Multiscale Dataflow Programming

Version 2014.1

MAXELER
Technologies
MAXIMUM PERFORMANCE COMPUTING

Version 2014.1
May 19, 2014

Contact Information

Sales/general information: info@maxeler.com

US Office

Maxeler Technologies Inc
Pacific Business Center
2225 E. Bayshore Road
Palo Alto, CA 94303, USA.
Tel: +1 (650) 320-1614

UK Office

Maxeler Technologies Ltd
1 Down Place
London W6 9JH, UK.
Tel: +44 (0) 208 762 6196

r41970

Foreword

Frequency scaling of silicon technology came to an end about a decade ago. Before this programmers came to expect that processors would simply double their speed every two years or so by increasing processor frequency rate. But with the increasing frequency came increasing power density and, ultimately, heat which proved to be a hard barrier. So while transistor density continues to increase, implementations now turn to some form of parallel processing to improve computational performance.

And there is a dramatic need for performance in many large applications: 3D imaging for geophysics and medical analysis, financial risk analysis, air flow simulations in aerodynamics — the list is extensive. These applications often require large buildings with megawatts for power to support the computers — High Performance Computing (HPC) is an expensive proposition.

The obvious form of parallel processor is simply a replication of multiple processors starting with a single silicon die ("multi core") and extended to racks and racks of interconnected processor+memory server units. Even when the application can be expressed in a completely parallel form, this approach has its own limitations especially accessing a common memory. The more processors used to access common memory data the more likely contention develops to limit the overall speed.

Maxeler Technologies developed an alternative paradigm to parallel computing: Multiscale Dataflow Computing. Dataflow computing was popularized by a number of researchers in the 1980's, especially J. B. Dennis. In the dataflow approach an application is considered as a dataflow graph of the executable actions; as soon as the operands for an action are valid, the action is executed and the result is forwarded to the next action in the graph. There are no load or store instructions as the operational node contains the relevant data. Creating a generalized interconnection among the action nodes proved to be a significant limitation to dataflow realizations in the 1980's. Over recent years the extraordinary improvement in transistor array density allowed emulations of the application dataflow graph. The Maxeler dataflow implementations are a generalization of the earlier work employing static, synchronous dataflow with an emphasis on data streaming. Indeed "multiscale" dataflow incorporates vector and array processing to offer a multifaceted parallel compute platform.

At the heart of Multiscale Dataflow Computing is the programming environment, described in this tutorial. While all this is loosely termed the Maxeler compiler the work is much more than a high level translator. Embedded in it is the approach to writing optimized dataflow programs. There are at least three different optimization processes involved. The application actions are written in a dataflow graph type form, unrolling loops, specifying actions processing a data stream. Next the dataflow from memory must be described so that it can be properly scheduled into the dataflow engine. Finally, multiple dataflow engines can be configured together in various ways for maximum application acceleration. All this is done using familiar programming vernacular such as Java type vocabulary. The essence of the Maxeler programming approach is high performance with high productivity on the part of the programmer.

– Michael J. Flynn, Professor Emeritus, Stanford University

Welcome

Welcome to the Multiscale Dataflow Programming tutorial. To achieve Maximum Performance Computing we strive to combine optimizations on the algorithm level all the way down to the bit level. In this tutorial we show all the components that are at our disposal to balance computation with data movement, control and numerics, while addressing functionality and optimizations. We will start by using predefined dataflow programs before advancing to program Dataflow Engines with new dataflow programs.

The source code for the examples, exercise stubs and solutions in this tutorial are provided in the MaxCompiler distribution.

Document conventions

When important concepts are introduced for the first time, they appear in **bold**.
Italics are used for emphasis.
Directories and commands are displayed in `typewriter` font.
Variable and function names are displayed in `typewriter` font.
Java methods and classes are shown using the following format:

*DFEVar **io**.input(**String** name, DFEVar addr, DFEType type)*

C function prototypes are similar:

*max_engine_t∗ max_load(max_file_t∗ maxfile, **const char**∗ engine_id_pattern);*

Actual Java usage is shown without italics:

io.output("output", myRom, dfeUInt(32));

C usage is similarly without italics:

PassThrough(DATA_SIZE, dataIn, dataOut);

Sections of code taken from the source of the examples appear with a border and line numbers:

```
package chap01_gettingstarted.ex1_passthrough;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

public class PassThroughKernel extends Kernel {
	PassThroughKernel(KernelParameters parameters) {
		super(parameters);

		// Input
		DFEVar x = io.input("x", dfeUInt(32));
		// Output
		io.output("y", x, dfeUInt(32));
	}
}
```

1

Multiscale Dataflow Computing

The programming language is not simply a tool with which a preconceived task or function can be accomplished; it is an extensive basis of structure with which the imagination can interact.

– John Chowning

Maxeler's Multiscale Dataflow Computing is a combination of traditional synchronous dataflow, vector and array processors. We exploit loop level parallelism in a spatial, pipelined way, where large streams of data flow through a sea of arithmetic units, connected to match the structure of the compute task. Small on-chip memories form a distributed register file with as many access ports as needed to support a smooth flow of data through the chip.

Multiscale Dataflow Computing employs dataflow on multiple levels of abstraction: the system level, the architecture level, the arithmetic level and the bit level. On the system level, multiple dataflow engines are connected to form a supercomputer. On the architecture level we decouple memory access from arithmetic operations, while the arithmetic and bit levels provide opportunities to optimize the representation of the data and balance computation with communication.

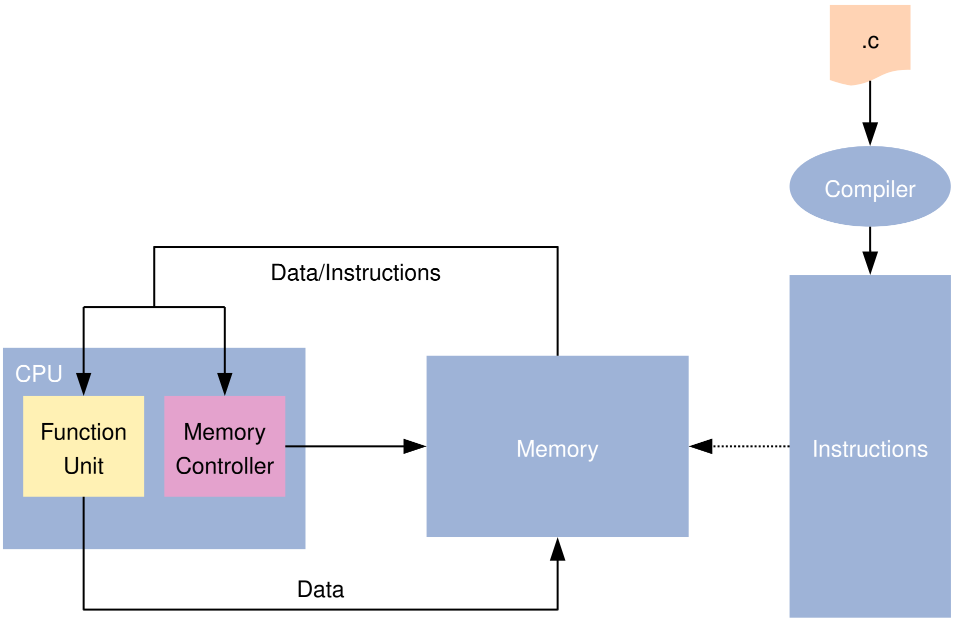


Figure 1: Reuse of functional units over time in a CPU

1.1 Dataflow versus control flow model of computation

In a software application, a program's source code is transformed into a list of instructions for a particular processor ('control flow core'), which is then loaded into the memory, as shown in *Figure 1*. Instructions move through the processor and occasionally read or write data to and from memory. Modern processors contain many levels of caching, forwarding and prediction logic to improve the efficiency of this paradigm, however the programming model is inherently sequential and performance depends on the latency of memory accesses and the time for a CPU clock cycle.

In a dataflow program, we describe the operations and data choreography for a particular algorithm (see *Figure 2*). In a Dataflow Engine (DFE), data streams from memory into the processing chip where data is forwarded directly from one arithmetic unit ('dataflow core') to another until the chain of processing is complete. Once a dataflow program has processed its streams of data, the dataflow engine can be reconfigured for a new application in less than a second.

Each dataflow core computes only a single type of arithmetic operation (for example an addition or multiplication) and is thus simple so thousands can fit on one dataflow engine. In a DFE processing pipeline every dataflow core computes simultaneously on neighboring data items in a stream. Unlike control flow cores where operations are computed at different points in time on the same functional units ("computing in time"), a dataflow computation is laid out spatially on the chip ("computing in space"). Dependencies in a dataflow program are resolved statically at compile time.

One analogy for moving from control flow to dataflow is replacing artisans with a manufacturing model. In a factory each worker gets a simple task and all workers operate in parallel on streams of cars and parts. Just as in manufacturing, dataflow is a method to scale up a computation to a large scale.

The dataflow engine structure itself represents the computation thus there is no need for instructions per se; instructions are replaced by arithmetic units laid out in space and connected for a particular

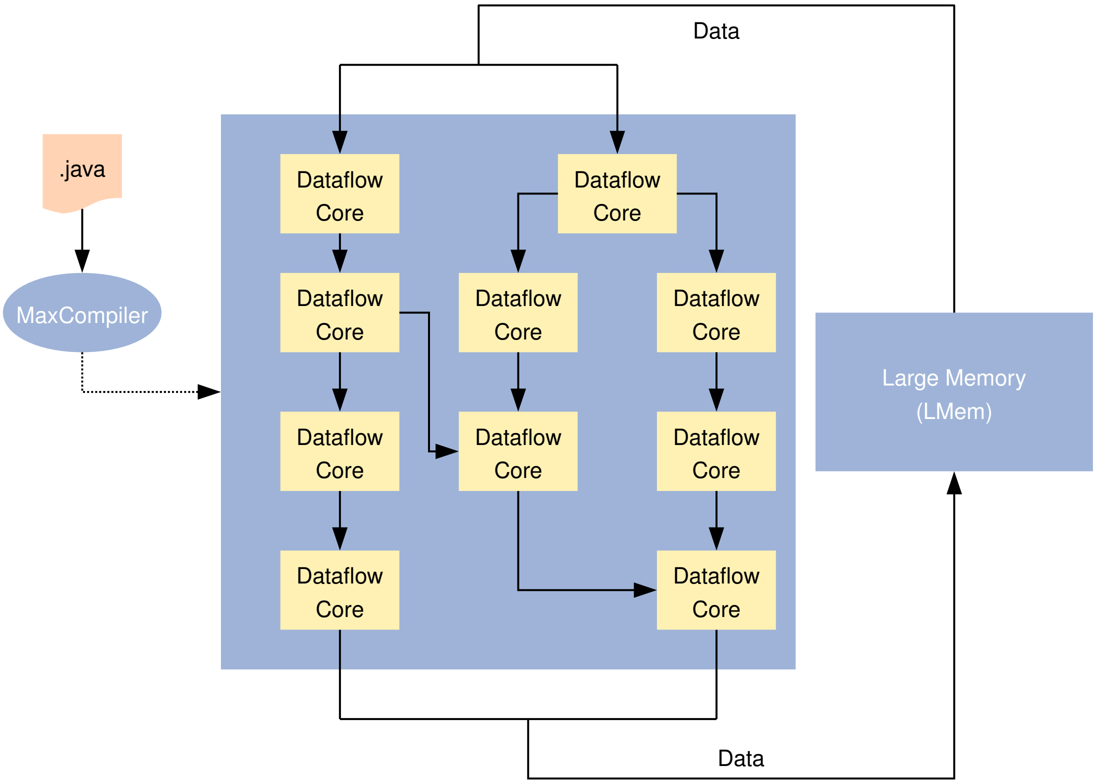


Figure 2: A dataflow program in action.

data processing task. Because there are no instructions there is no need for instruction decode logic, instruction caches, branch prediction, or dynamic out-of-order scheduling. By eliminating the dynamic control flow overhead, the full resources of the chip are dedicated to performing computation. At a system level, the dataflow engine handles computation of large scale data processing while CPUs running Linux manage irregular and infrequent operations, IO and inter-node communication.

1.2 Dataflow engines (DFEs)

Figure 3 illustrates the architecture of a Maxeler dataflow processing system which comprises dataflow engines (DFEs) with their local memories attached by an interconnect to a CPU. Each DFE can implement multiple kernels, which perform computation as data flows between the CPU, DFE and its associated memories. The DFE has two types of memory: **FMem** (Fast Memory) which can store several megabytes of data on-chip with terabytes/second of access bandwidth and **LMem** (Large Memory) which can store many gigabytes of data off-chip.

The bandwidth and flexibility of FMem is a key reason why DFEs are able to achieve such high performance on complex applications - for example a Vectis DFE can provide up to 10.4TB/s of FMem bandwidth within in chip. Applications are able to effectively exploit the full FMem capacity because both memory and computation are laid out in space so data can always be held in memory close to computation. This is in contrast to traditional CPU architectures with multi-level caches where only the smallest/fastest cache memory level is close to the computational units and data is duplicated through the cache hierarchy.

> Effectively exploiting the DFE's FMem is often the key to achieving maximum performance.

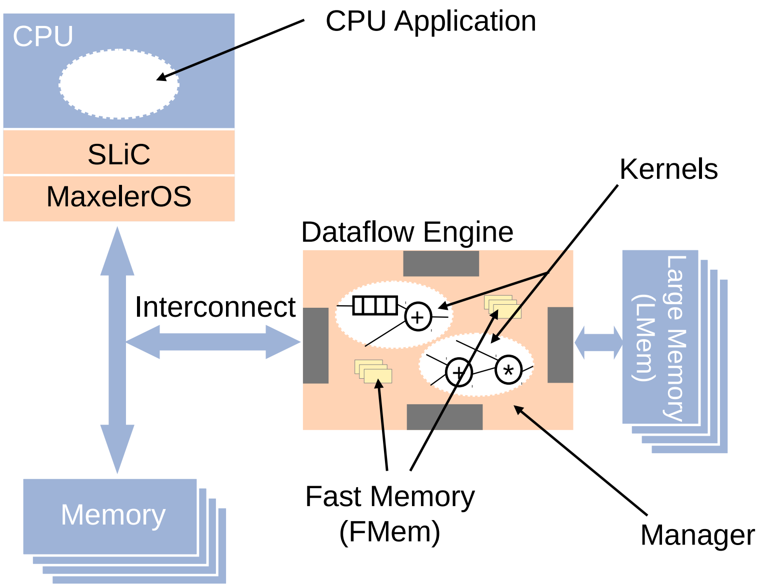


Figure 3: Dataflow engine architecture

The dataflow engine is programmed with one or more **Kernels** and a **Manager**. Kernels implement computation while the Manager orchestrates data movement within the DFE. Given Kernels and a Manager, **MaxCompiler** generates dataflow implementations which can then be called from the CPU via the SLiC interface. The SLiC (Simple Live CPU) interface is an automatically generated interface to the dataflow program, making it easy to call dataflow engines from attached CPUs.

The overall system is managed by MaxelerOS, which sits within Linux and also within the Dataflow Engine's manager. MaxelerOS manages data transfer and dynamic optimization at runtime.

1.3 System architecture

In a Maxeler dataflow supercomputing system, multiple dataflow engines are connected together via a high-bandwidth **MaxRing** interconnect, as shown in *Figure 4*. The MaxRing interconnect allows applications to scale linearly with multiple DFEs in the system while supporting full overlap of communication and computation.

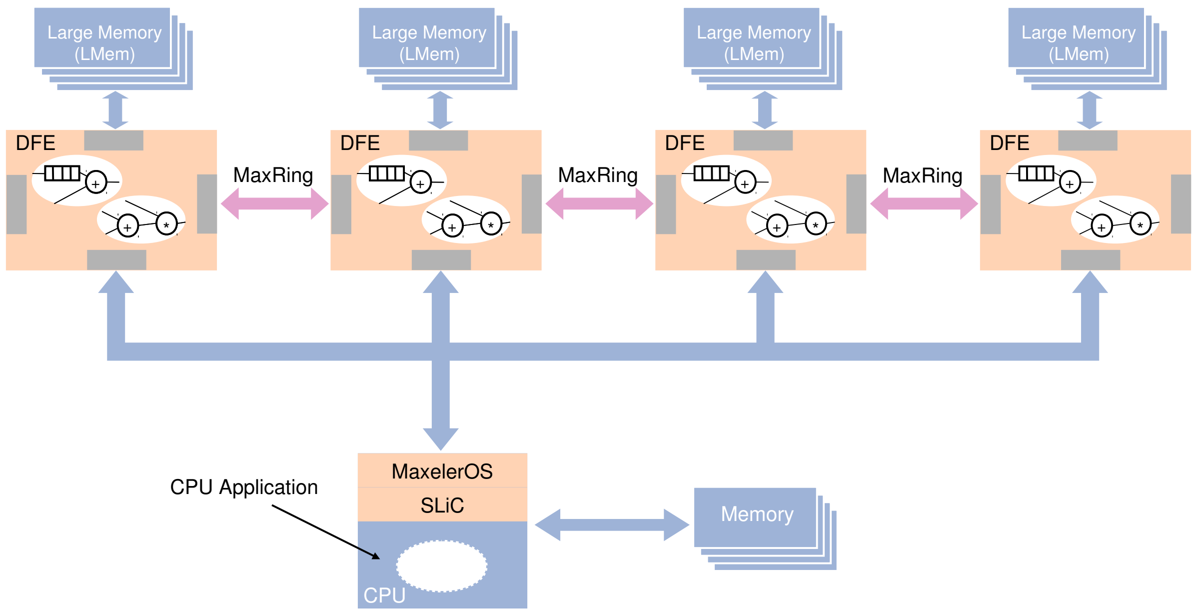


Figure 4: Maxeler dataflow system architecture

2

The Simple Live CPU Interface (SLiC): Using `.max` Files

Everything should be made as simple as possible, but no simpler.

– A. Einstein

A Maxeler dataflow supercomputer consists of CPUs and Dataflow Engines (DFEs). The CPUs run executable files while DFEs run configuration files called `.max` (dot-max) files.

The `.max` file is loaded by a CPU program and runs on an available dataflow engine. MaxelerOS manages the execution at runtime. Calling the Simple Live CPU (SLiC) API functions executes **actions** on the DFE, which include sending data streams and sets of parameters to the DFE.

2.1 A first SLiC example

To see how we can use the SLiC interface to interact with DFEs, we take the example of a three-point moving average `.max` file (we'll see the source code for this `.max` file in the next chapter, *section 3*).

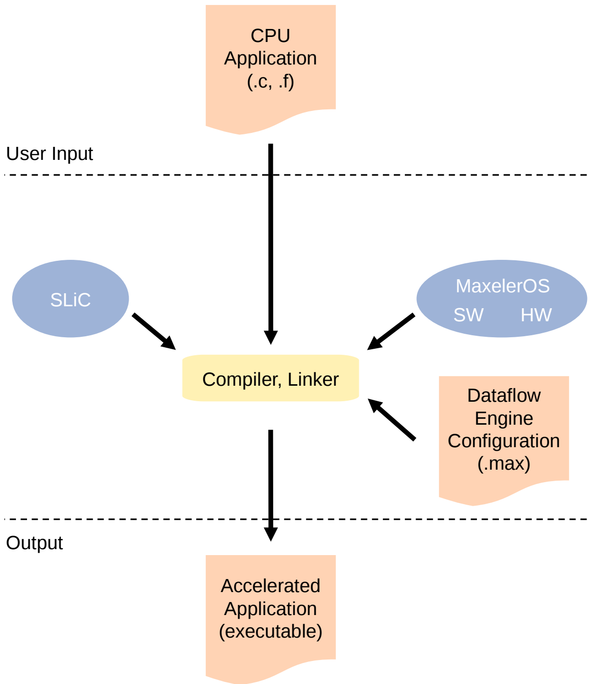


Figure 5: Software component interactions

A C implementation of the moving average would look like this:

```
void MovingAverageCPU(int size, float *dataIn, float *expected) {
    expected[0] = (dataIn[0] + dataIn[1]) / 2;
    for (int i = 1; i < size-1; i++) {
        expected[i] = (dataIn[i-1] + dataIn[i] + dataIn[i+1]) / 3;
    }
    expected[size-1] = (dataIn[size-2] + dataIn[size-1]) / 2;
}
```

The `.max` file for our moving average example has the name `MovingAverage.max` and has a header file `MovingAverage.h`. To use the SLiC functions in your C source code, you can either include the `.max` file itself or include its accompanying header file with the same name, which is smaller and easier to read. *Figure 5* shows the interaction of the various software components to build a program.

The header file shows the functions that are available for a particular `.max` file. SLiC supports multiple levels of interface for interacting with DFEs; the most straightforward SLiC interface is called Basic Static. The Basic Static level interface for this `.max` file has a single function:

```
void MovingAverage(
    int param_N,                /* number of floats in the input stream */
    const float *instream_x,    /* constant input (does not change) */
    float *outstream_y);        /* location of results */
```

This function loads the `.max` file onto an available DFE, streams the input array into the DFE and writes the results into the output array, returning once all the output data is written.

2.2 Using multiple engine interfaces within a .max file

An **engine interface** is a particular way to call a Dataflow Engine. Each engine interface has certain actions it performs. For example, `MovingAverageWeighted.max` adds another engine interface to the weighted average activity. In this second engine interface, you can also set the weights of the weighted average as follows:

```
void MovingAverageWeighted_weighted(
    int param_N,                 /* number of floats in the input and output arrays */
    const float param_weights[3], /* three coefficients */
    const float *instream_x,     /* constant input (does not change) */
    float *outstream_y);         /* location of results */
```

The complete include file for the two engine interfaces is `MovingAverageWeighted.h`.

2.3 Loading and executing .max files

The life-cycle of a `.max` file within a CPU application is as follows:

load - the `.max` file is loaded onto a DFE. The DFE is now exclusively owned by the calling CPU process.

> ✱ Loading the `.max` file takes in the order of 100ms to 1s.

execute actions - the CPU calls SLiC functions to execute actions on the DFE.

> ✱ A loaded `.max` file should be utilized for long enough to justify having waited up to a second to load the configuration.

unload - the DFE is released by the CPU process and returns to the pool of DFEs managed by MaxelerOS.

The Basic Static SLiC interface loads the `.max` file onto the DFE when the first SLiC function is called, and releases the DFE when the CPU program terminates.

2.4 Using multiple .max files

A CPU application can call multiple DFE functions to use multiple `.max` files, either running simultaneously on multiple DFEs or sequentially on the same DFE. This is done by simply including the header files for each `.max` file and calling the appropriate functions for each file.

Using the Basic Static SLiC interface level, each `.max` file is run on a different DFE. For example, imagine that we have our moving average `.max` file and another `.max` file called `Threshold.max` that thresholds its input stream. Running both DFE configurations requires passing the result of the moving average to the thresholding DFE:

```
#include "MovingAverage.h"
#include "Threshold.h"
#include <MaxSLiCInterface.h>
 ...
MovingAverage(size, dataIn, mavOut);
Threshold(size, mavOut, dataOut);
```

To run multiple `.max` files on the same DFE sequentially requires using the Advanced Static level (see *subsection 10.2*).

2.5 SLiC Skins

SLiC Skins allow Basic Static SLiC interface function calls to be made natively in languages other than C. Skins mean that DFE accelerated functions can be quickly integrated into applications/libraries written in the supported languages.

SLiC Skins are generated from `.max` files using the `sliccompile` tool bundled with MaxCompiler. `sliccompile` takes as one of it's arguments the *target* to generate a Skin for. See *Table 1* below for a list of supported targets.

Language	Target	Versions supported
Python	`python`	2.4 – 2.6
MATLAB	`matlab`	R2012b or higher
R	`R`	2.11 or higher

Table 1: Supported Skin Targets

2.5.1 Matlab

The MATLAB Skin uses MATLAB objects to provide a `.max` file's Basic Static SLiC interface's functionality. *Listing 1* shows a call to the Moving Average example from MATLAB.

Listing 1: MATLAB code for executing the Moving Average kernel (MovingAverageDemo.m).

```
m = MovingAverage();
dataOut = m.default([1, 0, 2, 0, 4, 1, 8, 3]);
disp(dataOut(2:7));
```

To create MATLAB bindings for the moving average `.max` file run the following command:

```
[user@machine]$ sliccompile -t matlab -m MovingAverage.max
```

This creates `mex_MovingAverage.mexa64`, `MovingAverage.m` and the `simutils` directory which together comprise the `MovingAverage` MATLAB toolbox. When MATLAB is started from a directory containing these files the `MovingAverage` class is made available in the environment. Output arguments appear on the left-hand side of method calls.

Run the following command to execute the MATLAB script shown in *Listing 2*:

```
[user@machine]$ matlab movingaverage.m
```

Listing 2: MATLAB code for executing the Moving Average kernel (MovingAverageDemo.m).

```
m = MovingAverage();
dataOut = m.default([1, 0, 2, 0, 4, 1, 8, 3]);
disp(dataOut(2:7));
```

The MATLAB binding creates access to Basic Static SLiC interface functions through a class named with the maxfile name. First create an instance of this class.

```
m = MovingAverage ();
```

This instance, `m`, has methods representing basic Static SLiC interface calls that are now available through MATLAB. These calls keep their original names. Argument names also stay the same but output arguments do not need to be passed as function arguments. They instead become values returned by the functions. Some functions may return more than one item. E.g. if a Basic Static SLiC interface function `doSomething` takes an argument `a` and returns arguments `b` and `c` they can be accessed as follows:

```
[b, c] = m.doSomething(a);
```

All method documentation is available through MATLAB's online help system. For help on a function `doSomething` belonging to the `MovingAverage.max` file, run

```
help ('MovingAverage.doSomething')
```

and all input and output arguments are described.

