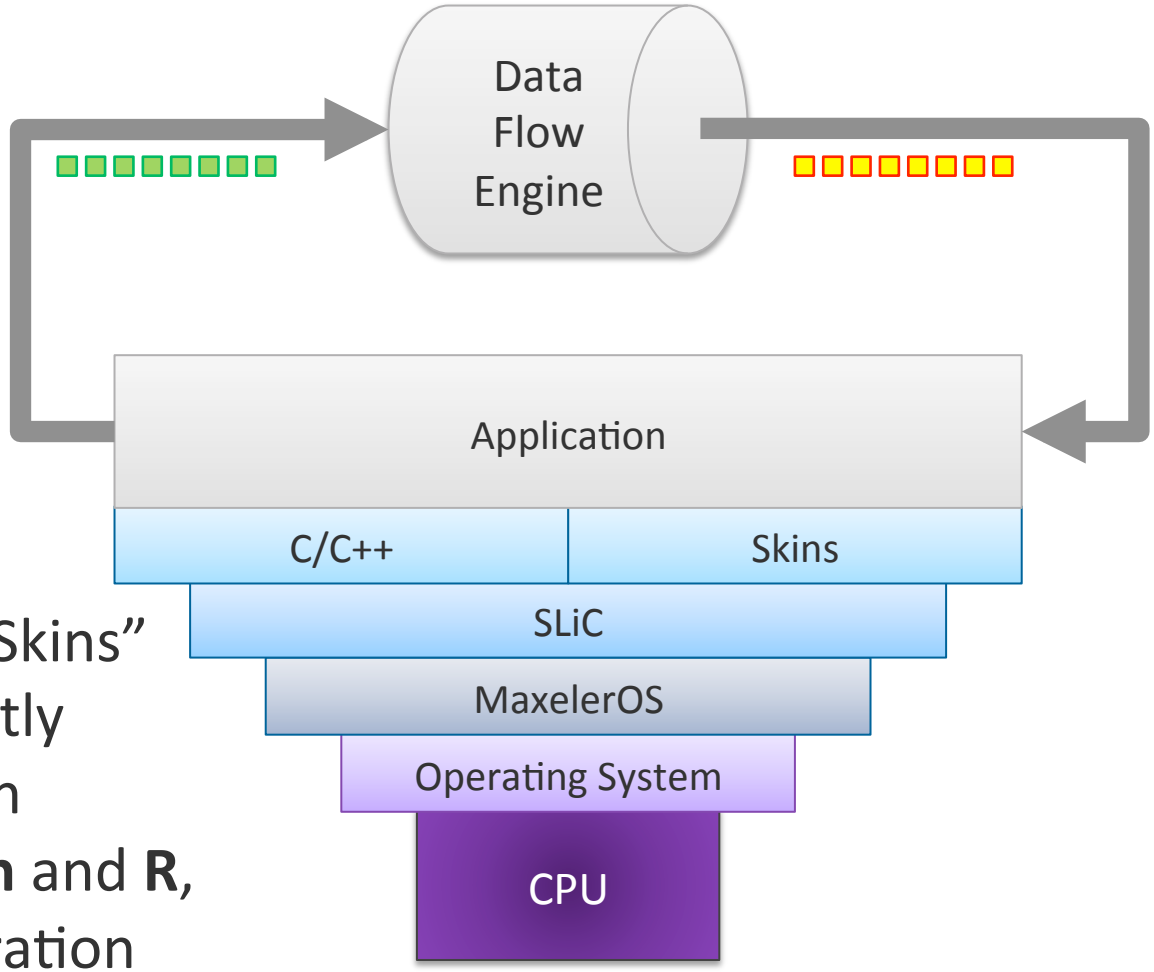
- By default all functions in all layers can be used from C

Example of Basic Static interface

```
#include "Convolve.h"
#include "MaxSLiCInterface.h"

int main(void)
{
  const int size = 384;
  int sizeBytes = size * sizeof(float);
  float *x, *y, *z1, *z2;
  int coeff1 = 3, coeff2 = 5;

  printf("Generating data...\n");
  // Initialize x, y data

  printf("Convolving on DFE...\n");
  Convolve(size, coeff1, x, y, z1);
  Convolve(size, coeff2, x, z1, z2);

  printf("Done.\n");
  return 0;
}
```

- Choose a DFE configuration which gets *linked* into your CPU application
- Call the DFE via standard function calls
- Data will be transferred to/from the DFE automatically to carry out the computation