Once the object is finished with it can be removed by running the following.

```
clear m;
```

Doing this ensures all DFE connections are closed and that memory is freed.

2.5.2 Python

The Python Skin works like a normal Python module. The example in *Listing 3* calculates the moving average of a Python list. If NumPy is installed then NumPy arrays can be used instead of Python lists.

To create Python bindings for the moving average `.max` file run the following command:

```
[user@machine]$ sliccompile -t python -m MovingAverage.max
```

Listing 3: Python code for executing the Moving Average kernel (MovingAverageDemo.py).

```
from MovingAverage import MovingAverage
dataOut = MovingAverage([1, 0, 2, 0, 4, 1, 8, 3])
for i in range(len(dataOut))[1:-1]:
    print "dataOut[%d] = %f" % (i, dataOut[i])
```

This creates two files, `MovingAverage.py` and `_MovingAverage.so`, and one directory named `simutils`. They encompass the Python module `MovingAverage` and must be kept together. To add the module to Python's search path start Python from the directory containing the module's files.

The following command executes the Python program in *Listing 4*:

```
[user@machine]$ python movingaverage.py
```

Listing 4: Python code for executing the Moving Average kernel (MovingAverageDemo.py).

```
from MovingAverage import MovingAverage
dataOut = MovingAverage([1, 0, 2, 0, 4, 1, 8, 3])
for i in range(len(dataOut))[1:-1]:
    print "dataOut[%d] = %f" % (i, dataOut[i])
```

Once the Python module search path is set appropriately the module can be imported into Python like any other module with the command:

```
import MovingAverage
```

where MovingAverage is the .max file name. When running a simulation Python must be launched with the generated simutils directory in the current working directory.

Online documentation is available and can be viewed for the module, `MovingAverage`, by running

```
help(MovingAverage)
```

All .max file constants belong to the imported module object and have the same name as defined in the engine interface. Basic Static SLiC functions are made available as functions that can be called in the imported module and keep their original defined names. The online documentation lists the method signatures for each of these functions. The function arguments have the same name but output arguments appear on the left-hand side of functions as return arguments. Where a function has more than one output argument the results are returned as a tuple. Streams in Python skin interfaces can be supplied in the form of nested Python lists or as NumPy arrays.

Nested Python Lists Python lists are suitable for small tests, quick prototypes or demos. They are easy to use and an attempt is made to do as much run-time checking as possible. They are not appropriate for high performance code but can be used for simple prototyping.

NumPy Arrays NumPy arrays should be used for high performance code. All NumPy array element types are typed and types must match the Engine interface requirements exactly. Element types of function arguments are specified in the online documentation. When using NumPy arrays it is important to pass arrays of C-style contiguous memory. Arrays not in this format will work but the interface may be considerably slower. These issues are covered in more detail below.

2.5.3 R

The **R** Skin is installed into R as a library and data is provided using R vectors or arrays. The moving average example called from R is shown in *Listing 5*.

Listing 5: R code for executing the Moving Average kernel (MovingAverageDemo.R).

```
library("MovingAverage")
dataOut <- MovingAverage(c(1, 0, 2, 0, 4, 1, 8, 3))
for (i in 2:7)
    cat('o[', i, '] =', dataOut[i], '\n')
```

To create R bindings for the moving average .max file run the following command:

```
[user@machine]$ sliccompile -t R -m MovingAverage.max
```

This creates `MovingAverage_0.1-1_R_x86_64-redhat-linux-gnu.tar.gz` which is an R package and a `simutils` directory. To install it run:

```
[user@machine]$ R CMD INSTALL -l . MovingAverage_0.1-1_R_x86_64-redhat-linux
    -gnu.tar.gz
```

This directory must then be added to R's library search path:

```
[user@machine]$ export R_LIBS="$(pwd):$R_LIBS"
```

The library can now be imported into R and run from R.

```
[user@machine]$ R --no-save < movingaverage.R
```

Listing 6: R code for executing the Moving Average kernel (MovingAverageDemo.R).

```
library("MovingAverage")
dataOut <- MovingAverage(c(1, 0, 2, 0, 4, 1, 8, 3))
for (i in 2:7)
    cat('o[', i, '] =', dataOut[i], '\n')
```

Once in the R environment and assuming the generated library (`MovingAverage`) has been made available to R, it can be imported with the following command.

```
library(MovingAverage)
```

This imports the MovingAverage namespace into the environment. All basic Static SLiC interface functions keep their original names and are imported under this namespace. These can either be called directly or called through the namespace. E.g. the moving average function can be called as `MovingAverage` or as `MovingAverage::MovingAverage`.

Online documentation is available for all R packages though the help command.

```
help(MovingAverage)
```

Function argument names stay the same but output arguments do not need to be passed as function arguments. They instead become values returned by the functions. When the SLiC function returns more than one item R gets the items as a list. E.g. if a Basic Static SLiC interface function `F` takes an argument `a` and returns arguments `b` and `c` they can be accessed as follows:

```
ret_list <- MovingAverage::F(a)
result_b <- ret_list$b
result_c <- ret_list$c
```

When testing a simulation `.max` file it is necessary to start and stop a simulator within R. To start the simulator call `startSimulator` and to stop the simulator call `stopSimulator`. Both of these functions are exposed as part of the generated R library.

2.5.4 Skin Target Summary

In addition to the language bindings `sliccompile` will also generate a `simutils` directory. This directory MUST be copied into the directory the R, MATLAB or Python process is started from to use simulation `.max` files.

All three language bindings allow interaction with DFEs by script or through interactive use. They all come with auto-generated documentation and have simpler interfaces than C taking advantage of high-level features of these languages. Details of how to build and use the language bindings can be found in *subsection 10.14*.

2.5.5 Installer bindings

Bindings can be distributed as installer files. Installer files generate the language bindings for the skins user. *Listing 7* shows a more complex Python example for a non-central χ^2 random number generator. The Python code to interact with the DFE is simple allowing the application writer to concentrate on the application itself.

To import a generated Python interface use the maxfile name as the module name. All Basic Static SLiC interface names stay the same. Functions can be imported like any other Python function.

```
from NCChiSquare import NCChiSquare
```

The call to the generated Python interface for the .max file is the following simple line.

```
dfeRes = NCChiSquare(degree, outputCount, lambdaVal)
```

All other code in the listing relates to timing, result checking and graph plotting.

To unpack the demo and binding run

```
[user@machine]$ ./NCChiSquare_installer -t python
```

The demo can then be executed by running

```
[user@machine]$ python NCChiSquareDemo.py
```

Figure 6 shows a screenshot of the demo application.

Listing 7: Python code for executing a random number generator from Python.

```
from NCChiSquare import NCChiSquare
import time
from ncchisquaremisc.ncchisquaremisc import *
from ncchisquaremisc.NCChiSquareCPP import NCChiSquareCPP

## Random Number Generator Parameters ##
degree = 160
outputCount = 1000000
lambdaVal = 1.0

## CPU Run ##
print "Running CPU version"
ts = time.time()
cpuRes = NCChiSquareCPP(degree, outputCount, lambdaVal)
te = time.time()
cpuTime = te - ts
print 'CPU run took: %2.4f sec' % (cpuTime)

## DFE Run ##
print "Running DFE version"
ts = time.time()
dfeRes = NCChiSquare(degree, outputCount, lambdaVal)
te = time.time()
dfeTime = te - ts
print 'DFE run took: %2.4f sec' % (dfeTime)

## Check Results ##
if not checkResults(cpuRes, dfeRes):
    print "Error: Results no not match"

## Plot graph ##
ncchiGraph(
    'Non-central chi squared distibution random number generator frequency distributions',
    outputCount,
    'DFE Implementation (%f seconds)' % dfeTime,
    dfeRes,
    'CPU Implementation (%f seconds)' % cpuTime,
    cpuRes,
    degree,
    lambdaVal
)
```

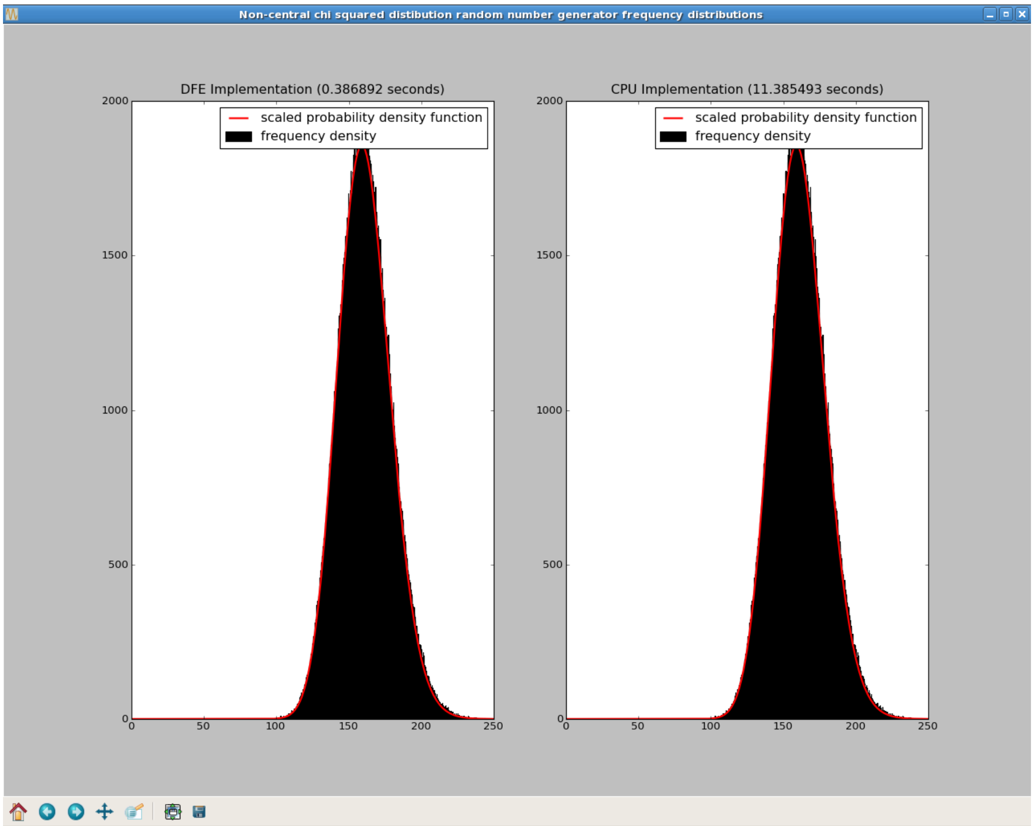


Figure 6: Non-central χ^2 random number Generation demo

2.6 SLiC Interface levels

Overall, SLiC functionality can be accessed on three levels:

Basic Static allows a single function call to run the DFE using static actions defined for the particular `.max` file.

Advanced Static allows control of loading of DFEs, setting multiple complex actions, and optimization of CPU and DFE collaboration.

Advanced Dynamic allows for the full scope of dataflow optimizations and fine-grained control of allocation and de-allocation of all dataflow resources.

The advanced SLiC interfaces are described in *section 10*.

> Non-blocking functions for all of the SLiC functions to run actions on the DFE are also available for all levels of the SLiC interface.

3

Dataflow Programming: Creating `.max` Files

> *I must create a system or be enslaved by another man's; I will not reason and compare: my business is to create.*
>
> – William Blake

A dataflow application consists mostly of CPU code, with small pieces of the source code, and large amounts of data, running on dataflow engines. We use a Java library to describe the code that runs on the dataflow engine.

We create dataflow implementations (`.max` files) by writing Java code and then executing the Java code to generate the `.max` file which can then be linked and called via the SLiC interface. A `.max` file generated by MaxCompiler for Maxeler DFEs comprises of two decoupled elements: **Kernels** and a **Manager**. **Kernels** are graphs of pipelined arithmetic units. Without loops in the dataflow graph, data simply flows from inputs to outputs. As long as there is a lot more data than there are stages in the pipeline, the execution of the computation is extremely efficient. With loops in the dataflow graph, data

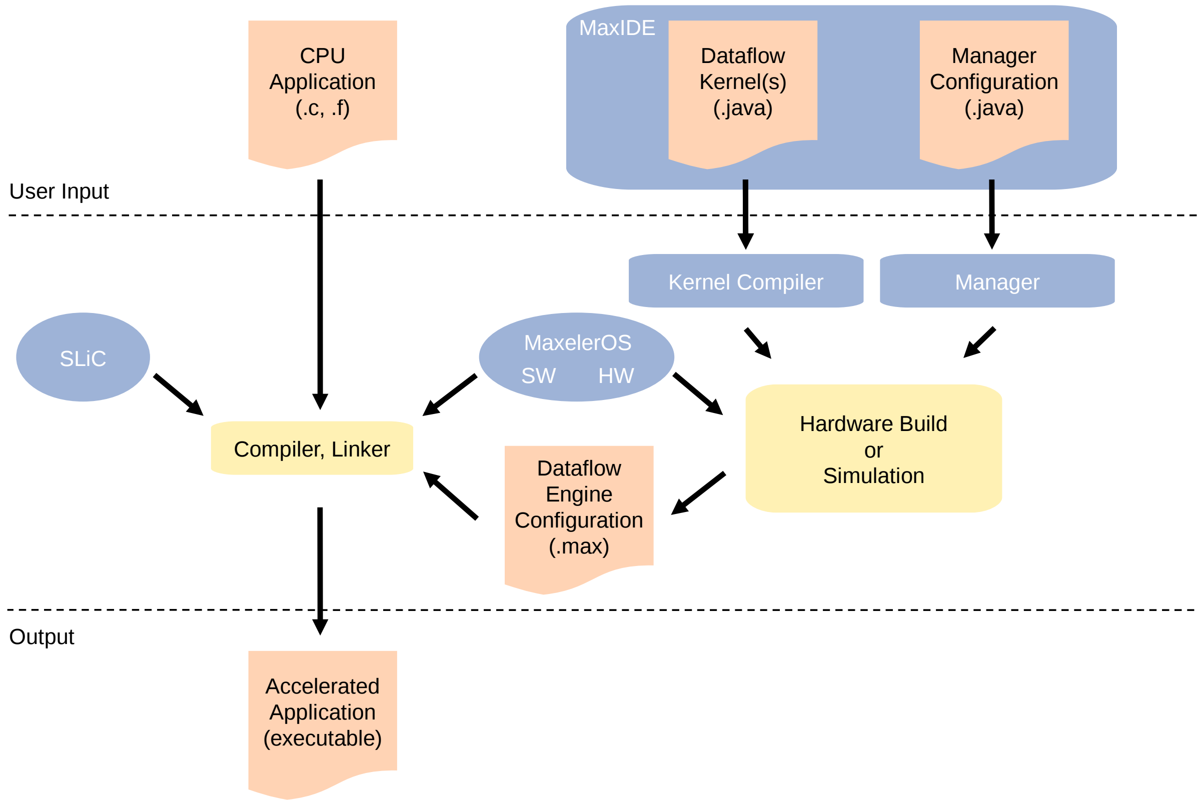


Figure 7: MaxCompiler component interactions (blue-gray objects denote MaxCompiler components)

flows in a physical loop inside the DFE, in addition to flowing from inputs to outputs.

The **Manager** describes the data flow choreography between Kernels, the DFE's memory and various available interconnects depending on the particular dataflow machine. By decoupling computation and communication, and using a flow model for off-chip I/O to the CPU, DFE interconnects and memory, Managers allow us to achieve high utilization of available resources such as arithmetic components and memory bandwidth. Maximum performance in a Maxeler solution is achieved through a combination of deep-pipelining and exploiting both inter- and intra-Kernel parallelism. The high I/O-bandwidth required by such parallelism is supported by flexible high-performance memory controllers and a highly parallel memory system.

MaxCompiler and MaxIDE use an extended version of Java called MaxJ which adds operator overloading semantics to the base Java language, enabling an intuitive programming style. MaxJ source files have the `.maxj` file extension to differentiate them from pure Java .

Figure 7 shows the development tools provided by MaxCompiler and how they interact to build an accelerated application.

Figure 8 shows the design flow for implementing a dataflow configuration using MaxCompiler. The next subsections describe each of these stages in detail.

3.1 Identifying areas of code for dataflow engine implementation

Traditionally, the first step is to analyze the application source code to determine which parts of the code should be implemented in a dataflow engine. For Multiscale Dataflow Computing, the data is more important than the source code. Thinking about moving data to the DFE is a much better first

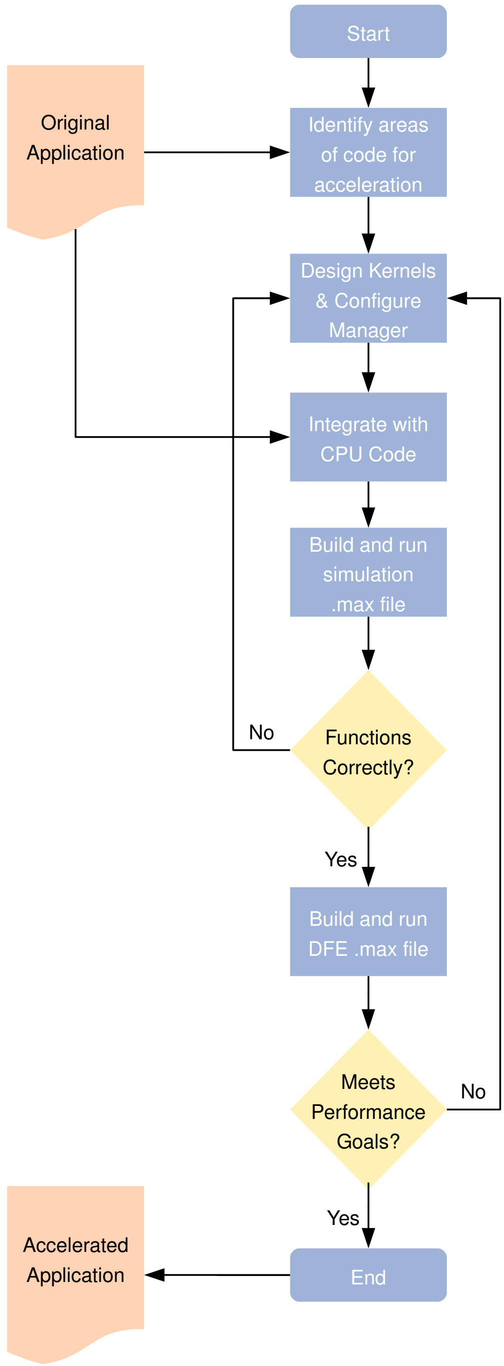


Figure 8: Diagram of the design flow

step. Once we know which data is on the DFE at which point in time, it is obvious which pieces of code need to run on the DFE as well. Of course in reality this is typically an iterative process.

- The first step in creating a Multiscale Dataflow program is to measure how long it takes to run the application on CPUs given a set of representative (large) datasets. Limiting the analysis to toy inputs is a waste of time since CPU memory systems do not scale linearly with problem size and dataflow technology is targeting large datasets.
- Next, a more detailed analysis provides the distribution of runtime of various parts of the application including, if possible, an analysis of time spent in computation and time spent in communication. Most of the analysis can be achieved with time counters and profiling tools such as gprof, oprofile, valgrind etc.

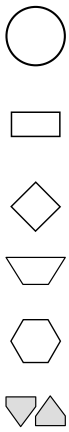
> Acceleration is not limited to the percentage of the application that is being accelerated because in real-world application development, a lot of programmer effort is spent in optimizing the core loops while very little effort is spent on optimizing the non-critical pieces of the application. Once the critical loops are accelerated and moved away from the CPUs memory system, it is typically possible to accelerate the non-critical code on the CPU and balance the execution time on the DFE and CPUs to maximize performance by maximizing utilization of all resources in the Multiscale Dataflow Computer.

- Maximizing regularity of computation: Dataflow engines operate best when performing the same operation repeatedly on many data items, for example, when computing an inner loop. To maximize regularity it is imperative to consider all possible loop transformations and estimate performance of dataflow implementations for each case.
- Minimizing communication between CPU and dataflow engines: Sending/receiving data between the CPU and dataflow engines is, relatively speaking, expensive since communication is usually slower than computation. By carefully selecting the parts of the application to implement in a dataflow engine, we strive to overlap communication over the CPU-DFE interconnect with computations on both DFEs and CPUs.

The computation-to-data ratio, which describes how many mathematical operations are performed per item of data moved, is a key metric for estimating the performance of the final dataflow implementation. Code that requires large amounts of data to be moved and then performs only a few arithmetic operations poses higher balancing challenges than code with significant localized arithmetic activity.

3.2 Implementing a Kernel

In this section, we will take a detailed look at a Kernel and the implementation of the arithmetic needed within an algorithm. The resulting graphs of arithmetic units are the implementation of the data flow shown in *Figure 2* in *subsection 1.1*. Kernel graphs contain a variety of different node types:



Computation nodes perform arithmetic and logic operations (e.g., +,*,<,&) as well as type casts to convert between floating point, fixed point and integer variables.

Value nodes provide parameters which are either constant or set by the CPU application at run-time.

Stream offsets allowing access to past and future elements of data streams.

Multiplexer (mux) nodes for taking decisions.

Counter nodes for directing control flow over time, for example, keeping track of the position in a stream for boundary calculations.

I/O nodes connecting data streams between Kernel and Manager.

Let's consider a simple moving average example such as the one we called in the previous section via the SLiC interface. The application computes a 3-point moving average over a sequence of N data values. At the boundaries (the beginning and the end of the data sequence), 2-point averages need to be applied. The moving average can be expressed as:

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

In a software implementation, an array would be used to hold the data and would be scanned through with a loop to compute the 3-point average for each index. The array boundaries would be checked specifically and 2-point averages computed at these positions:

```
void MovingAverageSimpleCPU(int size, float *dataIn, float *expected) {
    expected[0] = (dataIn[0] + dataIn[1]) / 2;
    for (int i = 1; i < size -1; i++) {
        expected[i] = (dataIn[i -1] + dataIn[i] + dataIn[i + 1]) / 3;
    }
    expected[size -1] = (dataIn[size -2] + dataIn[size -1]) / 2;
}
```

The complete Java source for the implementation of this Kernel with its corresponding graph is shown in *Figure 9*. The arrows in the diagram show which lines of Java code generated which nodes in the graph. The data flows from the input through the nodes in the graph to the output.

The first step in creating a dataflow kernel is to declare an input stream of the required type, in this case a C float type (8-bit exponent and a 24-bit mantissa):

```
        DFEVar x = io.input("x", dfeFloat(8, 24));
```

Array accesses turn into accesses into a stream of data. Thus the indices of i, $i-1$, and $i+1$ become the current, previous and next values in the input stream.

```
        DFEVar prev = stream.offset(x, -1);
        DFEVar next = stream.offset(x, 1);
```

The average of these three values can now be calculated:

```
        DFEVar sum = prev + x + next;
        DFEVar result = sum / 3;
```

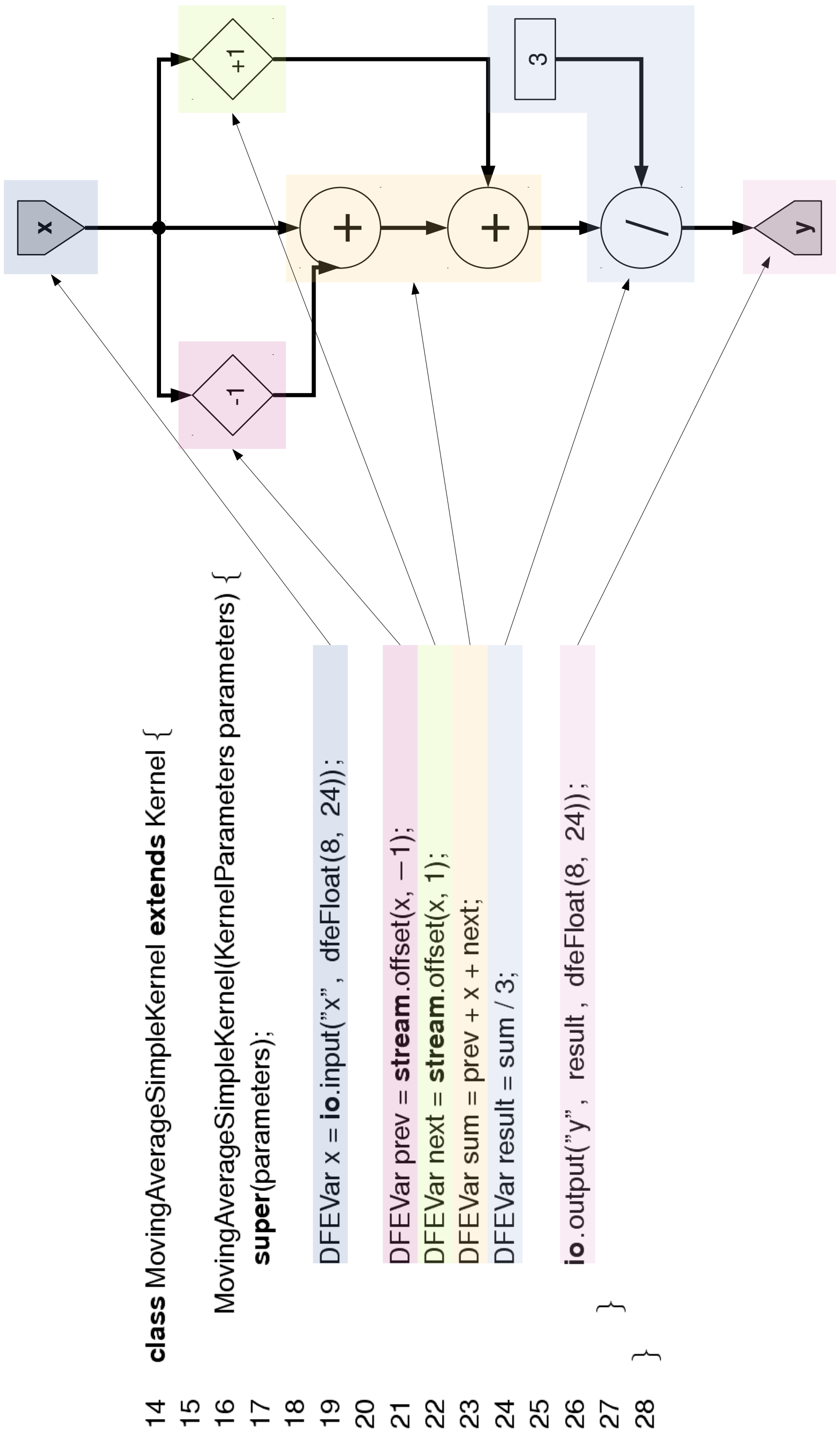


Figure 9: Source code for the simple moving average Kernel with the corresponding Kernel graph diagram.

Finally the result is written into an output stream:

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

To demonstrate the streaming of data over time through the Kernel, *Figure 10* shows a stream of six values passing through the Kernel. Labels have been added to show the value along the edges in the graph. This is the **programmer's view** of the data passing through the Kernel, where the actual pipelined operation of the Kernel is not considered.

> During one unit of time called a **tick**, the Kernel executes one step of the computation, consumes one input value and produces one output value.

Figure 11 shows the same six values passing through the Kernel, but this time showing how the kernel actually runs within the dataflow engine as a pipeline. This diagram makes the simplification that each node in the graph takes a single clock cycle to produce a result, which may not always be the case, but demonstrates the principle. The graph is labeled in gray with the filling stages, when it produces no output, and the flushing stages, when it continues to produce output but consumes no input. The related data in the graph appears in the same color to show its progress through the pipeline.

This pipelined style of computation is key to the performance of dataflow engines, since all operations can be computing in parallel on different data items within the stream. MaxCompiler automatically manages pipelining of the kernel so the programmer does not generally need to consider individual latencies within the pipeline but instead can program using the abstracted view of *Figure 10*.

3.3 Estimating performance of a simple dataflow program

One key advantage of dataflow computing is that we can estimate the performance of the dataflow implementation before actually implementing it, thanks to the static scheduling. For the three-point moving average filter above, the time to filter 1 million numbers, T, is the time for 1 million numbers to flow through the three-point filtering dataflow graph.

The first component in estimating performance in dataflow computation is the bandwidth in and out of the dataflow graph. For data in DFE memory, we simply look up the bandwidth of the particular device and memory storing the data. The second component is the speed at which the dataflow pipeline is moving the data forward. A unit of time in a DFE is called a tick, and the speed of movement through a dataflow pipeline is given in [ticks/second].

$T = min(bandwidth, computefrequency) \times 1M$. Bandwidth can be thought of as the "numbers per second" that can be read into or written out from the DFE chip. The *computefrequency* is how many ticks per second the kernels can run at. The frequency is between 100-300 million ticks per second as determined and displayed during the DFE compilation process, while bandwidth of DFEs can be between 200-1000 million numbers per second depending on the size of the numbers and the speed of the interconnect (LMEM, PCIe, Infiniband, or MaxRing).

> The performance of the three-point filter does not depend on the computations. A 100-point filter runs as fast as a three point filter, as long as it fits within the resources available on the DFE. This is the essence of "computing in space" compared to "computing in time".

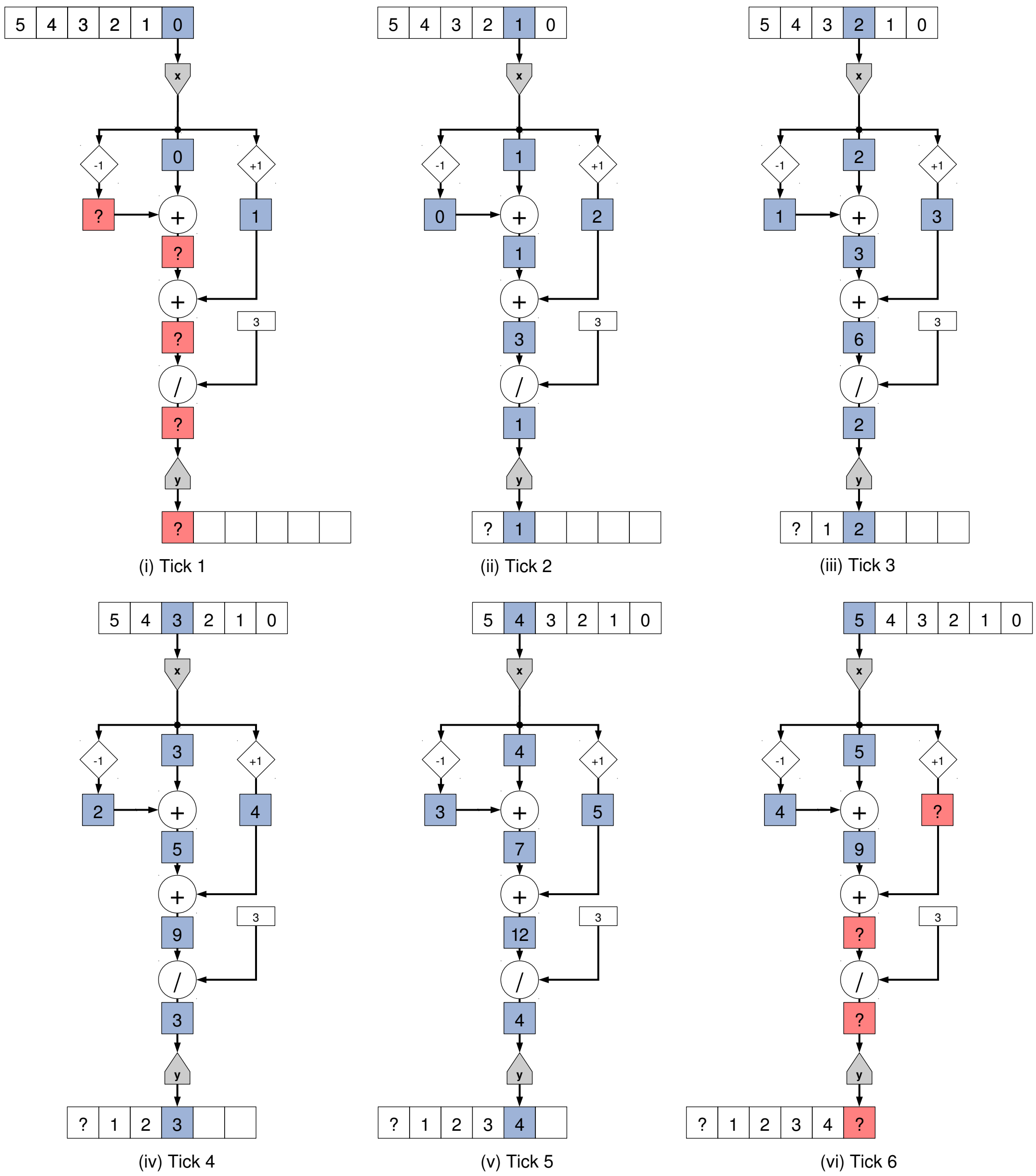


Figure 10: Programmer's view of the simple moving average Kernel over six ticks showing the input and output streams.

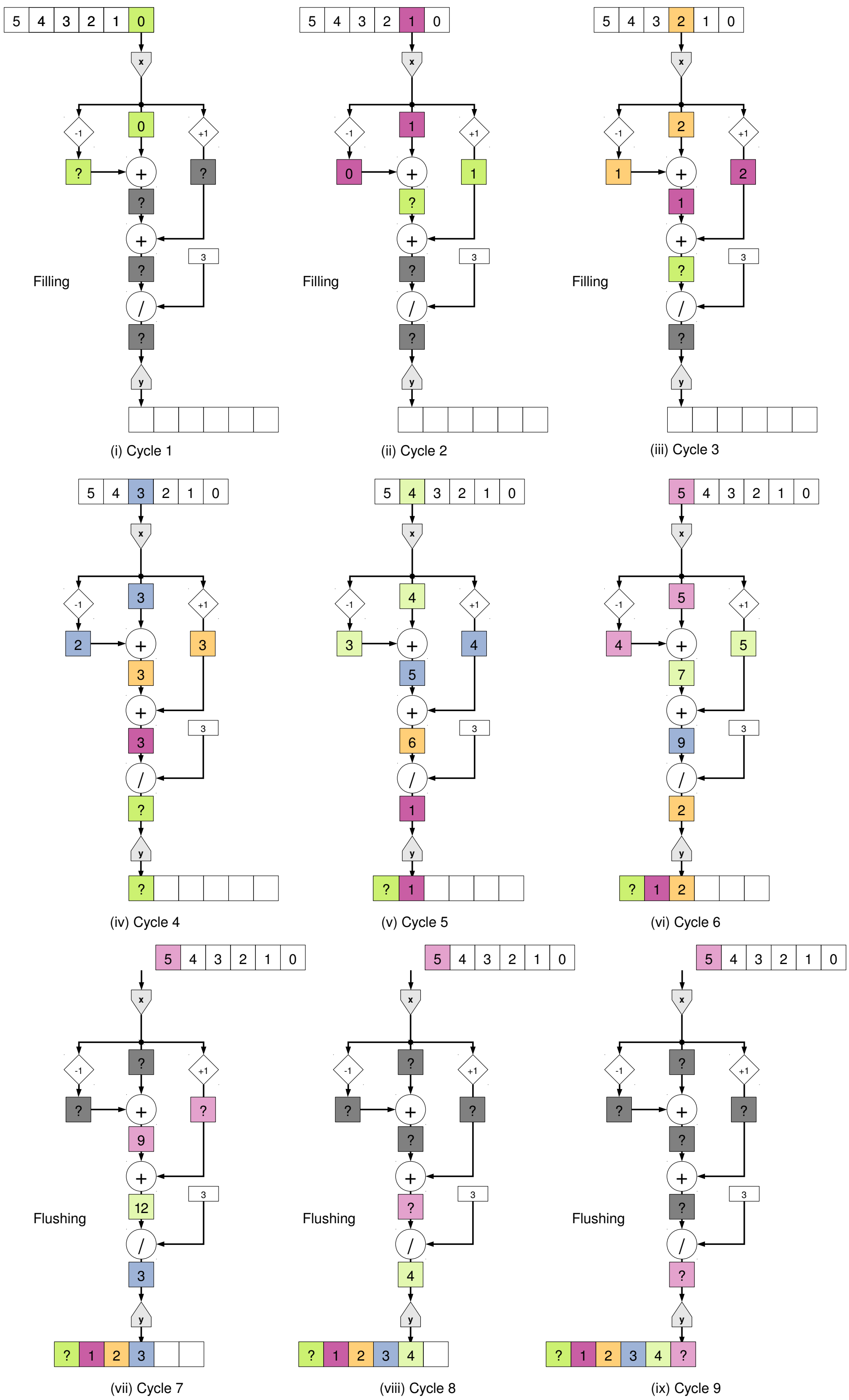


Figure 11: Pipelined view of simple moving average Kernel over nine clock cycles.

3.4 Conditionals in dataflow computing

There are three main methods of controlling conditionals that affect dataflow computation:

1. Global conditionals: These are typically large scale modes of operation depending on input parameters with a relatively small number of options. If we need to select different computations based on input parameters, and these conditionals affect the dataflow portion of the design, we simply create multiple `.max` files for each case. Some applications may require certain transformation to get them into the optimal structure for supporting multiple `.max` files.

   ```
   if (mode==1) p1(x); else p2(x);
   ```

   where `p1` and `p2` are programs that use different `.max` files.

2. Local Conditionals: Conditionals depending on local state of a computation.

   ```
   if (a>b) x=x+1; else x=x−1;
   ```

   These can be transformed into dataflow computation as

   ```
   x = (a>b) ? (x+1) : (x−1);
   ```

3. Conditional Loops: If we do not know how long we need to iterate around a loop, we need to know a bit about the loop's behavior and typically values for the number of loop iterations. Once we know the distribution of values we can expect, a dataflow implementation pipelines the optimal number of iterations and treats each of the block of iterations as an *action* for the SLiC interface, controlled by the CPU (or some other kernel).

The ternary-if operator (`?:`) selects between two input streams. To select between more than two streams, the `control.mux` method is easier to use and read than nested ternary-if statements.

Figure 12 shows a more complex three-point average Kernel where the boundary cases are taken into consideration.

At these boundary cases, we need to calculate the average of only two inputs. However, we cannot conditionally create a 2-input or 3-input average depending on the current position in the stream at run-time: we must instantiate any adders and dividers at compile-time to have them implemented in the logic of the dataflow engine. At run-time, we can choose which inputs to use for our adders and dividers to get the correct average.

In order to deal with boundaries, *Figure 12* shows how the operands are provided by *conditional assignments*, which are driven by a conditional expression using the ternary `if` operator (`?:`). One of the operands to the addition comes from a conditional assignment which selects between the previous stream value and the constant zero. Another operand is provided by the other conditional assignment which selects between the next stream value and the constant zero:

```
DFEVar prev = aboveLowerBound ? prevOriginal : 0;
DFEVar next = belowUpperBound ? nextOriginal : 0;
```

The third operand is always the current stream value. A final conditional assignment selects between a constant divisor 3 and a constant divisor of 2, depending on whether the current location is at the boundary or not.

The left-hand part of the Kernel graph in *Figure 12* shows the control logic to decide if we are at the boundary or not. We keep track of the stream position via a position counter. The method `control.count.simpleCounter` creates a counter and takes two parameters: the bit-width of the counter and maximum value (in this case, `size`):

```
        DFEVar count = control.count.simpleCounter(32, size);
```

The output of this counter is a stream of integer values. The counter is initialized to zero when the first input data value `x` enters the Kernel and is subsequently incremented for every newly arriving input data value.

A counter in a dataflow program is equivalent to a loop variable in CPU code.

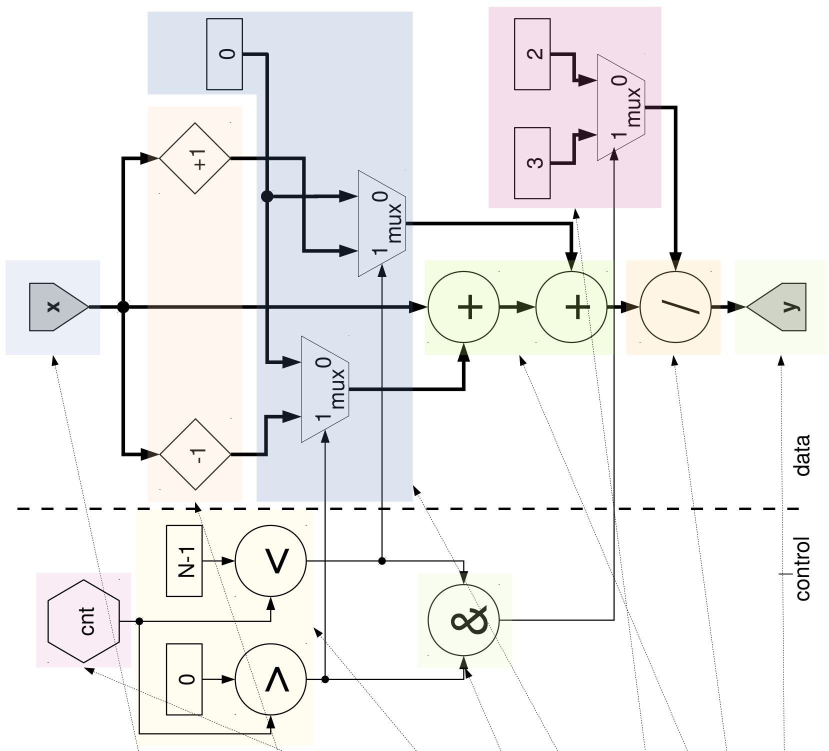
We can use standard relational and logical operators such as <, > and `&` to compute Boolean flags for the control input of a conditional assignment. In our case above, we compute a flag for the lower boundary, a flag for the upper boundary ($i < N{-}1$) and a flag for being in-between the two boundaries:

```
        DFEVar aboveLowerBound = count > 0;
        DFEVar belowUpperBound = count < size - 1;

        DFEVar withinBounds = aboveLowerBound & belowUpperBound;
```

Finally, we calculate the average using the standard operators ($+$ and $/$):

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




```
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));
    }
}
```

Figure 12: Source code for a moving average Kernel that handles boundary cases with the corresponding Kernel graph diagram

3.5 A Manager to combine Kernels into a DFE

Once we have our Kernels, we need to put them together in a Manager. MaxCompiler includes a number of parameterizable Managers, some of which are general purpose while others connect Kernels together in optimal configurations common for specific application domains.

Once the code for Kernels and the configuration of a Manager are combined they form a complete dataflow program. The execution of this program results in either the generation of a dataflow engine configuration file (`.max` file), or the execution of a DFE simulation. In either case, MaxCompiler always generates an include file to go with a `.max` file.

3.6 Compiling

There are several stages to compilation in MaxCompiler as a result of being accessed as a Java library:

1. As the Kernel Compiler and the Managers are implemented in Java, the first stage is **Java compilation**. In this stage the MaxCompiler Java compiler is used to compile user code with normal Java syntax checking etc. taking place.

2. The next stages of dataflow compilation take place at **Java run-time** i.e. the compiled Java code (in .class files) is executed. This process encapsulates the following further compilation steps:

   (a) **Graph construction**: In this stage user code is executed and a graph of computation is constructed in memory based on the user calls to the Kernel Compiler API.

   (b) **Kernel Compiler compilation**: After all the user code to describe a design has been executed the Kernel Compiler takes the generated graph, optimizes it and converts it into either a low-level format suitable for generating a dataflow engine, or a simulation model.

   (c) **Back-end compilation**: Generating DFE configurations including third-party tools to generate the configuration files for the chip.

3.7 Simulating DFEs

Kernels and entire DFE programs can be created in a trial-and-error programming model by using Maxeler DFE simulation. The simulator offers visibility into the execution of a Kernel and compiles in minutes rather than hours for building DFE configuration. The simulation of a DFE program runs much more slowly than a real implementation, so that it makes sense to first run small inputs on simulated DFEs and then run large inputs on actual DFEs.

3.8 Building DFE configurations

Executing a Manager results in the generation of a dataflow engine configuration file with a `.max` extension. This file contains both data used to configure the dataflow engine and meta-data used by software to communicate with this specific dataflow engine configuration. MaxCompiler automates the running of third-party tools to create this configuration seamlessly. This build process can take many hours for a complex design, so simulation is recommended for early verification of the design.

4

Getting Started

All truth passes through three stages: First, it is ridiculed. Second, it is violently opposed. Third, it is accepted as being self evident.

– Schopenhauer

This section takes you through a step-by-step process to write your own dataflow program in MaxIDE, the Maxeler development environment, based on the Eclipse open source platform. In the process we will be creating Kernel designs, configuring Managers, building `.max` files for simulation and DFEs, and programming the CPU application software using the SLiC Interface.

4.1 Building the examples and exercises in MaxIDE

To launch MaxIDE, enter the command `maxide` at a shell command prompt. *Figure 13* shows an excerpt of the welcome page displayed when MaxIDE is launched.

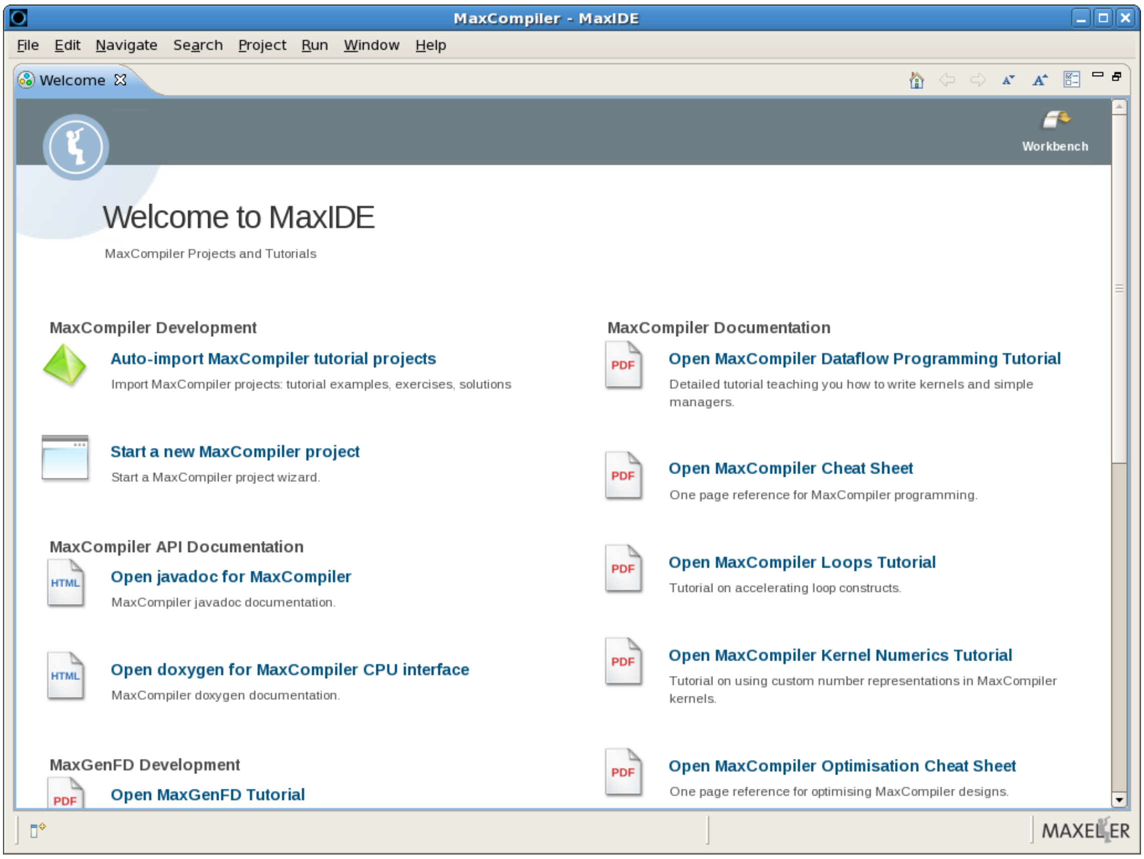


Figure 13: MaxIDE Welcome Page

4.1.1 Import wizard

To work through the examples and exercises, you can import the project source code into MaxIDE. Click on *Auto-import MaxCompiler tutorial projects* on the welcome page. This brings up the import wizard shown in *Figure 14*, which shows a list of project file hierarchies.

Each hierarchy listed in the import wizard dialog box corresponds to a particular tutorial document. The most important tutorials for new users are pre-selected. You can unselect any of these or choose additional selections using the check boxes. You can also click on the arrow to the left of each selection to expand it. Expanding a tutorial hierarchy reveals up to three children, namely `examples`, `exercises`, and `solutions`:

- `examples` contains complete projects suitable for building just as they appear in the tutorial.
- `exercises` contains partially written projects for you to finish as suggested in the tutorial.
- `solutions` are completed versions of the exercises for you to get help or to check your work.

Each of these can be further expanded to a list of projects, which allows you to import individual projects.

4.1.2 MaxCompiler project perspective

Click on the *Finish* button to import the source code and all supporting material for your selections.

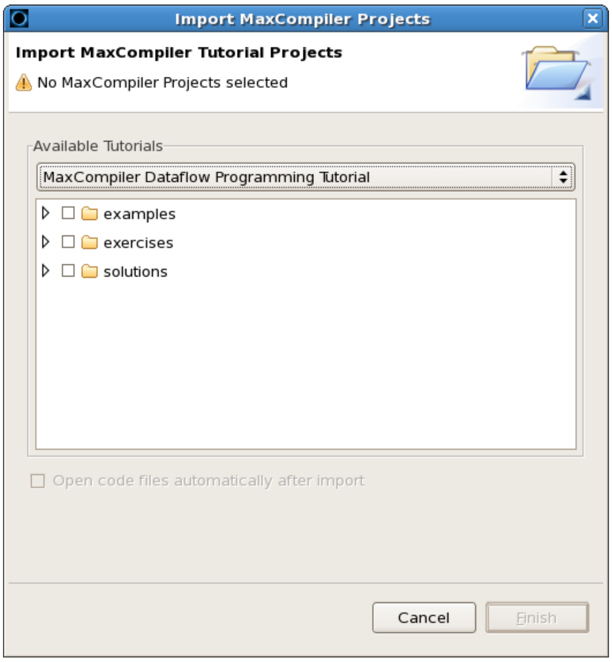


Figure 14: Screen shot of the project import wizard

When the import is complete, MaxIDE switches to the MaxCompiler Project perspective, with an appearance similar to *Figure 15*.