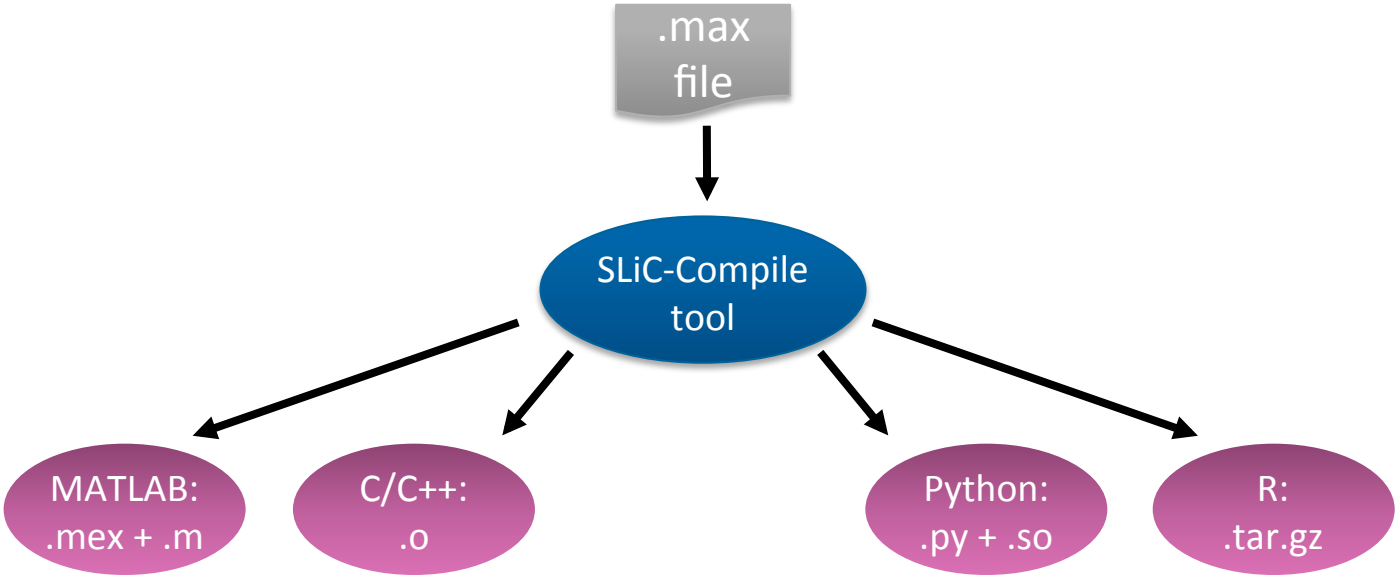
Support for different languages



We **automatically** create "Skins" that let us move data directly from applications written in languages like **Java**, **Python** and **R**, and also from fourth generation environments like **Matlab**

SLiC Skins

- Enable SLiC calls across many languages:
  - Currently: C, Python, MATLAB, R
  - Upcoming: Java, Excel
- Any .max-file usable from any supported language



Skins Example: Python

```
# Import DFEImageLib library functions
from DFEImageLib import *

# Set sharpening parameters
amount = 1.0 # amount of sharpening
sigma  = 0.8 # amount of blurring for generating the mask

# Load image
(h, w, img) = loadPNG("image.png")

# Execute the kernel on the DFE
(mask, sharp_img) = DFEImageLib_sharpen(h, w, amount, sigma, img)
```

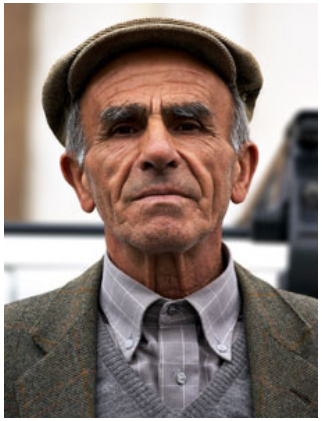
Skins Example: MATLAB

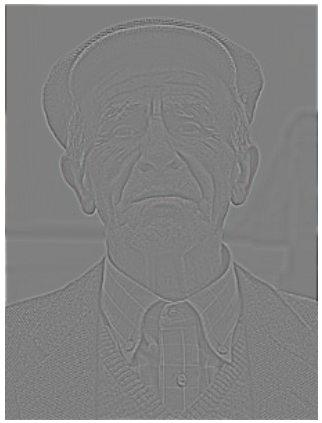
```
% Initialize DFE
dfe = DFEImageLib();

% Set sharpening parameters
amount =  1 % amount of sharpening
sigma  = .8 % amount of blurring for generating the mask

% Load image
img = imgread('image.png');

# Execute the kernel on the DFE
[mask, sharp_img] = dfe.sharpen(amount, sigma, img);
```

Skins Example: R (Statistical Modeling)

```
library(jpeg)
# Installed with “R CMD INSTALL DFEImageLib”
library(DFEImageLib)

# Load the image
img <- readPNG("image.png")

# Set sharpening parameters
amount <-  1; # amount of sharpening
sigma  <- .8; # amount of blurring

# Execute the kernel on the DFE
outputs <-
    DFEImageLib_sharpen(amount, sigma, img);

# Extract output from list
mask      <- outputs$outstream_mask;
Sharp_img <- outputs$outstream_usm;
```

img





mask

sharp_img

Exercises

1. Login into WebIDE and open the *movingAverage* example. Study all three project files on both sides of the system (CPUCode and EngineCode). Compile and study the kernel graphs.

2. Run the example and check if it produces the expected results.
   (Hint: you may consider changing the input data values)

3. Change the given 3 values moving average example to work on a 5 values window. Verify the correctness of the results and study the difference in the new graphs in comparison to the original graphs.