> You can return to the welcome page at any time by clicking on the *Help* menu at the top and selecting the *Welcome* option from the drop-down menu.

The *Project Explorer* panel on the left has a heading for each of the projects you imported. Each project can be expanded to show the three subheadings of *CPU Code*, *Engine Code*, and *Run Rules*.

- Navigating further below the *CPU Code* or *Engine Code* headings leads to individual C, C++, or MaxJ source code files that you can open for editing.
- Navigating below the *Run Rules* heading leads to a *DFE* run rule and a *Simulation* run rule. Right clicking on either of these brings up a menu to build, run, or set options for the project.

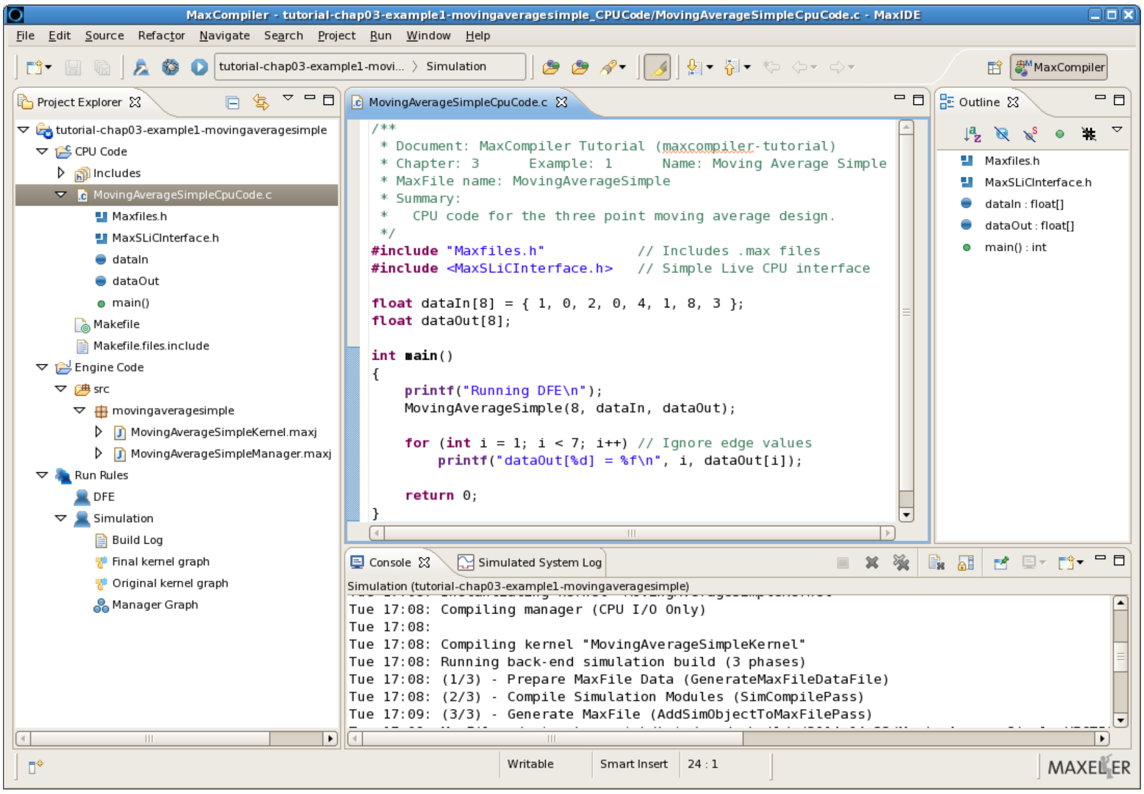


Figure 15: MaxIDE with an imported project

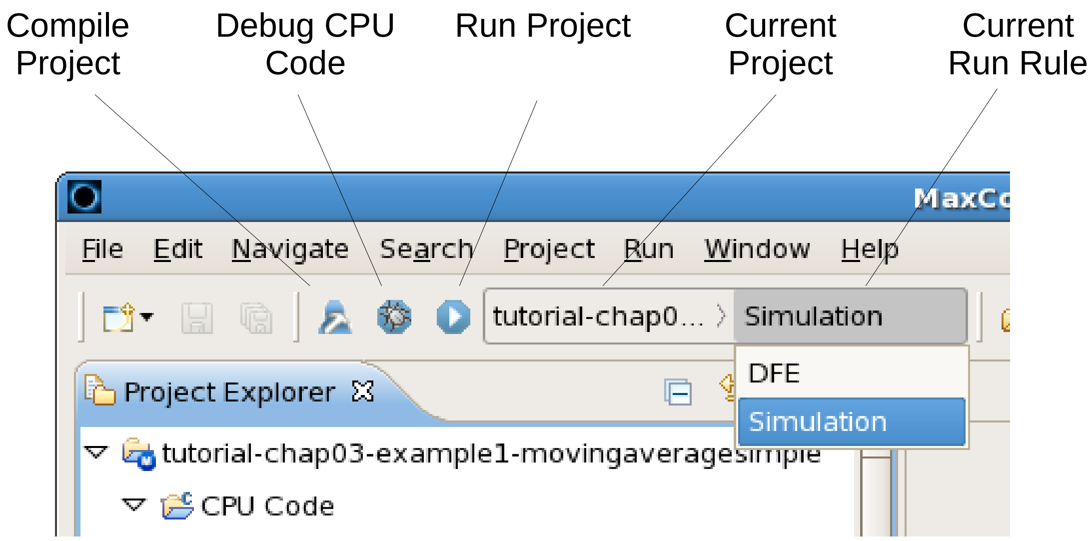


Figure 16: MaxIDE buttons for building and running a project

4.1.3 Building and running designs

You can build and run projects using the buttons in the toolbar at the top of MaxIDE, as shown in *Figure 16*. You can select a project and run rule combination using the drop down boxes, then click one of the buttons to build or run it.

The buttons perform the following actions:

1. Build either a simulation or DFE `.max` file, depending on the selected run rule.
2. This step differs for each of the buttons:
   - *Compile Project* - compiles the CPU source code.
   - *Debug CPU Code* - builds and runs the CPU source code in debug mode, where you can step through the CPU code.
   - *Run Project* - builds and runs the CPU source code in release mode.

Alternatively, a run rule can be built or run by right-clicking on it in the *Project Explorer* and selecting either *Build*, *Debug* or *Run*.

4.1.4 Importing projects

You can import any projects, including the tutorial projects, by:

- Right-clicking in the *Project Explorer* window and select *Import...*.
- Selecting *Import...* from the *File* menu.

Both methods open a dialog where you can select the type of project to import. Select *General→MaxCompiler Projects into Workspace* to import a MaxCompiler project, then in the next screen browse to the directory contain the projects you wish to import. The final screen allows you to select the projects that you want to import.

If you are using a shared install of MaxCompiler, you might consider checking the *Copy projects into workspace* option, otherwise you will be editing the projects in situ. Finally, the *Open code files automatically after import* option will close all windows and show the CPU code, Kernel code and Manager code side by side for the project, which is useful for demonstrating a project.

> Eclipse has extensive documentation and community support at `http://www.eclipse.org/`, which may be a helpful supplement to this tutorial.

4.2 Building the examples and exercises outside of MaxIDE

Although highly recommended, MaxIDE is not required for running the examples or any other imported projects. The source code for any project imported into MaxIDE is accessible in a directory under your designated MaxIDE workspace directory (typically `$HOME/workspace`). Project directory hierarchies are organized and named identically to the hierarchy of headings in the Project Explorer panel (without spaces). Hence, under each project subdirectory, there are sub-directories named `CPUCode`, `EngineCode`, and `RunRules`.

- The `CPUCode` directory contains C or C++ source files and header files for the project.
- The `EngineCode` directory contains a `src` subdirectory and possibly a `bin` subdirectory.
  - The `bin` subdirectory stores compiled MaxJ class files, if any.
  - The `src` subdirectory has exactly one subdirectory named after the project. This subdirectory contains MaxJ source code files.
- The `RunRules` directory contains a subdirectory named `DFE` and possibly a subdirectory named `Simulation`, each containing automatically generated configuration files and Makefiles.

To build a project outside of MaxIDE for a DFE or simulation, navigate to the corresponding `RunRules/DFE` or `RunRules/Simulation` directory of the project hierarchy, and invoke the `make` utility using one of the automatically generated Makefiles with an optional target.

- `make` – with no target builds either a simulation model or a DFE configuration `.max` file for the application without running it.
- `make startsim` – starts a simulator if invoked from the `Simulation` directory and there is no simulator already running, but has no effect if invoked in the `DFE` directory or when a simulator is already running.
- `make run` – builds the application if necessary, and then runs it either in a DFE or in simulation, depending on the directory.
  - For DFE runs, DFE hardware is needed.
  - For simulation runs, an already running simulator started by `make startsim` is needed.
- `make stopsim` – invoked from the `DFE` directory has no effect. From the `Simulation` directory, it either stops a simulator if one is running, or causes an error if not.
- `make runsim` – is equivalent to `make startsim run stopsim`

4.3 A basic kernel

The basic example that we will follow throughout this section is a Kernel that takes a single input stream `x` and applies a simple function:

$$f(x) = x^2 + x$$

The resulting stream is connected directly to the output stream `y`. *Figure 17* shows a graphical representation of this Kernel in the form of its **Kernel graph**. The Kernel graph shows the flow of data from inputs at the top to outputs at the bottom, passing through the nodes that are created by operations we describe within the Kernel.

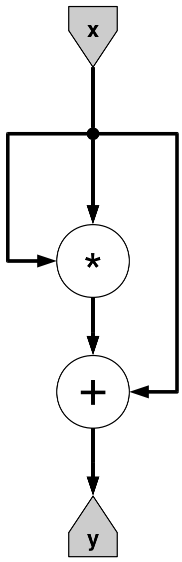


Figure 17: Graph for a simple Kernel

Listing 8 shows the Java code that implements this Kernel. We will go through this code line by line.

The first five lines specify the package for this example and import Java classes: `Kernel`, `KernelParameters` and `DFEVar` from the MaxCompiler Java libraries:

```
package simple;

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

It is common to have many package imports in MaxCompiler-based programs as MaxCompiler is implemented as a software library. MaxIDE automatically generates these imports in most circumstances.

The `Kernel` class provides an entry point into Kernel development. Within the `Kernel` class are directly or indirectly a large number of Java methods for creating Kernel designs.

We can create new Kernels by extending the class `Kernel`:

```
class SimpleKernel extends Kernel {
```

Listing 8: Program for the simple Kernel (SimpleKernel.maxj).

```
/**
 * Document: MaxCompiler tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 4        Example: 2        Name: Simple
 * MaxFile name: Simple
 * Summary:
 *    Takes a stream and for each value x calculates x^2 + x.
 */
package simple;

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

class SimpleKernel extends Kernel {
    SimpleKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar x = io.input("x", dfeFloat(8, 24));

        DFEVar result = x*x + x;

        // Output
        io.output("y", result, dfeFloat(8, 24));
    }
}
```

We now need to define a constructor for our new `SimpleKernel` class:

```
    SimpleKernel(KernelParameters parameters) {
```

In Java, a constructor serves as an initialization function executed when an object of a class is instantiated.

The constructor of a `Kernel` class needs at least one parameter, an object of class `KernelParameters`. In our program, we have chosen the name `parameters` for this object. Although our Kernel does not make much use of the `parameters` object, this object is used internally within the `Kernel` class. For this reason, we need to pass the `parameter` object to the constructor of the `Kernel` object. In Java, we do this using the `super` call:

```
        super(parameters);
```

The code in the body of the program generates the Kernel graph shown in *Figure 17*.

We use the method `io.input` to create a named input stream:

```
        DFEVar x = io.input("x", dfeFloat(8, 24));
```

`io.input` takes two parameters. The first is a string representing the name, in this case `x`, by which the stream can be referred to when configuring a Manager (see *section 13*) and running the Kernel from the C CPU code (see *subsection 4.5*). The second parameter specifies the data type for the stream. In our example we define the data type as an IEEE 754 single-precision floating point number.

The actual function is implemented intuitively using standard operators:

```
        DFEVar result = x*x + x;
```

We then connect the result directly to the output stream using `io.output`, which takes the name of the stream, the internal stream to connect to the output and the type of the stream:

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

Input and output streams are referred to as **external** as they are connected to the rest of the dataflow engine in a Manager. Depending on the Manager used, these external I/Os can be connected to memory, another dataflow engine or the CPU.

4.4 Configuring a Manager

After designing the Kernel, we need to configure a Manager to connect our Kernel to the outside world and build our design for either DFE output or simulation.

Listing 9 presents the Java code for the Manager that builds the DFE for our simple Kernel.

Listing 9: Program for building the simple dataflow example (SimpleManager.maxj).

```
/**
 * Document: MaxCompiler tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 4      Example: 2      Name: Simple
 * MaxFile name: Simple
 * Summary:
 *      Manager for the simple x*x + x kernel design.
 *    All IO is between the CPU and the DFE.
 */
package simple;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;

class SimpleManager {
    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        Manager manager = new Manager(params);
        Kernel kernel = new SimpleKernel(manager.makeKernelParameters());
        manager.setKernel(kernel);
        manager.setIO(IOType.ALL_CPU);
        manager.createSLiCinterface();
        manager.build();
    }
}
```

We first specify the package and import the class `Manager` from the Maxeler Standard Managers library:

```
package simple;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;
```

We then declare a new class `SimpleManager` that serves as a container for our DFE build program and contains a `main` method to run the build process:

```
class SimpleManager {
    public static void main(String[] args) {
```

We now create an object named `manager` of class `Manager` which is a Manager that can be used to connect Kernel streams to communicate directly with the CPU, to an inter-dataflow-engine MaxRing link or to LMEM directly attached to the dataflow engine:

```
        EngineParameters params = new EngineParameters(args);
        Manager manager = new Manager(params);
```

Whether the Manager is built for DFE configurations or simulation is determined by the `EngineParameters` object passed to the constructor.

MaxIDE passes whether or not the `.max` file is to be built for simulation or DFEs, as well as other information, from the run rule via an environment variable that is parsed by MaxCompiler. The Managers that we use for DFEs and simulation are identical in all other respects for this design.

We create an instance of our Kernel and pass it to the Manager:

```
        Kernel kernel = new SimpleKernel(manager.makeKernelParameters());
        manager.setKernel(kernel);
```

For our simple example, we want all the inputs and outputs to be connected to the CPU application. We do this using the `setIO` method:

```
        manager.setIO(IOType.ALL_CPU);
```

A single function call builds the default SLiC interface for the CPU code:

```
        manager.createSLiCinterface();
```

Finally, we call the `build()` method of the standard Manager class, which runs all the steps required for building the dataflow engine, such as calling various back-end tools:

```
        manager.build();
```

Listing 10 and *Listing 11* show example console output from the build process.

4.4.1 Building the `.max` file

The results of the build process are the files `Simple.max` and `Simple.h`, which are copied to the run rule directory of the project by MaxIDE.

> You can view the build log for a run rule by right-clicking on the run rule in the project explorer window and selecting *Show Build Log*.

Listing 10: DFE Build Output (part 1)

```
Tue 12:28: ##########################################
Tue 12:28: Compiling.
Tue 12:28: ##########################################

Tue 12:28: MaxCompiler version: 2013.3.
Tue 12:28: Build DFE Run Rule for tutorial-chap04-example2-simplekernel start time: Tue Feb 25 12:28:36 GMT 2014.
Tue 12:28: Project location: /home/user/tutorial-chap04-example2-simplekernel.
Tue 12:28: Detailed build log: /home/user/workspace/tutorial-chap04-example2-simplekernel/RunRules/DFE/build.log.

Tue 12:28: Compiling Engine Code.
Tue 12:28: MaxCompiler version: 2013.3
Tue 12:28: Build "Simple" start time: Tue 25 12:28:41 GMT 2014
Tue 12:28: Main build process running as user user on host host.maxeler.com
Tue 12:28: Build location: /home/user/builds/Simple_VECTIS_DFE
Tue 12:28: Detailed build log available in "_build.log"
Tue 12:28: Instantiating manager
Tue 12:28: Instantiating kernel "SimpleKernel"
Tue 12:28: Compiling manager (CPU I/O Only)
Tue 12:28:
Tue 12:28: Compiling kernel "SimpleKernel"
Tue 12:29: Generating input files (VHDL, netlists, CoreGen)
Tue 12:31: Running back-end build (12 phases)
Tue 12:31: (1/12) - Prepare MaxFile Data (GenerateMaxFileDataFile)
Tue 12:31: (2/12) - Synthesize DFE Modules (XST)
Tue 12:32: (3/12) - Link DFE Modules (NGCBuild)
Tue 12:32: (4/12) - Prepare for Resource Analysis (EDIF2MxruBuildPass)
Tue 12:33: (5/12) - Generate Preliminary Annotated Source Code (PreliminaryResourceAnnotationBuildPass)
Tue 12:33: (6/12) - Report Resource Usage (XilinxPreliminaryResourceSummary)
```

Listing 11: DFE Build Output (part 2)

```
Tue 12:33:
Tue 12:33: PRELIMINARY RESOURCE USAGE
Tue 12:33: Logic utilization:   9809 / 297600 (3.30%)
Tue 12:33:  LUTs:               6787 / 297600 (2.28%)
Tue 12:33:  Primary FFs:        7539 / 297600 (2.53%)
Tue 12:33: Multipliers (25x18):    2 / 2016  (0.10%)
Tue 12:33:  DSP blocks:            2 / 2016  (0.10%)
Tue 12:33: Block memory (BRAM18): 21 / 2128  (0.99%)
Tue 12:33:
Tue 12:33: About to start chip vendor Map/Place/Route toolflow. This will
    take some time.
Tue 12:33: For this compile, we estimate this process may take up to 30
    minutes.
Tue 12:33: We recommend running in simulation to verify correctness before
    building a DFE configuration.
Tue 12:33:
Tue 12:33: (7/12) - Prepare for Placement (NGDBuild)
Tue 12:33: (8/12) - Place and Route DFE (XilinxMPPR)
Tue 12:33: Executing MPPR with 1 cost tables and 1 threads.
Tue 12:33: MPPR: Starting 1 cost table
Tue 12:40: MPPR: Cost table 1 met timing with score 0 (best score 0)
Tue 12:40: (9/12) - Prepare for Resource Analysis (XDLBuild)
Tue 12:41: (10/12) - Generate Resource Report (XilinxResourceUsageBuildPass)
Tue 12:41: (11/12) - Generate Annotated Source Code (
    XilinxResourceAnnotationBuildPass)
Tue 12:41: (12/12) - Generate MaxFile (GenerateMaxFileXilinx)
Tue 12:43:
Tue 12:43: FINAL RESOURCE USAGE
Tue 12:43: Logic utilization:   7563 / 297600 (2.54%)
Tue 12:43:  LUTs:               6332 / 297600 (2.13%)
Tue 12:43:  Primary FFs:        5845 / 297600 (1.96%)
Tue 12:43:  Secondary FFs:      1360 / 297600 (0.46%)
Tue 12:43: Multipliers (25x18):    2 / 2016  (0.10%)
Tue 12:43:  DSP blocks:            2 / 2016  (0.10%)
Tue 12:43: Block memory (BRAM18): 23 / 2128  (1.08%)
Tue 12:43:
Tue 12:43: MaxFile: /home/user/builds/Simple_VECTIS_DFE/results/Simple.max (
    MD5Sum: 47ecae3e026eaceb661784b29e19c784)
Tue 12:43: Build completed: Tue Feb 25 12:43:32 GMT 2014 (took 14 mins, 51
    secs)
```

4.5 Integrating with the CPU application

The final step in the development of a dataflow implementation is integrating the CPU application with the **SLiC Interface**, the C API for communicating with the dataflow engine. *Listing 12* show the CPU application for our simple example.

The header files for SLiC and the `.max` file created by the build process need to be included in the source:

```
#include <MaxSLiCInterface.h>
#include "Maxfiles.h"
```

MaxIDE sets up an auto-generated `Maxfiles.h` file to include the header files for all the `.max` files in your program. If you prefer you can include the header file for each one of your `.max` files files manually.

The first part of the main program code is to allocate memory for two arrays of length `size`, one for input to the Kernel and one for output. The input data array is set to the values `1-size` and the output data array is initialized to zero:

```
    for(int i = 0; i < size; i++) {
        dataIn[i] = i + 1;
        dataOut[i] = 0;
    }
```

This example uses the simplest form of the SLiC Interface to the dataflow engine, which consists of a single function. The function prototype is automatically generated by MaxCompiler from the Java source and can be found in the header file `simple.h` after the `.max` file has been built:

```
void Simple(
    int32_t param_size,          // Number of items to process
    const float *instream_x,     // Input stream pointer
    float *outstream_y);         // Output stream pointer
```

`param_size` is the size of the input and output stream in items, where each item is a 32-bit float. `instream_x` is a pointer to the array containing the data to stream into the DFE and `outstream_y` is a pointer to the array for the data we wish to stream back from the DFE.

The size of the array to stream to/from the DFE must be a multiple of 16 bytes, otherwise SLiC produces an error message.

When called, the function streams both the input data array from the CPU memory to the dataflow engine and the dataflow engine's output back to the output data array in CPU memory:

```
    Simple(size, dataIn, dataOut);
```

Once the Kernel has been run we can verify the correctness of the result:

```
for(int i=0; i < size; i++)
{
    if (dataOut[i] != expected[i]) {
        fprintf (stderr, "Output data @ %d = %1.8g (expected %1.8g)\n",
            i, dataOut[i], expected[i]);
        status = 1;
    }
}
```

4.6 Kernel graph outputs

The Kernel Compiler can generate a number of graph files representing the Kernel at various stages during the compilation process. You can inspect these graphs to analyze the output derived from the input Java and debug the design.

The graphs are viewed from MaxIDE. After building the kernel you will find the graphs appear under the RunRule that was built. Typically there are two graphs:

- `Original Kernel Graph`
  This graph, shown in *Figure 18*, presents the Kernel graph before any optimization or scheduling has taken place. This graph is the same as the one drawn in *Figure 17*.
- `Final Kernel Graph`
  This graph, shown in *Figure 19*, displays the final Kernel after scheduling and optimisation has taken place and inputs are aligned by buffering.

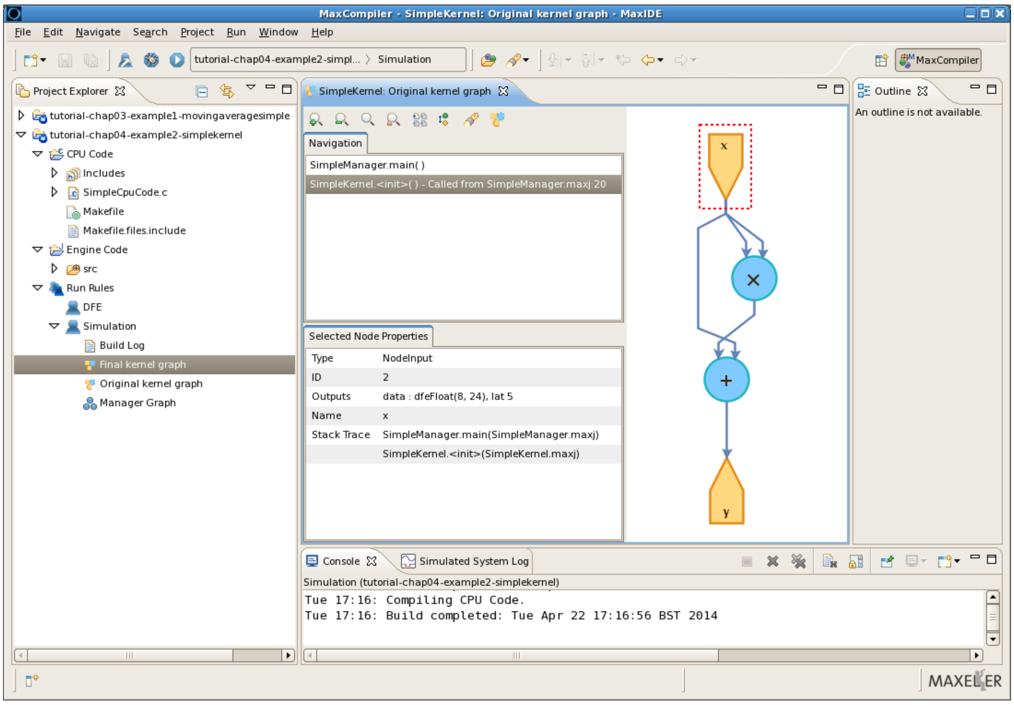


Figure 18: Kernel Graph Viewer showing the Original graph for the simple Kernel in MaxIDE

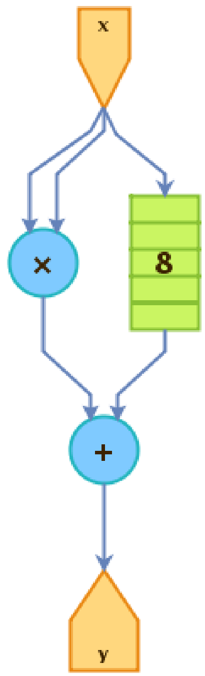


Figure 19: Final graph for the simple Kernel

> To make graphs more readable, nodes created inside functions are grouped together into a single node. Double click on the node to view the nodes inside.

4.7 Analyzing resource usage

MaxCompiler annotates your Java code with the number of DFE resources used for each line. These annotated files are output into a folder called `src_annotated` in the build directory of the project, along with a summary report, during the build process. Another folder called `src_annotated_preliminary` is created, which contains the initial resource estimates before any optimization.

> The build directory is determined by your `MAXCOMPILER_BUILD_DIR` environment variable, and is unrelated to your workspace directory. Check the `Build Location:` message in the console output at compile-time for its location.

> Resource annotation is only available for DFE configuration builds.

The resource annotation folders are copied from your build directory into your RunRule folder, where they can be viewed within MaxIDE.

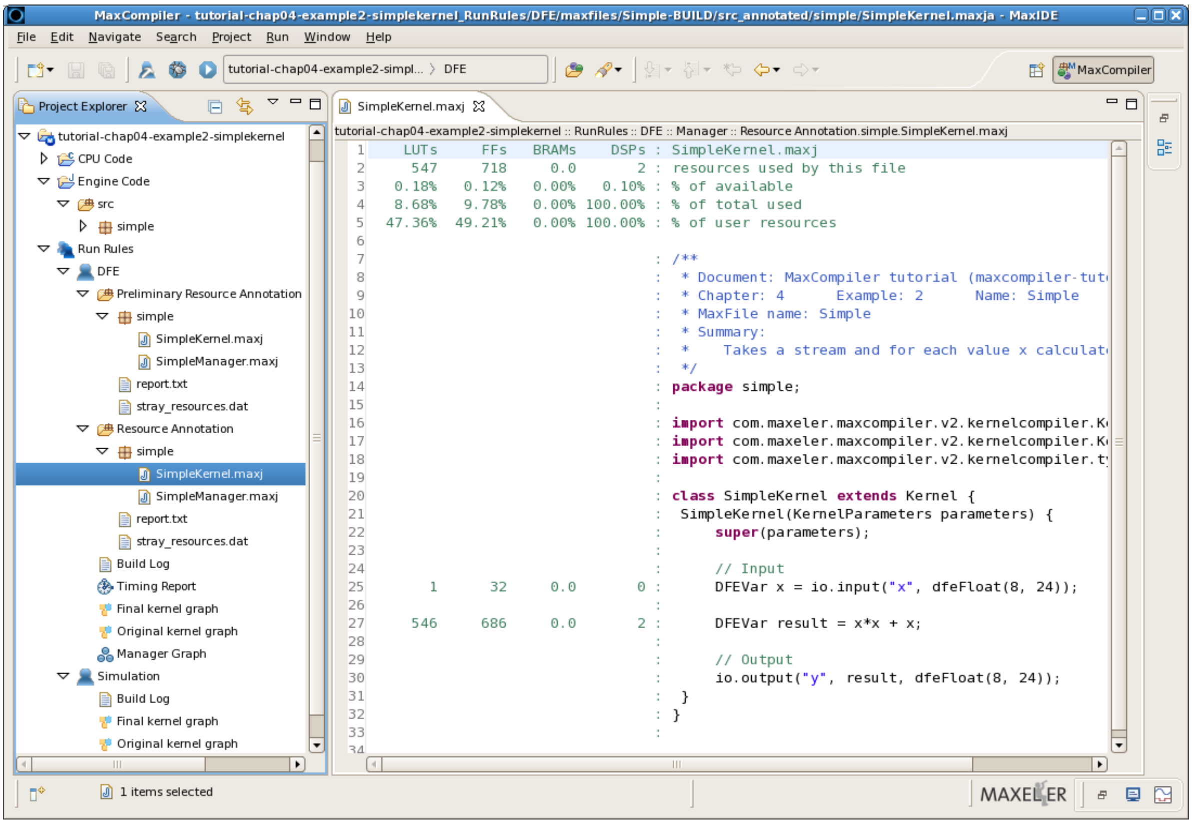


Figure 20: Annotated-Source Viewer showing the resource statistics for the simple Kernel in MaxIDE

Figure 20 presents the resource statistics for our simple Kernel in MaxIDE. The right-hand side of the listing displays the Kernel program, and the left-hand side annotates the program with the used resources line by line. The resources include look-up tables (`LUTs`), flip-flops (`FFs`), block RAMs (`BRAMs`) and DSP blocks (`DSPs`).

In general, LUTs and FFs are used for kernel operations such as arithmetic, DSPs are specifically used for multiplication operations and BRAMs are used for FMem memory. Some LUTs, FFs and BRAMs are also used by MaxelerOS in the Manager.

Line 2 lists the total number of resources required by our Kernel design and shows that our simple Kernel uses 2 DSPs but no block RAMs. Line 3 shows that the design uses only 0.18% of the device's look-up tables, 0.12% of its flip-flops and 0.10% of its DSPs. This leaves ample room for increasing the performance by exploiting more parallelism (for example, by simply replicating the Kernel graph).

By analyzing the resources required for each code line, you can optimize the design for minimal resource requirements which increases the dataflow configuration's performance. Furthermore, you can gain insight into the resource trade-offs of different styles of Kernel programming.

We can see from line 27 that nearly all of the resources used by the simple Kernel are due to the arithmetic operations. The input statement takes 32 flip-flops, and the output statement takes none (lines 25 and 30).

4.7.1 Enabling resource annotation

When using MaxIDE, or any Ant scripts generated by MaxIDE, resource annotation is set up and run automatically.

If you are using an alternative method to build your code, the MaxCompiler infrastructure must know where your source code files are located on the file system. MaxCompiler uses the `MAXSOURCEDIRS` environment variable to specify the directories containing source code used by your project. These files are then copied to the build directory during the build process and annotated with resource usage information.

The `MAXSOURCEDIRS` environment variable specifies one or more source code directories separated by colons (`:`). For example: `/home/user/src/dir1:/home/user/src/dir2`.

> Each directory in `MAXSOURCEDIRS` must be the parent of a package directory named in the `package` declaration of a source file.

Given a source file `SimpleKernel.maxj` located in the directory

`/home/user/workspace/tutorial_chap02_example2_simple/EngineCode/src/simple`

containing the declaration `package simple`, this example would require a value of

`/home/user/workspace/tutorial_chap02_example2_simple/EngineCode/src`

in `MAXSOURCEDIRS`.

> If you change `MAXSOURCEDIRS` using MaxIDE's *Run→Run Configurations...* dialog, your setting is displayed in the dialog window and used. Unsetting it reverts it to the default and stops it from being displayed.

Exercises

Exercise 1: M-Fold simple Kernel

Modify the simple example Kernel (a copy of which is provided in the `exercises` folder for you to change) that we have worked through in this section to work with M parallel streams. The Kernel should serve M independent streams as sketched in *Figure 21*. Replicating a single stream computation (also called a *pipe*) within a Kernel is a common design pattern for creating dataflow engine implementations. For applications that show sufficient parallelism, mapping multiple pipes into a Kernel design greatly increases the dataflow engine performance.

Name the stream inputs and outputs `x1`, `x2`, ..., `xM` and `y1`, `y2`, ..., `yM`, respectively. The number of streams M should be passed to the Kernel program as a constructor argument.

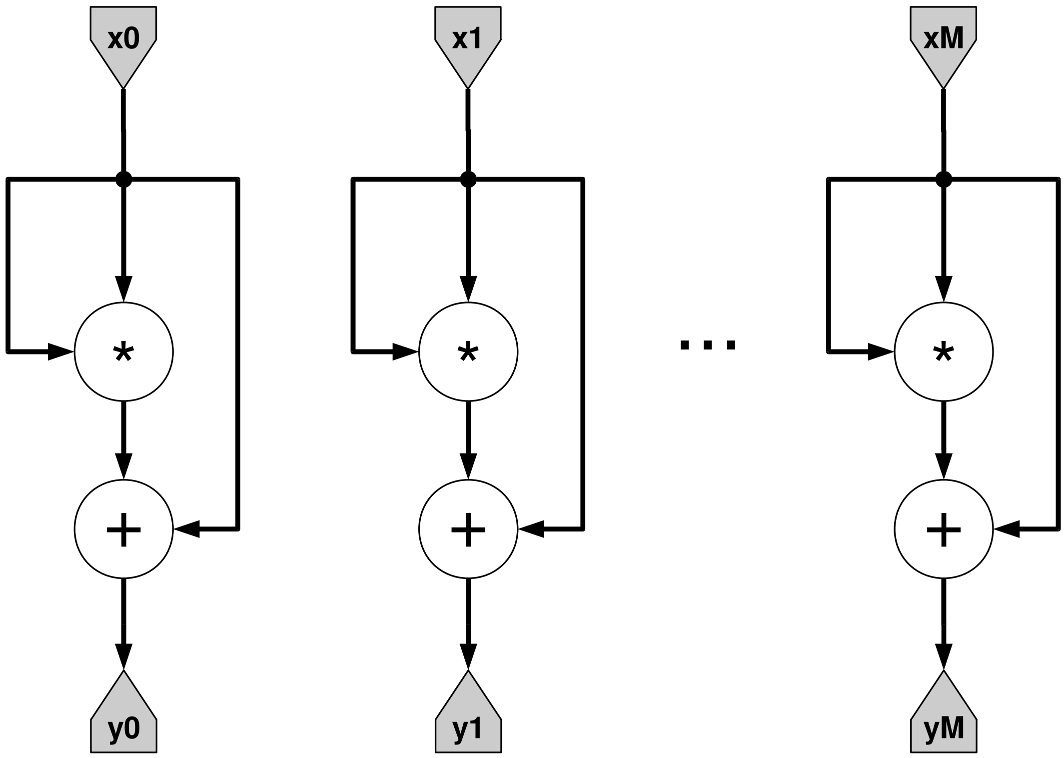


Figure 21: Sketch of the M-fold simple Kernel

Listing 12: Simple CPU application (SimpleCpuCode.c).

```
/**
 * Document: MaxCompiler tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 4      Example: 2      Name: Simple
 * MaxFile name: Simple
 * Summary:
 *    Takes a stream and for each value x calculates x^2 + x.
 */
#include <stdint.h>
#include <MaxSLiCInterface.h>
#include "Maxfiles.h"

int check(float *dataOut, float *expected, int size)
{
    int status = 0;
    for(int i=0; i < size; i++)
    {
        if (dataOut[i] != expected[i]) {
            fprintf (stderr, "Output data @ %d = %1.8g (expected %1.8g)\n",
                i, dataOut[i], expected[i]);
            status = 1;
        }
    }
    return status;
}

void SimpleCPU(int size, float *dataIn, float *dataOut)
{
    for (int i=0 ; i<size ; i++) {
        dataOut[i] = dataIn[i]*dataIn[i] + dataIn[i];
    }
}

float dataIn[1024];
float dataOut[1024];
float expected[1024];
const int size = 1024;

int main()
{
    for(int i = 0; i < size; i++) {
        dataIn[i] = i + 1;
        dataOut[i] = 0;
    }

    SimpleCPU(size, dataIn, expected);
    Simple(size, dataIn, dataOut);

    printf ("Running DFE.\n");
    int status = check(dataOut, expected, size);
    if (status)
        printf ("Test failed.\n");
    else
        printf ("Test passed OK!\n");
    return status;
}
```

5

Debugging

[...] the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

– Maurice Wilkes

MaxCompiler offers a number of methods for debugging a design. We can debug Kernels using **watches**, **simulation** `printf` and **DFE** `printf`. Watches tell us the value of any specified `DFEVar` for every tick that a Kernel is running. DFE and simulation `printf` allow us to print and format the print values explicitly from streams within the Kernel on every tick, optionally enabled via a Boolean stream. Watches and `simPrintf` are available in simulation only and ignored for DFE hardware runs, whereas `dfePrintf` are available for both simulation and DFE hardware runs.

Two more advanced methods for debugging a DFE as it is running, or after it has run, are covered in *subsection 5.3*.

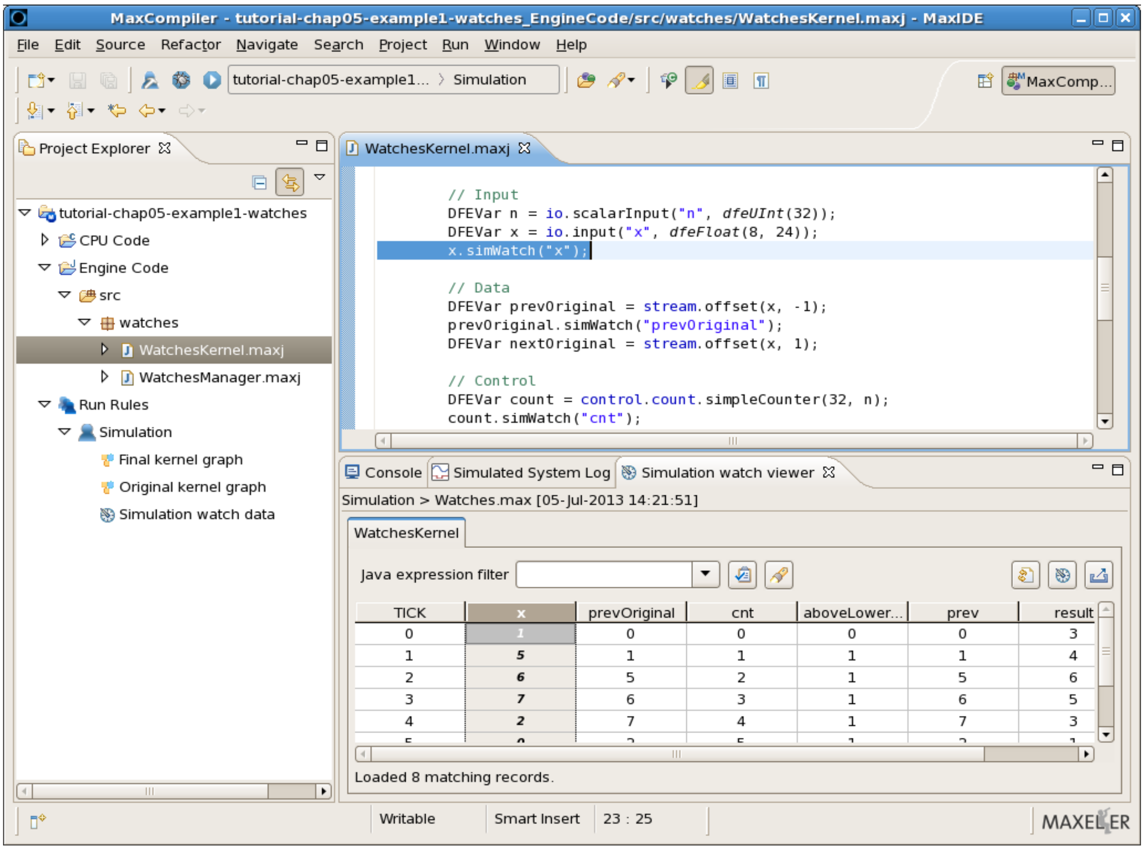


Figure 22: MaxIDE displaying simulation watch data

5.1 Simulation watches

A watch can be added to a stream in simulation to record the value of the stream on every Kernel tick.

To add a watch, we use a method of the class `DFEVar` called `simWatch` which takes a string argument that is used to label the watch. When the Kernel is run in simulation it generates a report in CSV (comma-separated values) format, containing data for variables which have been watched.

Listing 13 shows our moving average example with watches added to several of the streams. For example, we can watch the input to the Kernel:

```
DFEVar x = io.input("x", dfeFloat(8, 24));
x.simWatch("x");
```

We can also watch intermediate values in the middle of the computation:

```
DFEVar prev = aboveLowerBound ? prevOriginal : 0;
prev.simWatch("prev");
```

Upon completion of the run, the CSV report can be accessed from within MaxIDE by way of the "Simulation watch data" item that appears under the associated RunRule in the Project Explorer, as seen in *Figure 22*.

The Simulation watch viewer can display large datasets and makes it possible to filter, search through and export the data presented:

- the data may be filtered by way of a "Java expression filter" text field: boolean expressions written

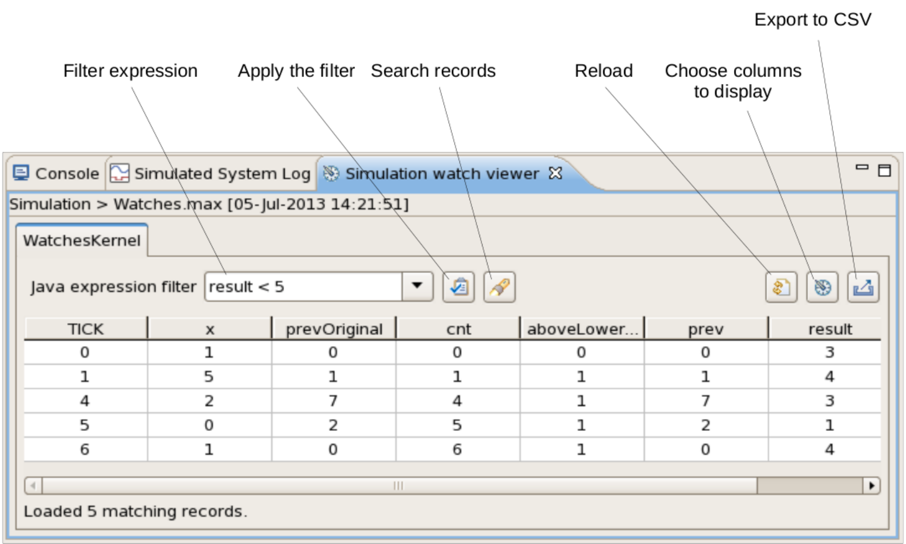


Figure 23: MaxIDE simulation watch viewer utilities

in the Java language may be used to display only these records for which the expression is true. The string names used in the `simWatch` call may be used as variables in these expressions;

- the records displayed may be searched by way of a Java expression of the same nature as with the Java expression filter;
- a column selection tool may be used to display only selected columns;
- finally, the filtered data displayed may be exported in CSV format for use with external applications.

The tools and widgets used in the above operations are indicated in *Figure 23*.

When an application is run from within MaxIDE, the file containing the original data presented in the Simulation watch viewer appears under the `debug` directory of the RunRule folder of the project. When the application is run from the command line, this file appears under a debug directory in the working directory. This directory may be named `debug` or have a name of the form `debug_N`, where `N` is an index used to make this directory name unique. The naming of this debug directory is covered further in *subsection 10.13*.

The output file name is formed by concatenating `watch_` with the name of Maxfile and the name of your Kernel, e.g. `watch_Watches_WatchesKernel.csv`.

By default, debug output is produced for all ticks while the kernel is running. The function `max_watch_range` is used to limit debug output for ticks to a given range:

```
void max_watch_range(
        max_actions_t    *actions,
        const char       *kernel_name,
        int               start_tick ,
        int              num_ticks);
```

Listing 13: Program for the moving average Kernel with watches (WatchesKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 5      Example: 1      Name: Watches
 * MaxFile name: Watches
 * Summary:
 *        Kernel that computes a three point moving average with boundaries,
 *   while printing watch information.
 */
package watches;

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

class WatchesKernel extends Kernel {

    WatchesKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar n = io.scalarInput("n", dfeUInt(32));
        DFEVar x = io.input("x", dfeFloat(8, 24));
        x.simWatch("x");

        // Data
        DFEVar prevOriginal = stream.offset(x, -1);
        prevOriginal.simWatch("prevOriginal");
        DFEVar nextOriginal = stream.offset(x, 1);

        // Control
        DFEVar count = control.count.simpleCounter(32, n);
        count.simWatch("cnt");
        DFEVar aboveLowerBound = count > 0;
        DFEVar belowUpperBound = count < n - 1;
        DFEVar withinBounds = aboveLowerBound & belowUpperBound;
        aboveLowerBound.simWatch("aboveLowerBound");

        DFEVar prev = aboveLowerBound ? prevOriginal : 0;
        prev.simWatch("prev");
        DFEVar next = belowUpperBound ? nextOriginal : 0;

        DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;

        DFEVar result = (prev + x + next) / divisor;
        result.simWatch("result");

        // Output
        io.output("y", result, dfeFloat(8, 24));
    }

}
```

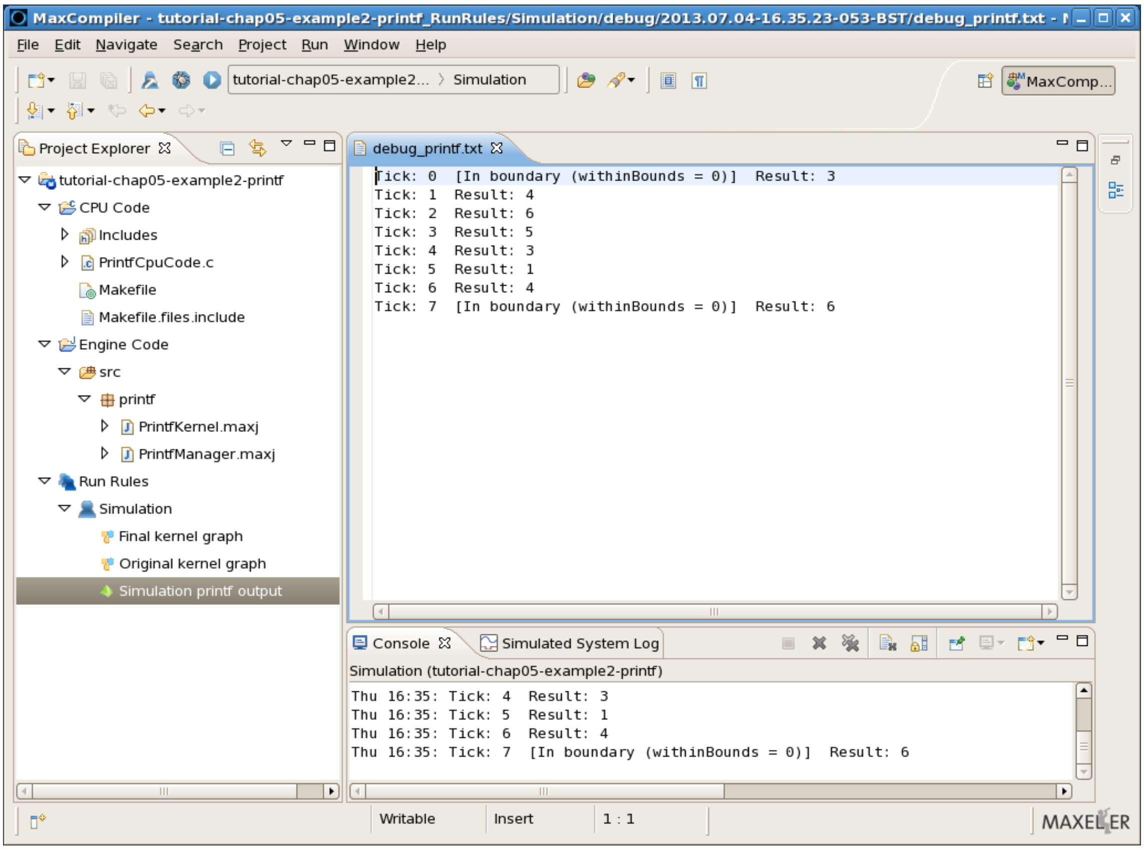


Figure 24: Simulation printf output in MaxIDE

5.2 Simulation and DFE `printf`

`simPrintf` and `dfePrintf` behave similarly to `printf` in C.

The basic form of a `simPrintf` and `dfePrintf` prints the formatted message to the standard output of the application with the current value of the supplied list of objects for every tick:

__void__ debug.simPrintf(__String__ message, Object... args)
__void__ debug.dfePrintf(__String__ message, Object... args)

To print the message only when a certain condition is met, a Boolean `DFEVar` stream can be supplied as a condition:

__void__ debug.simPrintf(DFEVar condition, __String__ message, Object... args)
__void__ debug.dfePrintf(DFEVar condition, __String__ message, Object... args)

The message is only printed when `condition` evaluates to `1`.

The arguments `args` can contain Java variables (`Byte`, `Short`, `Integer`, `Long`, `Float` or `Double`) or `DFEVar` streams. The argument `message` should contain corresponding format specifiers. All familiar format specifiers from C `printf` are provided, with the variation that `%o` prints in *binary*, rather than octal.

The outputs of `dfePrintf` and `simPrintf` appear on the standard output of the application and are also saved to files upon completion of a run:

- `dfePrintf` produces an output file per allocated engine. These files, available in the debug directory, are named using the scheme `debug_dfeprintf.TAG.txt`, where `TAG` is chosen to ensure that file names are unique. The outputs of dfePrintf are flushed upon deallocation of the engine used or, if the basic static interface is used, upon deallocation of the maxfile using the MAXFILE_free() function, where MAXFILE is the name of the maxfile;

- the outputs of simPrintf are collated into a single debug_printf.txt file in the debug directory of the RunRule;
- simPrintf accepts an extra string argument NAME for outputs that should be written to a separate file:

  ```
  void debug.simPrintf(String NAME, String message, Object... args)
  void debug.simPrintf(String NAME, DFEVar condition, String message, Object... args)
  ```

  The output will then appear in a file named debug_printf_NAME.txt in the debug directory.

Listing 14 shows our familiar moving average example with several simPrintfs.

The first simPrintf shows the current value output by the counter, which gives us the current tick count on every Kernel tick:

```
		debug.simPrintf("Tick: %d ", count);
```

The second simPrintf conditionally prints a message when the current position in the stream is in a boundary:

```
		debug.simPrintf(~withinBounds, "[In boundary (withinBounds = %d)] ", withinBounds);
```

The final simPrintf prints the result produced in the current Kernel tick:

```
		debug.simPrintf("Result: %.3g\n", result);
```

The output from the application is shown in *Listing 15*.

When the application is run from within MaxIDE, this output is also visible in files that appear under the associated RunRule in the Project Explorer, upon completion of the run, as seen in figure *Figure 24*. The on-disk file containing the simPrintf and dfePrintf outputs, similar to watch data, is available in a debug directory in the associated RunRule folder.

When the application is run from the command line, the output files are available in the debug subdirectory of the current directory, or in a directory of the form debug_N, where N is chosen to ensure that this directory name is unique.

The naming of the debug directories and the configuration variables that affect it are described in *subsection 10.13*.

Listing 14: Program for the moving average Kernel with printfs (PrintfKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 5      Example: 2      Name: Printf
 * MaxFile name: Printf
 * Summary:
 *        Kernel that computes a three point moving average with boundaries
 *    printing debug information.
 */
package printf;

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

class PrintfKernel extends Kernel {

    PrintfKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar n = io.scalarInput("n", dfeUInt(32));
        DFEVar x = io.input("x", dfeFloat(8, 24));

        // Data
        DFEVar prevOriginal = stream.offset(x, -1);
        DFEVar nextOriginal = stream.offset(x, 1);

        // Control
        DFEVar count = control.count.simpleCounter(32, n);
        debug.simPrintf("Tick: %d ", count);
        DFEVar aboveLowerBound = count > 0;
        DFEVar belowUpperBound = count < n-1;
        DFEVar withinBounds = aboveLowerBound & belowUpperBound;

        DFEVar prev = aboveLowerBound ? prevOriginal : 0;
        DFEVar next = belowUpperBound ? nextOriginal : 0;

        debug.simPrintf(~withinBounds, "[In boundary (withinBounds = %d)] ", withinBounds);
        DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;

        DFEVar result = (prev+x+next)/divisor;

        debug.simPrintf("Result: %.3g\n", result);
        // Output
        io.output("y", result, dfeFloat(8, 24));
    }
}
```

Listing 15: Example printf output

```
Tick: 0 [In boundary (withinBounds = 0)] Result: 3
Tick: 1 Result: 4
Tick: 2 Result: 6
Tick: 3 Result: 5
Tick: 4 Result: 3
Tick: 5 Result: 1
Tick: 6 Result: 4
Tick: 7 [In boundary (withinBounds = 0)] Result: 6
```

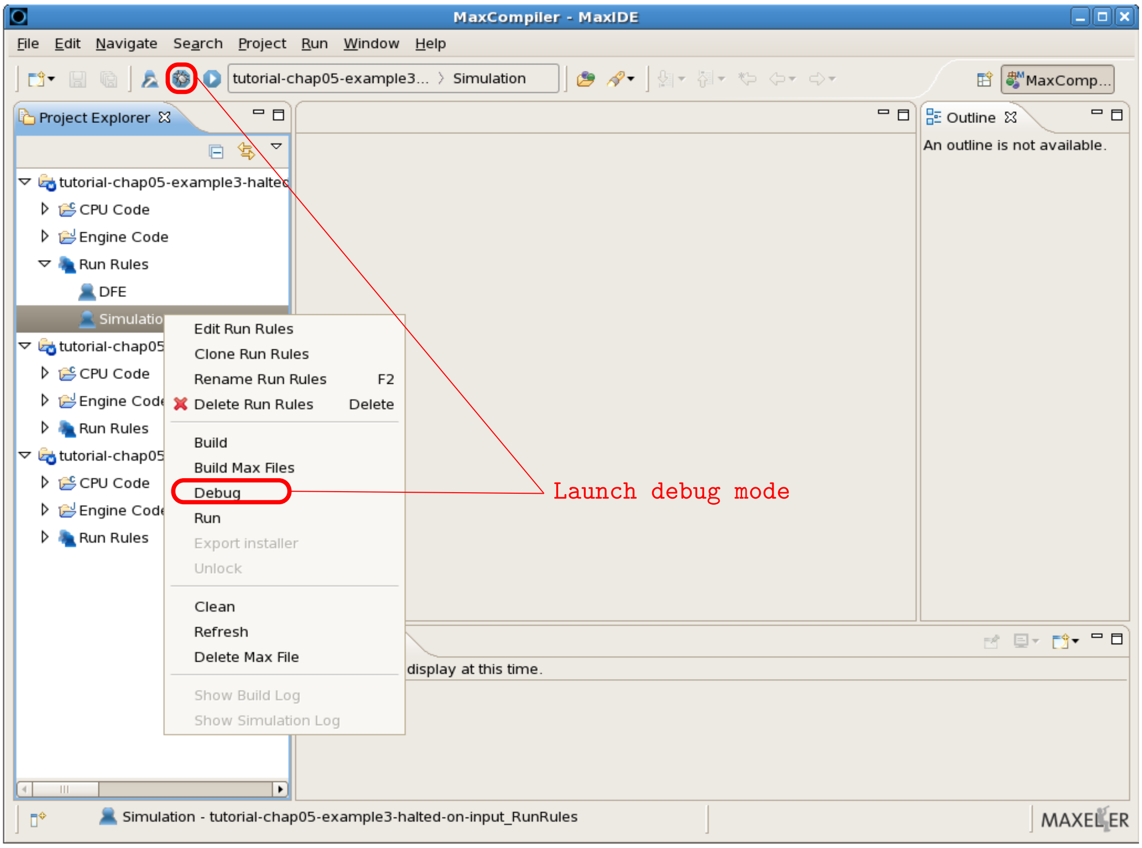


Figure 25: Launching a MaxCompiler project in debug mode.

5.3 Advanced debugging

Debugging with **watches**, **simulation** `printf` and **DFE** `printf` makes it possible to solve a range of issues that may appear in a design; such methods are primarily useful for inspecting the state of kernels.

Since the Manager is a parallel and asynchronous system with many processing units, it is possible to write control code for the CPU that leads to deadlocks such as insufficient data being either produced or consumed by the Kernels. This section presents two further tools that help to identify such issues with the Manager and with the control software on the CPU:

- the graphical debugger in MaxIDE makes it possible to checkpoint the CPU Code of the application and inspect the state of the DFE interactively;
- the MaxDebug tool is a command line utility that obtains similar information on systems where MaxIDE may not be present, and allows inspection of debug snapshots that are optionally produced by SLiC applications as their actions are completed.

The examples in this section are based on a bitstream that performs a computation of the type $s = x + y$, where `x` and `y` are input streams and `s` is an output stream, as seen in the following code

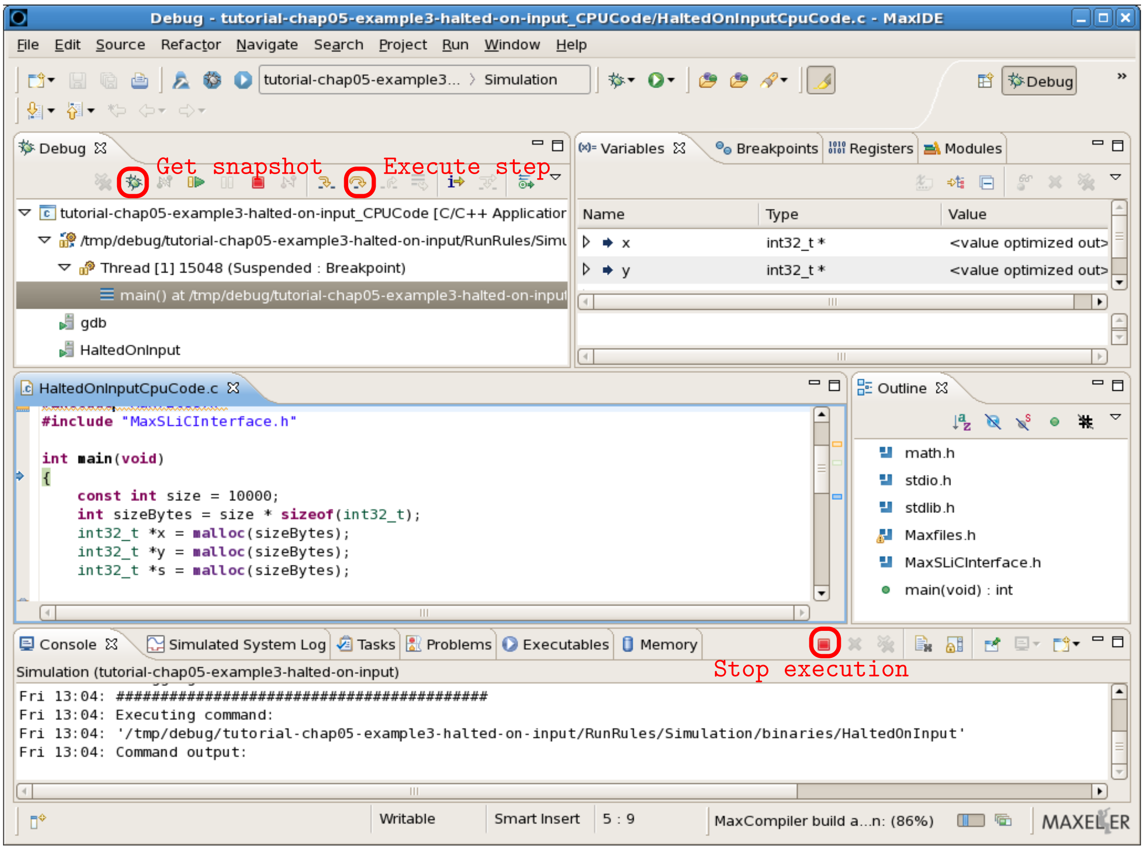


Figure 26: MaxIDE entering the debug mode.

snippet:

```
DFEVar x = io.input("x", type);
DFEVar y = io.input("y", type);
DFEVar sum = x + y;
io.output("s", sum, type);
```

This Kernel expects to read one word of data from each input stream per tick, and to write one word of data to the output stream per tick. The design malfunctions if the amount of data supplied or consumed by the CPU does not match the number of ticks of the Kernel. The examples below demonstrate how to diagnose such issues in live DFEs.

5.3.1 Launching MaxIDE's debugger

A MaxCompiler project can be launched in debug mode from within MaxIDE by pushing the debug button in the toolbar, or by selecting the "Debug" item in the contextual menu of a Run Rule. This is illustrated in *Figure 25*.

After the project has been built, MaxIDE switches to a debugging perspective and presents various views of the program being run, as displayed in *Figure 26*. This mode makes it possible to execute the program one instruction at a time, enabling us, in particular, to find locations where the program might stall. At any time after a run has been launched, it is possible to obtain a snapshot of the state of the DFE by pushing the "Get snapshot" button in the debugging toolbar; this is the button with the bug icon towards the left-hand of the toolbar. MaxIDE then shows a graphical view of the Manager graph.

5.3.2 Kernel halted on input

Running the DFE or Simulation Run Rules for the example project `tutorial-chap5-example3` as distributed, results in applications that hang: the applications either fail with a timeout or need to be terminated manually by way of the red *Terminate* button in the console window.

> When running on a DFE, SLiC functions return an error when streaming operations last more than a prescribed time; simulation does not support such time-outs, and simulated applications with stalling designs should be terminated manually.

Launching the application in debug mode and tracing its execution step by step allows us to see that the function `HaltedOnInput()` in the CPU Code is called but does not return.

At this stage, the state of the Manager can be retrieved by pushing the snapshot button, producing the view in *Figure 27*. The Kernel is represented by the yellow rectangle in the middle of the graph, surrounded by nodes that connect its inputs and outputs to the CPU.

Selecting the Kernel, by clicking with the left mouse button, indicates that its status is "`Halted on input`". In general, the yellow coloring of this state helps to locate problems in the data flow. The Kernel would be colored green if it had run for the required number of ticks.

Examining the CPU code, we see that the number of ticks to run the Kernel has a value that is too large by 5000: the Kernel was running for too long and its input could not read data after `size` ticks.

```
HaltedOnInput(size + 5000, x, sizeBytes, y, sizeBytes, s, sizeBytes);
```

> Note that the current tick count in the MaxDebug output for a Kernel may not be exactly the number expected, given the number of inputs consumed or the number of ticks that it is set to run for, as MaxCompiler may schedule extra ticks into the Kernel.

5.3.3 Kernel halted on output

The project `tutorial-chap5-example4` displays a different problem from the previous one: in the code listing below, the size of the output stream `s` is now too small to accommodate the data produced by the Kernel, leading to a stall.

```
HaltedOnOutput(size, x, sizeBytes, y, sizeBytes, s, sizeBytes -5000*sizeof(int32_t));
```

The result of this is shown graphically in *Figure 28*: the Kernel is marked as "`Halted on output`" and the output stream `s` is marked as "`stalled`".

When the extra subtraction is removed, the application functions correctly; if run in the debugger with a breakpoint added after the call to `HaltedOnOutput()`, then the ensuing snapshot of the DFT shows a graph wherein each status is colored green.

5.3.4 Stream status blocks

Managers have an option to globally enable **stream status blocks**, which are additional blocks that collect information on the streams in and out of a Kernel. The status of these blocks can be read by

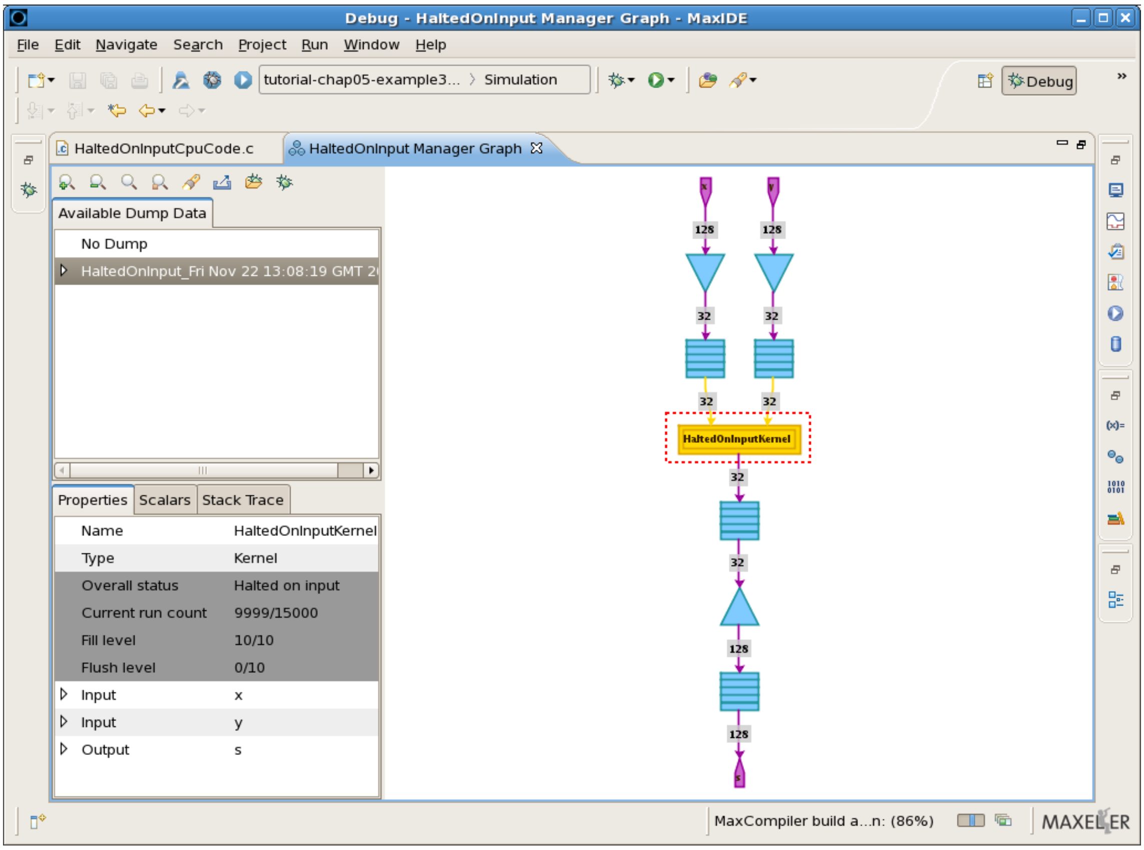


Figure 27: State of the Manager while the function `HaltedOnInput()` is hanging: the Kernel requires more data than is available.

the debugging tools to get additional information from the DFE at run-time. See *section 13* for more information on Managers.

> The Manager requires rebuilding when stream status blocks are enabled, as they must be built into the `.max` file.

We can take the previous example and enable the stream status blocks using the `setEnableStreamStatusBlocks()` method on the Standard Manager:

```
manager.setEnableStreamStatusBlocks(true);
```

For a Custom Manager, stream status blocks are enabled in the `debug` property of the Manager (see the “MaxCompiler Manager Compiler Tutorial” document).

The project `tutorial-chap5-example5` contains a Manager with stream status enabled. Launching the application in debug mode with a breakpoint after the call to `DebugWithStatusBlocks()`, and displaying the state of the DFE yields the graph in *Figure 29*. Selecting a stream status block by clicking in the graph displays a number of properties of the stream, including the amount of data transported and its overall status, along with various performance-related characteristics.

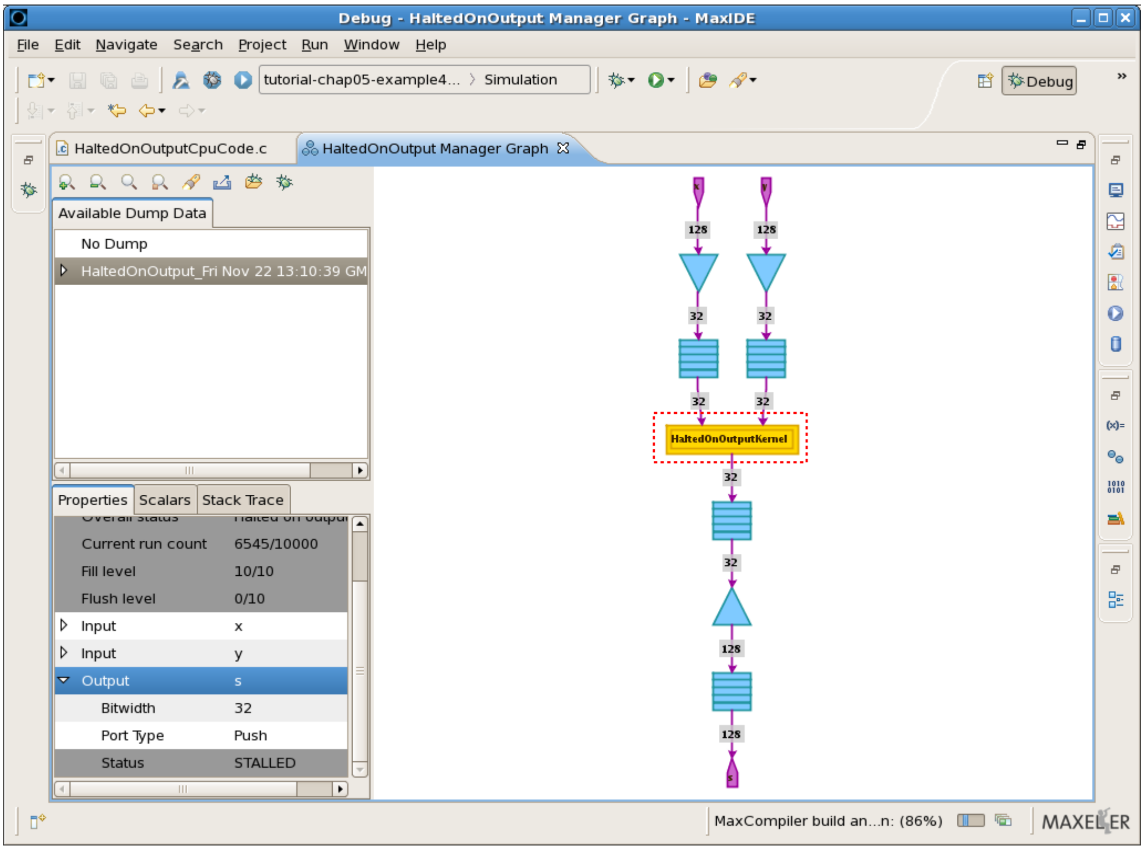


Figure 28: State of the Manager while the function `HaltedOnOutput()` is hanging: the Kernel produces more data than is consumed by the output stream `s`.

5.3.5 Debugging with MaxDebug

MaxDebug is a command line utility that makes it possible to inspect the state of a dataflow engine in a non-interactive fashion while it is running of after is has run. MaxDebug's capabilities are as follows:

- it can obtain the status of a local hardware dataflow engine while it is running or after it has run;
- it can obtain the status of a simulated dataflow engine while it is running or after it has run, as long as the simulated system is running;
- it can be used to display debug snapshots optionally generated by SLiC applications at the completion of each action, both for local engines and engines provided by an MPC-X appliance.

To enable the tool to have this flexibility, some of its features are controlled by environment variables, and these must be modified according to the nature of the dataflow engine to be interrogated:

- for hardware dataflow engines, the environment variable `MAXELEROSDIR` must point to the `MaxelerOS` installation, and the library `$MAXELEROSDIR/lib/libmaxeleros.so` must be preloaded by MaxDebug:

  ```
  export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
  ```

- for simulation, the environment variable `MAXCOMPILERDIR` must point to the simulation libraries in the `MaxCompiler` installation, and the environment must be set as:

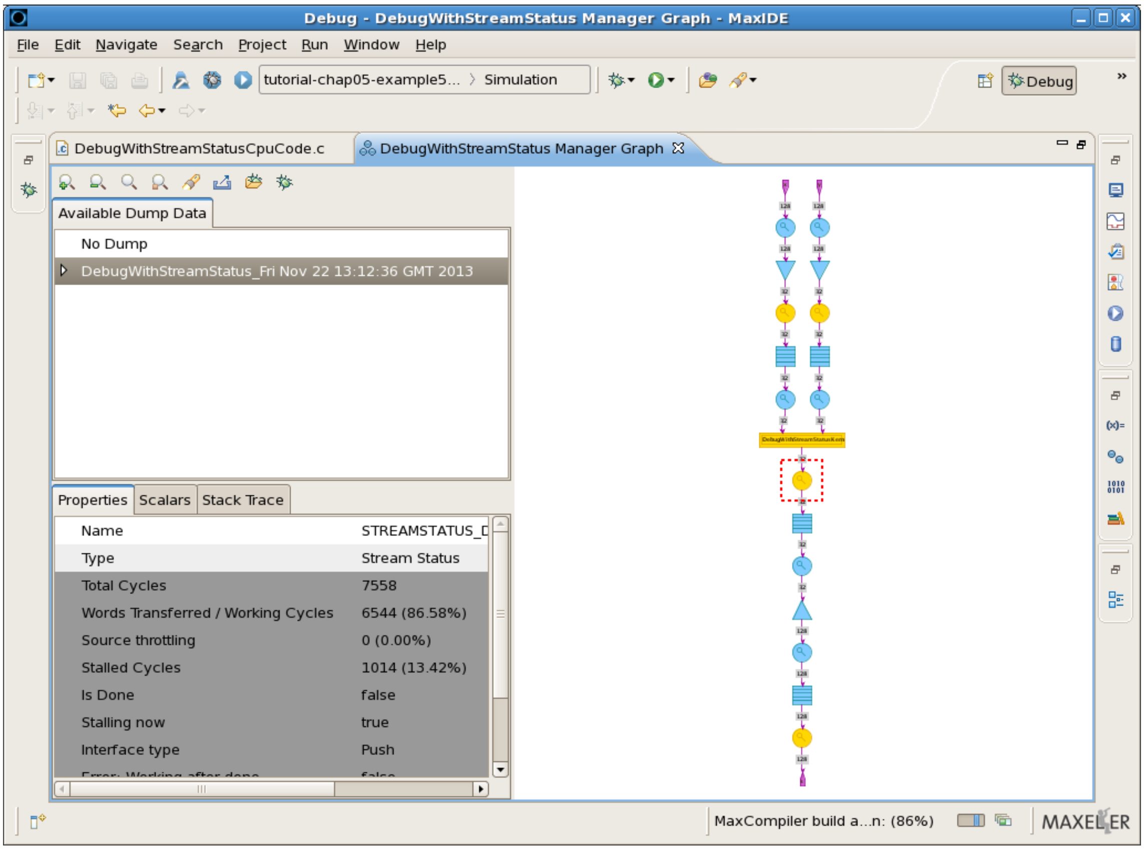


Figure 29: State of the Manager after the function `DebugWithStatusBlocks()` has completed: **stream status blocks** indicate the streams' data throughput and overall status.

```
export MAXELEROSDIR=$MAXCOMPILERDIR/lib/maxeleros-sim
export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

Further, in the case of simulation, the name of of the simulated engine must be specified to MaxDebug by way of the `-d` option. This name is based on the socket name of the simulated system, which is available in the `Simulator` tab of the Run Rule editor of the corresponding project.

Listing 16: MaxDebug command-line options

```
$ maxdebug
MaxDebug version 2014.1

Usage:
        maxdebug [-v] [-g <prefix>] [-s <prefix>] [-n] [-r]
            [-k <kernel>] [-L|-a <name>|-i <id>] [-R <mpcx>]
            [-d <device>] [<maxfile>]
where:
        -r          dump scalar inputs / mapped register values
        -m          dump mapped memories
        -g          draw a manager graph annotated with runtime information
```

```
                    as a .png file for every device
        -k <kernel> limit output to the given kernel
        -x          print all numbers in hexadecimal format
        -v          more verbose output
        -s          draw the static manager graph from the .max file
        -d <device> use <device> when debugging the design (e.g sim0:sim,
                    /dev/maxeler0, or index if used with -a, -i or -f)
        -f <file>   file containing the maxdebug snapshot to use
        -a <action> name of the debug snapshot to retrieve from an MPC-X
                    appliance
        -i <id>     id of the debug snapshot to retrieve from an MPC-X
                    appliance
        -L          retrieve a list of maxdebug snapshots held by an MPC-X
                    appliance
        -R <mpcx>   use <mpcx> as the IP address or hostname of an MPC-X
                    appliance
        <maxfile>   bitstream .max file (mandatory, unless -L is present)
```

Kernel halted on input on a simulated dataflow engine Running the Simulation Run Rules in the example project `tutorial-chap5-example3`, as above, produces applications that hang. To investigate this problem using MaxDebug, we open a terminal, change directory to the root of the MaxCompiler project, and set the environment for simulation:

```
$ export MAXELEROSDIR=$MAXCOMPILERDIR/lib/maxeleros-sim/
$ export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

We can then launch MaxDebug:

```
$ maxdebug -d jHaltedOnInput0:jHaltedOnInput -g graph RunRules/Simulation/
   maxfiles/HaltedOnInput.max
```

Here, the arguments supplied to MaxDebug are:

- `-d jHaltedOnInput0:jHaltedOnInput` specifies the name of the engine, which is constructed as `<engine-name>:<socket-name>`, based on the index of the engine in the simulated system and the socket name of the simulated system.

  In this instance, the index of the engine is “0” (there being only one engine), and the socket name of the simulated system is “`jHaltedOnInput`”.

  For a hardware dataflow engine, this argument is not required.

- `-g graph` instructs MaxDebug to output an image of the state of the design. The generated file will have “`graph`” as a prefix;

- `RunRules/Simulation/maxfiles/HaltedOnInput.max` is the path to the `.max` file.

The console output of this command is shown in *Listing 17* and the `.png` file shown in *Figure 30* is produced.

Listing 17: MaxDebug console output

```
MaxDebug version 2014.1

============================
Kernel : HaltedOnInputKernel
============================

Kernel summary
--------------
Name              : HaltedOnInputKernel
Fill level        : 10 / 10
Flush level       : 0 / 10
Flushing          : False
Ran for           : 9999 / 15000 cycles
Derived status    : Halted on input


Stream summary
--------------
Name  Id    Type    #Outstanding Reads      Derived Status
----  --    ----    ------------------      --------------
 x     0    input          1               reading / no data available
 y     1    input          1               reading / no data available
 s     0    output                         writing
```

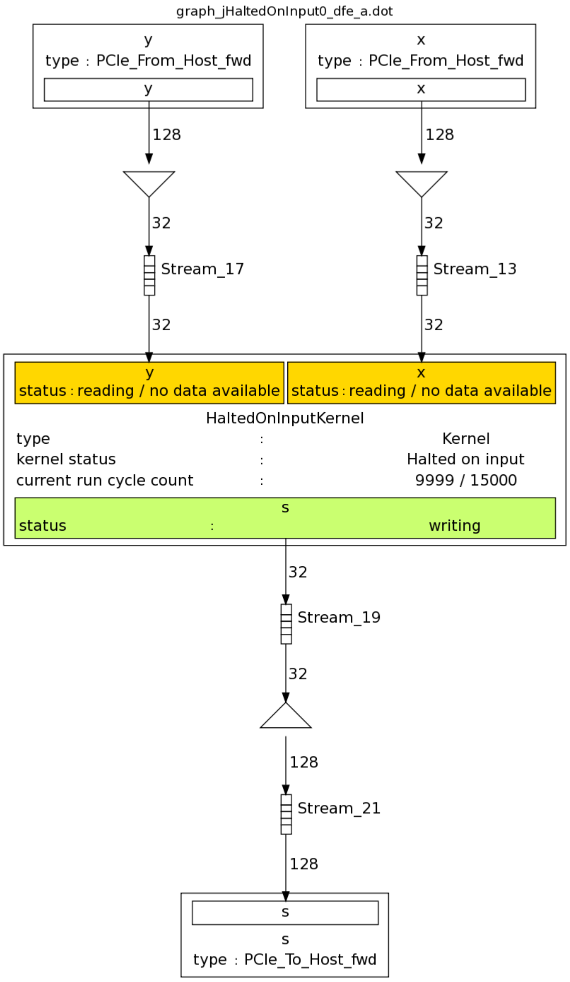


Figure 30: State of the Manager while the function `HaltedOnInput()` is hanging, as represented by `MaxDebug`.

Saving MaxDebug snapshots for later use In the sections above, the information provided by MaxDebug was taken directly from an engine at runtime. In certain cases, we may want to save a snapshot of the content of the DFE for later use. To do this, we instruct SLiC to save a snapshot automatically on completion of an action, by setting `default_maxdebug_mode` in the `SLIC_CONF` environment variable:

```
$ env SLIC_CONF="default_maxdebug_mode=MAX_DEBUG_ALWAYS" ./HaltedOnInput
```

In SLiC's dynamic interface, the same result can be achieved on a per-action basis:

```
void max_set_debug(
        max_actions_t    *actions,
        const char       *name,
        max_debug_mode_t debug_mode);
```

where the enum `debug_mode` must be one of:

- `MAX_DEBUG_NEVER`: where no debug snapshot should be saved for the action;
- `MAX_DEBUG_ON_ERROR`: where a debug snapshot should only be saved in case of stall;
- `MAX_DEBUG_ALWAYS`: where SLiC should always produce a debug snapshot;

and where `name` is used to identify the snapshot.

In SLiC's basic and advanced static interfaces, the MaxFile stem is used as the name of the snapshot.

On completion of the actions, the debug snapshots are saved in the default debug directory with file names of the form "`maxdebug_NAME.TIMETAG`", where `NAME` is the parameter specified in the call to `max_set_debug` or the MaxFile stem name, and where `TIMETAG` is a timestamp used to make the file unique. These snapshots may be examined by the `maxdebug` command, using an additional `-f <file>` flag to specify the snapshot file to use, for example:

```
$ maxdebug -r -f <snapshot> <maxfile>
```

Where arrays of engines are used, the content of a specific engine may be obtained by using the `-d <index>` option, where the index ranges between 0 and $N-1$ for an array of N devices.

6

Dataflow Variable Types

A Type is defined as the range of significance of a [...] function.
– Bertrand Russell, Logic and Knowledge, 1971

Variables in a dataflow program are in fact portals through which streams of numbers pass while being represented in a specific way by zeros and ones. Contrary to CPUs, a multiscale dataflow computer allows us to use zeros and ones in any way we like (or don't like) to represent the number. "Multiscale" dataflow computing means that if necessary, one can extend the optimizations and numerical algorithm design all the way to the bit level. For example, a loop variable that goes from 0 to 100 only really needs 7 bits.

Luckily, MaxCompiler can infer most variables, and initially, an implementation only needs to declare input variables. MaxCompiler uses the declarations of input and output variables to generate the SLiC interface for the Kernel implementation. The basic type for any other variable is `DFEVar`.

From an algorithmic perspective, any variable in a program has a range and precision requirements. From a practical perspective, we are used to assigning standard types such as `int`, `float`, and `double` to variables in our calculations. Consequently, MaxCompiler offers `DFEInt` and `DFEFloat`, corresponding to matching variable declarations in the resulting SLiC interface.

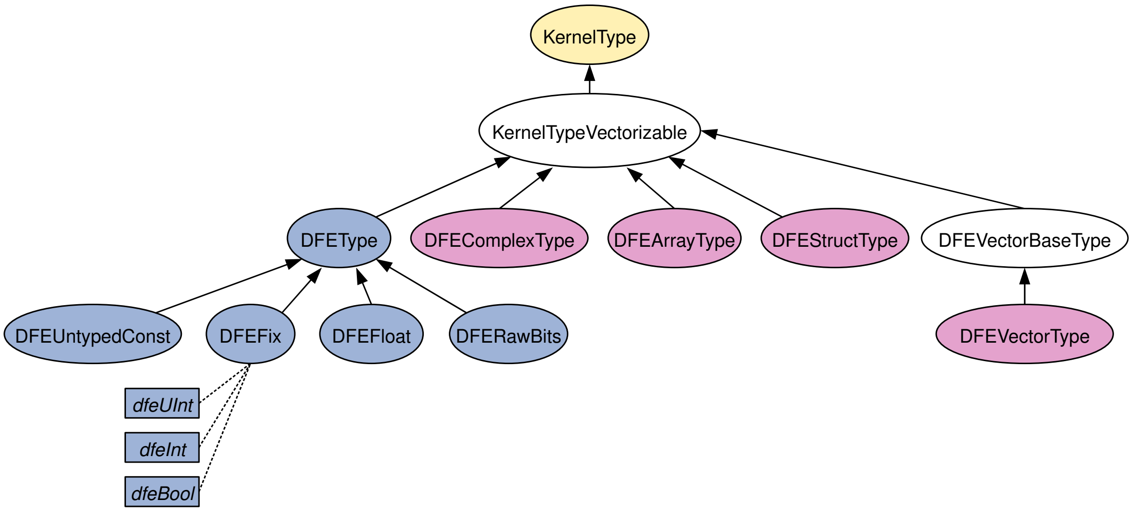


Figure 31: Class hierarchy for data types

Figure 31 gives an overview of the class hierarchy for data types used in the Kernel Compiler. There are two categories of Kernel types within the Kernel Compiler:

Primitive types (for example `DFEFloat`) inherit from `DFEType` and can be used with `DFEVar` variables as we have seen in previous chapters.

Composite types (for example `DFEComplexType`) do not extend `DFEType`: these internally translate operations into operations on multiple `DFEVar` variables.

MaxCompiler and MaxIDE use an extended version of Java called MaxJ which adds operator overloading semantics to the base Java language. This enables an intuitive programming style, for example with arithmetic expressions including `DFEVar` objects being possible. MaxJ source files have the `.maxj` file extension to differentiate them from pure Java .

6.1 Primitive types

Every `DFEVar` instance has an associated representation type which is represented by a `DFEType` object. These `DFEType` objects are usually assigned automatically by the Kernel Compiler or you can specify them through casting or specification of I/O types. You can query the type of a `DFEVar` object at any point using the `DFEVar.getType` method.

When it is necessary to explicitly specify a type for `DFEVar`, for example when creating an input or optimizing a component, instances of `DFEType` are created indirectly using one of the following methods which are available from the `Kernel` class:

- `dfeFloat(int exponent_bits, int mantissa_bits)`
  Creates a floating-point type parameterized with mantissa and exponent bit-widths. With 8 exponent bits and 24 mantissa bits, the format is equivalent to single-precision floating point. Similarly double-precision has an exponent of 11 bits and a mantissa of 53 bits.

- `dfeFixOffset(int num_bits, int offset, SignMode sign_mode)`
  Creates a fixed-point type with parameterizable size and binary point offset and a choice of unsigned (`SignMode.UNSIGNED`) or two's complement (`SignMode.TWOSCOMPLEMENT`) modes for number representation.

Listing 18: Kernel demonstrating types and type casting (TypeCastKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 6       Example: 1       Name: Typecast
 * MaxFile name: TypeCast
 * Summary:
 *         Kernel that casts from an unsigned int to a float and back.
 */

package typecast;

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

class TypeCastKernel extends Kernel {

    TypeCastKernel(KernelParameters parameters) {
        super(parameters);

        // Type declarations
        DFEFloat singleType = dfeFloat(8, 24); // useful: makes it easy to change types consistently

        // Input
        DFEVar a = io.input("a", dfeUInt(8));
        DFEVar x = io.input("x", singleType);

        // Cast input 'a' from unsigned 8-bit integer to
        // IEEE single precision float using named type,
        // then add 10.5
        DFEVar result = a.cast(singleType) + 10.5;

        // Cast input 'x' from IEEE single precision float
        // to unsigned 8-bit integer using explicit type
        DFEVar x_int = x.cast(dfeUInt(8));

        // Output
        io.output("b", result, singleType);
        io.output("y", x_int, dfeUInt(8));
    }

}
```

- `dfeUInt(int bits)`
  An alias for `dfeFixOffset(bits, 0, SignMode.UNSIGNED)`.

- `dfeInt(int bits)`
  An alias for `dfeFixOffset(bits, 0, SignMode.TWOSCOMPLEMENT)`.

- `dfeBool()`
  An alias for `dfeFixOffset(1, 0, SignMode.UNSIGNED)`. `dfeBool` can safely be used for all Boolean operations with numeric values `1` and `0` representing true or false respectively.

- `dfeRawBits()`
  A Kernel type representing a binary word of user-defined length which does not have a specific Kernel data type. `DFERawBits` streams are used to prevent invalid operations being performed inadvertently on collections of bits for which operator rules are not valid. For example, it is invalid to apply any floating-point operation to the result of `DFEVar.slice`. Streams of `DFERawBits` can

be cast to any other type of the same bit-width with no overhead.

The dataflow program in *Listing 18* defines a number of primitive-type streams and operations on their data types. The example creates an input stream and specifies its data type by calling `dfeUInt(8)`:

```
        DFEVar a = io.input("a", dfeUInt(8));
```

This `dfeUInt` method call creates and returns an object of class `DFEFix`, which directs the input method to create a `DFEVar` object for this input with a type of unsigned 8-bit integer.

Developers may explicitly change the type of data as it flows through the graph using type casts. Type casts are typically used to change number representations as variables are reused in a dataflow program. Type casts are introduced by calling the `cast` method on `DFEVar` instances, for example:

```
        DFEVar x_int = x.cast(dfeUInt(8));
```

In addition to integers and floating point numbers, multiscale dataflow also supports fixed point numbers (`DFEFix`). Fixed point numbers allow us to store fractions and for small range are a lot more efficient than floating point numbers. The main drawback of fixed point numbers is that the algorithm designer has to think through range and precision requirements on the variables, and consider the distribution of values for a particular variable very carefully. For iterating algorithms it is also necessary to occasionally re-normalize the values to avoid dealing with a large range of values.

> Type-casting operations, particularly those converting between floating and fixed point types, can be costly in terms of resource usage. The use of type casts should therefore be minimized.

To avoid unexpectedly large designs, MaxCompiler does not automatically infer type casts but rather insists that type casting be used explicitly where necessary. For example, adding a floating point number to a fixed point number leads to a compilation error and prompts you to explicitly cast one input type to match the other.

> When casting between a floating-point number and an integer, MaxCompiler rounds to the nearest integer. This is different to the behavior in many other programming languages (such as C/C++, Java and Fortran), where the floating-point number is truncated.

Users need to also be aware that whenever constants are used in a Kernel design without an explicit type, the Kernel Compiler assigns these constants a type of `DFEUntypedConst`. An example of an untyped constant can be seen in the example:

```
        DFEVar result = a.cast(singleType) + 10.5;
```

This `DFEUntypedConst` type also propagates forward through the Kernel graph until the type for an operation can be determined. In the example, the addition operator works with the `DFEVar` called `a`, which is cast to a type of single precision floating-point. In this case the value 10.5 is also interpreted as a single-precision floating-point number. In some situations, a Kernel design may use an untyped constant in a context where it is not possible to determine the types to use and it is not possible to forward-propagate the untyped constant type. For example, with a conditional assignment where both options are constants we get a compile-time error until at least one of the inputs is explicitly given a type.

6.2 Composite types

While `DFEVar` and `DFEType` provide the fundamental design elements needed to create Kernel designs, more succinct solutions can often be made using composite constructs.

The composite types available in the Kernel Compiler are `DFEComplexType`, `DFEVectorType`, and `DFEStructType`.

6.2.1 Composite complex numbers

Listing 19 shows a Kernel design that takes in two complex numbers, adds them together and multiplies the result by the real number 3.

A complex number type is defined to be used throughout the design using an `DFEFloat` object returned from a `dfeFloat` call:

```
public DFEComplexType cplxType =
    new DFEComplexType(dfeFloat(8,24));
```

This declares that the real and imaginary parts for instances of complex numbers created with this type are to be stored as floating-point numbers. We could equally have used more or fewer bits for our floating-point type or a `DFEFix` type.

This type is used to specify the types of the inputs and output:

```
    DFEComplex cplxIn1 = io.input("cplxIn1", cplxType);
    DFEComplex cplxIn2 = io.input("cplxIn2", cplxType);
```

```
    io.output("cplxOut", result, cplxType);
```

In order to actually refer to our complex variables we use Java objects of type `DFEComplex`. The `DFEComplex` class is conceptually paired with the `DFEComplexType` class in the same way that `DFEVar` and `DFEType` are paired.

`DFEComplex` type supports the familiar multiplication, subtraction and addition operators:

```
    DFEComplex result = (cplxIn1 + cplxIn2) * 3;
```

In order to run the example on a DFE, we stream the data into the dataflow engine in the correct format. The is done as a continuous stream of single-precision floating-point numbers. Pairs of floating-point numbers are interpreted as real followed by imaginary parts.

In the CPU code for this example (*Listing 20*), two streams of complex numbers are initialized and streamed to the dataflow engine. The first stream contains the complex numbers {`(1 + 2i), (3 + 4i)`} and the second stream contains {`(5 + 6i), (7 + 8i)`}:

```
float cplxIn1[] = {
    1, 2, // (real, imaginary)
    3, 4 };

float cplxIn2[] = { 5, 6, 7, 8 };
```

Listing 19: Kernel demonstrating complex number support (ComplexKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 6      Example: 2      Name: Complex
 * MaxFile name: Complex
 * Summary:
 *        Kernel that performs complex arithmetic.
 */
package complex;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEComplex;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEComplexType;

class ComplexKernel extends Kernel {
	public DFEComplexType cplxType =
		new DFEComplexType(dfeFloat(8,24));

	ComplexKernel(KernelParameters parameters) {
		super(parameters);

		// Inputs
		DFEComplex cplxIn1 = io.input("cplxIn1", cplxType);
		DFEComplex cplxIn2 = io.input("cplxIn2", cplxType);

		DFEComplex result = (cplxIn1 + cplxIn2) * 3;

		// Output
		io.output("cplxOut", result, cplxType);
	}
}
```

Listing 20: Main function for CPU code demonstrating complex number I/O (ComplexCpuCode.c).

```
int main()
{
    const int size = 2;
    float cplxIn1[] = {
        1, 2, // (real, imaginary)
        3, 4 };

    float cplxIn2[] = { 5, 6, 7, 8 };

    float expectedOut[] = {
        18, 24, // (real, imaginary)
        30, 36 };

    float *actualOut = malloc(sizeof(expectedOut));
    memset(actualOut, 0, sizeof(expectedOut));

    ComplexCPU(
        size,
        cplxIn1,
        cplxIn2,
        actualOut);

    printf("Running DFE.\n");
    Complex(
        size,
        cplxIn1,
        cplxIn2,
        actualOut);

    int status = check(actualOut, expectedOut, size);
    if (status)
        printf("Test failed.\n");
    else
        printf("Test passed OK!\n");

    return status;
}
```

6.2.2 Composite vectors

Similarly to `DFEComplex`/`DFEComplexType` variables, `DFEVector`/`DFEVectorType` variables allow multiple variables to be grouped together. These vectors can be used to hold any type of Kernel data-type.

> As `DFEVectors` are a composite type, this means they are effectively just a wrapper around multiple `DFEVars` and so are not appropriate for storing chunks of data in a DFE. Creating a `DFEVector` with more than a few elements will result in excessive DFE resource usage. For details on how to store data in a DFE, see *section 12*.

Listing 21: Kernel demonstrating vector support (VectorsKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 6
 * Example: 3
 * Summary:
 *        Kernel that doubles values in a vector.
 */
package vectors;

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

class VectorsKernel extends Kernel {

    VectorsKernel(KernelParameters parameters, int vectorSize) {
        super(parameters);

        DFEVectorType<DFEVar> vectorType =
            new DFEVectorType<DFEVar>(dfeUInt(32), vectorSize);

        // Input
        DFEVector<DFEVar> inVector = io.input("inVector", vectorType);

        // Explicitly double each vector element
        DFEVector<DFEVar> doubledVector =
            vectorType.newInstance(this);

        for (int i = 0; i < vectorSize; i++)
            doubledVector[i] <== inVector[i] * 2;

        // Double vector by multiplying with another
        // (constant) vector [2, 2].
        DFEVector<DFEVar> quadroupledVector =
            doubledVector * constant.vect(2, 2);

        // Double vector by multiplying all elements by a single value
        DFEVector<DFEVar> octupledVector =
            quadroupledVector * 2;

        //Output
        io.output("outVector", octupledVector, vectorType);
    }

}
```

Listing 21 shows a Kernel which uses a vector of two `DFEVar` variables and multiplies this pair of streams by 8.

We first create the `DFEVectorType` to represent this vector:

```
DFEVectorType<DFEVar> vectorType =
    new DFEVectorType<DFEVar>(dfeUInt(32), vectorSize);
```

The type is parameterized with the type of element the vector holds in angle-brackets, in this case `DFEVar`, and the type of the contained element, which in this case is an unsigned 32-bit integers.

This `vectorType` is used to create an input which gives us our initial vectorized stream of `DFEVar` pairs.

```
DFEVector<DFEVar> inVector = io.input("inVector", vectorType);
```

Now we have our input, the example multiplies the vector by 2, three different ways to demonstrate various uses of `DFEVector`.

The first way we double our `DFEVector` is to operate on each of the elements in the vector individually (similarly to how we might work with an array in C or Java). To do this, we must first declare a new "sourceless" `DFEVector` instance:

```
DFEVector<DFEVar> doubledVector =
    vectorType.newInstance(this);
```

This `DFEVector` is sourceless until each of its elements are connected with streams of computation. We connect these up using a Java for-loop as follows:

```
for (int i = 0; i < vectorSize; i++)
    doubledVector[i] <== inVector[i] * 2;
```

In this loop each element of the `DFEVector` is connected to a stream indexed from the input `DFEVector` using the <== operator (equivalent to `x.connect(y)`).

The above snippets demonstrate how to access individual elements of a vector, however in many cases this can be quite cumbersome. In this case the multiplication operator for `DFEVector` is overloaded such that a new `DFEVector` can be constructed by multiplying two `DFEVectors` together as follows:

```
DFEVector<DFEVar> quadroupledVector =
    doubledVector * constant.vect(2, 2);
```

The `constant.vect()` methods create new and constant `DFEVector` streams.

Again this example can be simplified further as we are multiplying all elements in the vector with the same value. As such, we can take advantage of overloaded operators in `DFEVector` allowing an operation to be applied to all elements using a single source stream as follows:

```
DFEVector<DFEVar> octupledVector =
    quadroupledVector * 2;
```

Finally, we connect our resultant vector to the output:

```
io.output("outVector", octupledVector, vectorType);
```

In order to run the example on a DFE, we stream data into the dataflow engine as a continuous block of 32-bit unsigned integers. In the following snippets from the CPU code for this example, a stream of input vectors of 2 elements is initialized with incrementing values and transferred to the dataflow engine. The series of vectors is therefore initialized to `{{0,1}, {2,3}, {4,5}...}`.

```
size_t sizeBytes = vectorSize * streamSize * sizeof(uint32_t);
uint32_t *inVector = malloc(sizeBytes);
```

```
for (int i = 0; i < vectorSize * streamSize; i++) {
    inVector[i] = i;
}
```

```
Vectors(streamSize, inVector, expectedVector);
```

6.3 Available dataflow operators

Dataflow operators are overloaded operators that enable us to simply write expressions which then get translated into dataflow graphs, and finally into DFE configurations. Both primitive and composite streams implement overloaded operators. The operators that are available for a stream depend on the underlying Kernel type of that stream (see *Table 2*).

Kernel Type	=	+[1]	−[1]	*[1]	/[1]	−[1]	<, <=, >, >=	<<, >>, >>>[1]	&, ^, |[1]	?:	~	[][5]	<==[6]	===, !==
`DFEFix`, `DFEInt`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[2]	✓	✓[3]	✓	✓
`DFEUInt`, `dfeBool`	✓	✓	✓	✓	✓		✓	✓	✓	✓[2]	✓	✓[3]	✓	✓
`DFEFloat`	✓	✓	✓	✓	✓	✓	✓					✓[3]	✓	✓
`DFERawBits`	✓							✓	✓	✓[2]	✓	✓[3]	✓	✓
`DFEComplexType`	✓	✓	✓	✓	✓	✓							✓	✓
`DFEStructType`	✓												✓	✓
`DFEVectorType`	✓											✓[4]	✓	✓

[1] Includes compound assignment operators (+=, *=, >>= etc.).
[2] Kernel Type must be 1-bit wide.
[3] Equivalent of `.slice(i)` to select a single bit.
[4] Returns indexed element from the `DFEVector` stream.
[5] Not allowed as left-hand-side in a statement (i.e cannot do `x[y]=z;`).
[6] <== is the *connect* operator, where `x<==y` is equivalent to `x.connect(y)`.
[7] Contained type of the multi-pipe stream must be 1-bit wide.

Table 2: Overloaded operators available by Kernel Type.

The == and != operators have a special meaning in Java: they compare *reference equality*. To compare the equality of streams, use the operators === and !==.

The logical NOT (!), logical AND (&&) and logical OR (||) operators are *not* overloaded for streams: it is not possible to replicate the conditional evaluation (or "short-circuiting") semantics in a dataflow program. You can directly replace these operators with the bit-wise AND & and bit-wise OR | operators in many circumstances or use the ternary-if ?: operator where conditional behavior is required.

The modulus (%), increment (++) and decrement (--) operators are not implemented for any streams.

Exercises

Exercise 1: Vectors

Design and test a Kernel that takes an input vector of four 32-bit unsigned integers, reverses the order of the elements in the vector and sends them back out again.

7

Scalar DFE Inputs and Outputs

In addition to streaming data back and forth between CPU and DFE, we also have the option to transfer single values to and from the DFE at runtime. For example, the coefficients of the three-point moving average can be declared as scalar inputs and can be set at runtime by the CPU. You can also think of these **scalar inputs** as a set of registers that is mapped into the CPUs address space.

Or put in another way, in contrast to a parameter passed to the constructor of the Kernel class, which requires recompilation, a scalar input can be set by the CPU application dynamically at runtime.

The dataflow program for a Kernel with a scalar input is shown in *Listing 22*. *Figure 32* displays the corresponding Kernel graph.

In this example, we use the method `io.scalarInput` to define `b` as a scalar input:

```
        DFEVar b = io.scalarInput("b", singleType);
```

The method takes two parameters: a string for the name and its data type. The scalar input is represented in the Kernel graph *Figure 32* as two concentric rectangles.

MaxCompiler automatically adds the scalar inputs to the SLiC interface function for running the design, as shown in the CPU code for this example:

```
    AddScalar(size, scalarIn, dataIn, dataOut);
```

Transferring single scalar inputs and outputs back and forth from and to the DFE is not fast. The

Listing 22: Program for the adder Kernel (AddScalarKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 7        Example: 1       Name: Add scalar
 * MaxFile Name: AddScalar
 * Summary:
 *          Kernel that adds a scalar to a stream of values.
 */

package addscalar;

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

class AddScalarKernel extends Kernel {

    AddScalarKernel(KernelParameters parameters) {
        super(parameters);

        // Typedef
        DFEFloat singleType = dfeFloat(8, 24);

        // Stream Input
        DFEVar a = io.input("a", singleType);

        // Scalar Input
        DFEVar b = io.scalarInput("b", singleType);

        DFEVar result = a + b;
        // Stream Output
        io.output("c", result, singleType);
    }
}
```

way to use scalar inputs effectively is to set all scalar inputs in one go, then to run the kernel for a long time, and then possibly to read all scalar output results back to the CPU. When accessing many scalar IO variables with a single SLiC call, all the scalar values are streamed in a single transaction.

Exercises

Exercise 1: Scalar inputs

Given an example to calculate a moving average of a stream, modify it to include a scalar input indicating the length of the input stream.

Test the Kernel design in simulation and a DFE with streams of different lengths.

Hint: You may find the method `control.count.simpleCounter(int bit_width)` more useful than the `control.count.simpleCounter(int bit_width, int max)` method used in the original example.

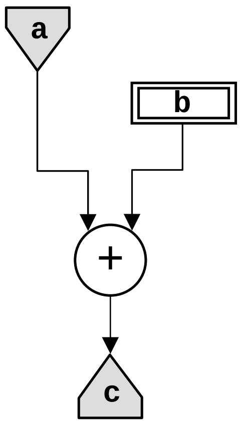


Figure 32: Kernel graph for the adder, as created by the program in *Listing 22*

Listing 23: Host code for the adder Kernel (AddScalarCpuCode.c).

```c
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 7        Example: 1        Name: Add scalar
 * MaxFile Name: AddScalar
 * Summary:
 *          Example showing the use of scalar inputs.
 */
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include "Maxfiles.h"
#include <MaxSLiCInterface.h>

void generateData(int size, float *dataIn)
{
    for (int i = 0; i < size; i++)
        dataIn[i] = i;
}

void AddScalarCPU(int size, float scalarIn, float *dataIn, float *dataOut)
{
    for (int i = 0; i < size; i++)
        dataOut[i] = dataIn[i] + scalarIn;
}

int check(int size, float *dataOut, float *expected) {
    int status = 0;
    for (int i = 0; i < size; i++)
        if (dataOut[i] != expected[i]) {
            fprintf(stderr, "Output data @ %d = %f (expected %f)\n",
                i, dataOut[i], expected[i]);
            status = 1;
        }
    return status;
}

int main()
{
    const int size = 1024;
    int sizeBytes = size * sizeof(float);
    float *dataIn   = malloc(sizeBytes);
    float *dataOut = malloc(sizeBytes);
    float *expected = malloc(sizeBytes);
    float scalarIn   = 5.0;

    generateData(size, dataIn);

    printf("Setting scalar and running DFE.\n");
    AddScalar(size, scalarIn, dataIn, dataOut);

    AddScalarCPU(size, scalarIn, dataIn, expected);

    int status = check(size, dataOut, expected);
    if (status)
        printf("Test failed.\n");
    else
        printf("Test passed OK!\n");

    return status;
}
```

8

Navigating Streams of Data

A Stream is a steady succession of words or events.

– Webster Dictionary

In some of the earlier examples in this document, we have seen the use of `stream.offset` to access values at different points in the data stream. The various forms of this method are key to making efficient dataflow computing implementations and are the focus of this section.

8.1 Windows into streams

A core concept of dataflow computing is operating on windows into data streams. The data window is held in on-chip memory on the dataflow engine, minimizing off-chip data transfers.

Stream offsetting allows us to access data elements within a stream relative to the current location. The distance from the largest to the smallest offset forms the window of data that is held in the dataflow engine. *Figure 33* shows a data stream `A` over three ticks. In the first tick, the current data item (or **head** of the stream) has a value of 23. A dataflow program accessing the head of the stream and also a data item four elements into the past (with a value of 11 in tick 1 of *Figure 33*) creates a window of size five into stream `A`. On each tick, the data in the stream moves through the window. In contrast to conventional software, stream offsetting makes the memory cost of accessing non-local elements explicit and allows applications to be explicitly architected to minimize off-chip memory access.

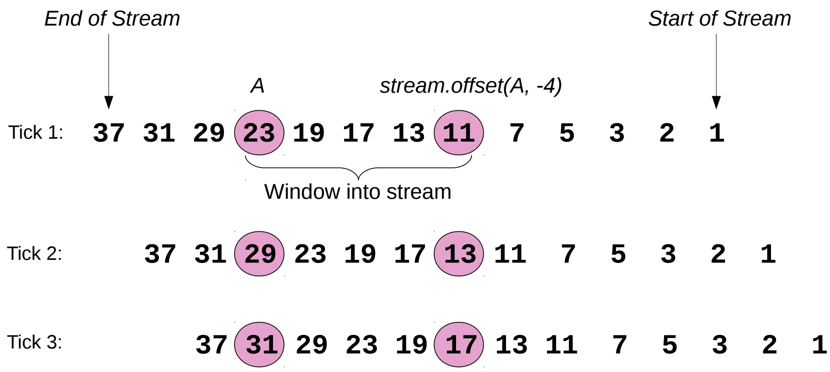


Figure 33: Stream offsets form a window into a data stream

Any conventional software array operation can be expressed in terms of stream offsets. For example, *Figure 34* shows one way in which the collection of points for a 3x3 2-dimensional convolution operator can be expressed in terms of offsets. The element `A` is the head of the stream, and the other points are read from the past (negative) or the future (positive) of the stream. The dotted line shows the order in which the data is streamed into the Kernel.

The total size of the window into the 8x8 data set in *Figure 34* amounts to 19 data items (all the items highlighted in gray or pink): 9 items from the past of the stream, 9 items from the future and one for the current item. The total size of the window in a 2D offset like this depends on the width of the 2D data set being streamed into the Kernel: the wider the data set, the larger the required window to buffer the line data.

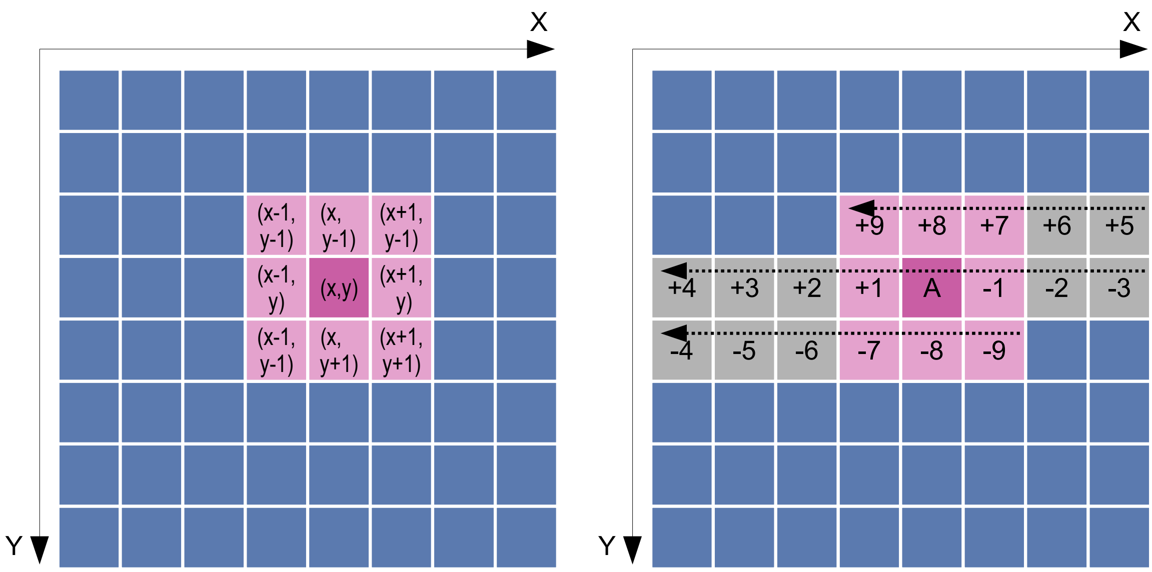


Figure 34: Points for a 2D convolution expressed in terms of coordinates (left) and offsets (right)

MaxCompiler supports three kinds of stream offsets that differ in how their size is specified:

- **static offsets** have a size fixed at compile-time
- **variable offsets** have their size set at run time before a stream is processed
- **dynamic offsets** have their sizes set at run time during stream processing

In this section, we will see how to use these different kinds of offsets and discuss how to decide which type to use.

8.2 Static offsets

Listing 24 shows one of the more straight-forward uses of stream offsets: to retrieve values immediately adjacent to the current value of the stream. Our example takes an input stream and sums every three elements to create a new output stream.

Note that the dataflow program in *Listing 24* does not specifically handle boundary cases and, therefore, the behavior of this Kernel at the beginning and end of the stream is undefined.

Positive stream offsets return values from the *future* of the stream; negative offsets return values from the *past* of the stream. Future values are later rescheduled by the compiler.

Our example implies a window of size 3 into the input stream from the furthest positive point, +1, to the furthest negative offset, -1:

```
DFEVar inPrev = stream.offset(inStream, -1);
DFEVar inNext = stream.offset(inStream, 1);

DFEVar result = inPrev + inStream + inNext;
```

The storage cost for the implementation of this Kernel in a dataflow engine is therefore 3 elements. Luckily, a typical DFE has many MBs of on-chip data.

8.3 Variable stream offsets

Static offsets are simple and can be highly optimized by the compiler, however they are not very flexible. If we want to change the length of an offset in a design, we need to recompile that bitstream, which is quite inconvenient.

Consider, the more complicated example shown in *Listing 25*. This dataflow program describes a Kernel implementing an averaging filter that averages 9 points in 2 dimensions. The input data set is of size $nx \times ny$ and the data elements are streamed row-by-row. The position in the stream at offset 0 is defined as the coordinate (x, y) and the filter collects values from $(x-1, y-1)$ to $(x+1, y+1)$ in a 3×3 grid.

In order to collect points in two dimensions the offset becomes a function of nx, the size of the dimension x, as we can see from the following code:

```
// Extract 8 point window around current point
DFEVar window[] = new DFEVar[9];
int i = 0;
for (int x=-1; x<=1; x++)
    for (int y=-1; y<=1; y++)
        window[i++] = stream.offset(inStream, y*nx+x);
```

Listing 24: Simple example of using stream offsets (SimpleOffsetKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial)
 * Chapter: 8      Example: 1      Name: Simple offset
 * MaxFile name: SimpleOffset
 * Summary:
 *      Kernel that applies an offset to a stream.
 */

package simpleoffset;

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

class SimpleOffsetKernel extends Kernel {

    SimpleOffsetKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar inStream = io.input("inStream", dfeFloat(8, 24));

        // Offsets and Calculation
        DFEVar inPrev = stream.offset(inStream, -1);
        DFEVar inNext = stream.offset(inStream, 1);

        DFEVar result = inPrev + inStream + inNext;

        // Output
        io.output("outStream", result, dfeFloat(8, 24));
    }

}
```

Now suppose we want to change the size of the 2D data set the dataflow engine is filtering: we can change ny easily, since it is not used in the Kernel description at all, however if we change nx we need to recompile.

A solution to this problem is shown in *Listing 26*. This performs the same operation as *Listing 25*, but allows the value of nx to change at run time. The program declares a variable nx using the `stream.makeOffsetParam` method:

```
        OffsetExpr nx = stream.makeOffsetParam("nx", 3, nxMax);
```

This nx variable forms the basis of an **offset expression** to specify an offset size which can be varied at run-time.

When nx is declared, it is specified with upper and lower bounds (in this case $nxMax$ and 3 respectively). At run time, nx may only be varied within these bounds. Specifying reasonable ranges for minimum and maximum values for stream offset expression parameters is very important, because the compiler optimizes the dataflow engine to allocate storage as necessary for precisely that range. Specifying very wide bounds can result in very high on-chip resource usage.

Listing 25: A 2D 9-point averaging filter using static stream offsets (TwoDAverageStaticKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial)
 * Chapter: 8     Example: 2     Name: Two Dimensional Average Static
 * MaxFile name: TwoDAverageStatic
 * Summary:
 *     Kernel that averages within an 8-point window.
 */

package twodaveragestatic;

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

class TwoDAverageStaticKernel extends Kernel {

    TwoDAverageStaticKernel(KernelParameters parameters, int nx) {
        super(parameters);

        // Input
        DFEVar inStream = io.input("inStream", dfeFloat(8, 24));

        // Extract 8 point window around current point
        DFEVar window[] = new DFEVar[9];
        int i = 0;
        for (int x=-1; x<=1; x++)
            for (int y=-1; y<=1; y++)
                window[i++] = stream.offset(inStream, y*nx+x);

        // Sum points in window and divide by 9 to average
        DFEVar sum = constant.var(dfeFloat(8, 24), 0);
        for (DFEVar dfeVar : window) {
            sum = sum + dfeVar;
        }

        DFEVar result = sum / 9;

        // Output
        io.output("outStream", result, dfeFloat(8, 24));
    }

}
```

Simple linear algebra can be performed on the `OffsetExpr` objects:

```
        DFEVar window[] = new DFEVar[9];
        int i = 0;
        for (int x=-1; x<=1; x++)
            for (int y=-1; y<=1; y++)
                window[i++] = stream.offset(inStream, y*nx+x);
```

`OffsetExpr` variables can only have a limited number of operations performed on them, which include: addition and subtraction of other `OffsetExpr` instances or Java compile-time constants, and multiplication by Java compile-time constants.

Offset expressions must be linear, i.e. of the form $A + Bx + Cy + \ldots$, where A, B and C are constants and x and y are offset parameters.

Listing 26: A 2D 9-point averaging filter using variable stream offsets (TwoDAverageVariableKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 8      Example: 3      Name: Two-dimensional average variable
 * MaxFile name: TwoDAverageVariable
 * Summary:
 *      Kernel that averages within an 8-point window.
 */

package twodaveragevariable;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class TwoDAverageVariableKernel extends Kernel {

    public TwoDAverageVariableKernel(KernelParameters parameters, int nxMax) {
        super(parameters);

        // Input
        DFEVar inStream = io.input(”inStream”, dfeFloat(8, 24));

        OffsetExpr nx = stream.makeOffsetParam(”nx”, 3, nxMax);

        // Extract 8 point window around current point
        DFEVar window[] = new DFEVar[9];
        int i = 0;
        for (int x=-1; x<=1; x++)
            for (int y=-1; y<=1; y++)
                window[i++] = stream.offset(inStream, y*nx+x);

        // Sum points in window and divide by 9 to average
        DFEVar sum = constant.var(dfeFloat(8, 24), 0);
        for (DFEVar dfeVar : window) {
            sum = sum + dfeVar;
        }

        DFEVar result = sum / 9;

        //Output
        io.output(”outStream”, result, dfeFloat(8, 24));
    }

}
```

The offset expression is automatically added as an argument to the SLiC function for running the design. The excerpt below from the CPU code for this example shows the value of nx being set at run time:

```
    TwoDAverageVariable(NX*NY, NX, dataIn, expectedOut);
```

8.3.1 3D convolution example using variable offsets

As only linear expressions are permitted, the approach to creating variable offsets into a 3D volume using offset expressions of `nx` and `ny` and writing `z*nx*ny+y*nx+x` as the offset for z does not compile. To create our 3D offsets, we need offset expressions of `nx` and `nxy` and use `z*nxy+y*nx+x` as the offset for z. `nxy` is then set to `nx*ny` by the CPU.

Example 4 extends the 2D moving average from Example 3 to demonstrate the creation of such offsets in a 3D moving average function that can operate on a variable-sized volume. The full kernel is shown in *Listing 27*.

We first create the two offset expressions, one for `nx` and one for `nxy`.

```
OffsetExpr nx = stream.makeOffsetParam("nx", 3, nxMax);
OffsetExpr nxy = stream.makeOffsetParam("nxy", 3 * nx, nxMax * nx);
```

These are then used to create the 27-point cube window into the input data stream:

```
DFEVar window[] = new DFEVar[27];
int i = 0;
for (int x=-1; x<=1; x++)
    for (int y=-1; y<=1; y++)
        for (int z=-1; z<=1; z++)
            window[i++] = stream.offset(inStream, z*nxy+y*nx+x);
```

8.4 Dynamic offsets

Variable offsets allow the value of an offset to be changed on a per-stream basis, however the offset is fixed for the duration of a stream.

In some applications, it is necessary to change the value of an offset during a stream. One way of achieving this is to use multiple offsets and a **multiplexer** (`control.mux`) to select between them. A multiplexer is a generalized version of the ternary-if operator (`?:`) that allows us to select between multiple streams. The first argument is the control stream that decides which stream to select. For example:

```
DFEVar x = input("x", dfeInt(32));
DFEVar offset = input("offset", dfeUInt(2));
DFEVar y = output("y", dfeInt(32));
y <== control.mux(offset, stream.offset(x, 0), stream.offset(x, 1), stream.offset(x, 2), stream.offset(x, 3));
```

In this example the offset is limited to a range of 0 to 3, and the multiplexer option is satisfactory. However, if the possible range of offsets is larger, then this approach scales poorly in terms of space on the DFE.

Dynamic offsets are offsets where the offset value is specified as an `DFEVar` input that can vary on a tick-by-tick basis at run time. *Listing 28* shows an example Kernel that uses two dynamic offsets to extract two points from an input stream and interpolate between them to generate an output point.

Dynamic offsets are instantiated using the `stream.offset` method. This method is parameterized with the stream to offset, the offset value, the minimum offset value and the maximum offset value (determining the amount of memory needed):

```
stream.offset(KernelObject src, DFEVar offset, int min_offset, int max_offset)
```

In *Listing 28* the two stream offsets both range from `-maxTraceSize` to `maxTraceSize` and are offsets on the same input stream `inStream`. However the value of each offset on each tick differs (`lowerPointPos` and `upperPointPos`):

```
DFEVar lowerPointPos = KernelMath.floor(moveByInUnits, dfeInt(16));
DFEVar upperPointPos = lowerPointPos + 1;
```

```
        DFEVar pointLower = stream.offset(inStream, lowerPointPos, -maxTraceSize, maxTraceSize);
        DFEVar pointUpper = stream.offset(inStream, upperPointPos, -maxTraceSize, maxTraceSize);
```

In this example `upperPointPos=lowerPointPos+1`, and part of the power of dynamic offsets is that the relationship can be entirely arbitrary, as long as it remains within the specified minimum and maximum bounds.

Note that dynamic offsets create a new stream and not just an offset of the original stream. To demonstrate this point we compare two ways of obtaining two points from the input stream. The first, used in our example in *Listing 28*, uses two dynamic offsets to obtain two points, at addresses that happen to be always one point apart:

```
        DFEVar lowerPointPos = KernelMath.floor(moveByInUnits, dfeInt(16));
        DFEVar upperPointPos = lowerPointPos + 1;
```

```
        DFEVar pointLower = stream.offset(inStream, lowerPointPos, -maxTraceSize, maxTraceSize);
        DFEVar pointUpper = stream.offset(inStream, upperPointPos, -maxTraceSize, maxTraceSize);
```

An alternative, but incorrect, approach might appear to be:

```
DFEVar pointLower = stream.offset(inStream, lowerPointPos, -maxTraceSize, maxTraceSize);
DFEVar pointUpper = stream.offset(pointLower, 1);
```

Here we attempt to use a single dynamic offset and a static offset to achieve the same result, however the resulting data streams are *not* the same. A dynamic offset generates a *new* stream consisting of a perturbation of the input stream. In the code above, the static offset is requesting an offset into the new stream, not a further offset of 1 into the original stream.

There is no special run-time software code needed to use dynamic offsets, as the size of the offset is just a stream like any other on the DFE. Since dynamic offsets are generally used when offsets need to vary on a tick-by-tick basis, typically the offset value is either an input stream or computed on the dataflow engine. It is also possible to connect the offset size stream to a scalar input for less fine-grained control.

Generally it is good practice to minimize the use of dynamic offsets when static or variable offsets can be used instead, since dynamic offsets are much more costly to implement in a DFE.

8.5 Comparing different types of offset

Each type of offset has a different resource cost, with static offsets being the cheapest and dynamic offsets being the most expensive. This is shown in *Table 3*.

Table 3: Resource usage for three implementations of the same 9-point averaging filter with different kinds of stream offset

	LUTs	FFs	BRAM
Static offsets	11,076	13,749	28
Variable offsets	11,172	13,946	29
Dynamic offsets	12,341	14,662	71

For example, consider the 2D 9-point averaging filter we saw in the previous section. *Table 3* shows the device resources required to implement this filter using the three different kinds of stream offsets. In terms of LUTs and FFs, the three kinds of offsets are broadly similar, static offsets being the most efficient and dynamic offsets the least efficient, however dynamic offsets use many more BRAMs than the static or variable offsets.

The increased on-chip memory usage occurs because dynamic offsets are completely arbitrary in size at run time, so the compiler cannot optimize them. With static and variable offsets, requesting `stream.offset(x, 1000)` and `stream.offset(x, 2000)` requires approximately 2000 elements of storage, because the 1000-element offset can be provided as a “tap” from the 2000-element offset. However, with dynamic offsets, the 1000 and 2000 values are not known at compile-time, so the compiler is forced to allocate 3000 elements of storage.

Table 4 summarizes the different characteristics of the three types of stream offset.

Table 4: Characteristics of the different types of stream offset

	Static Offsets	Variable Offsets	Dynamic Offsets
Size configurable	At compile-time	Before a stream	tick-by-tick
On-chip resource cost	Low	Moderate	High
Compiler optimizes	Yes	Yes	No

8.6 Stream hold

A *stream hold* can be used to conditionally either output the current value of a stream or output a previous value. A stream hold is created using `Reductions.streamHold`:

DFEVar Reductions.streamHold(DFEVar input, DFEVar store)
DFEVar Reductions.streamHold(DFEVar input, DFEVar store, ***Bits*** *reset_val)*

The `store` stream is a Boolean stream that determines whether the current or stored value should be output. The store behavior can be summarized as:

- When `store` is 1, the stream hold stores the current value from the input stream.
- When `store` is 0, the stream hold ignores the current value from the input stream.

And the output behavior as:

- When `store` is 1, the stream hold outputs the current value from the input stream.
- When `store` is 0, the stream hold outputs the currently stored value.

The output of a stream hold is 0 when the Kernel is reset, or `reset_val` in the case of the second version of the method.

Listing 27: A 27-point averaging filter using variable stream offsets (ThreeDAverageVariableKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 8        Example: 4       Name: Three-dimensional average variable
 * MaxFile name: ThreeDAverageVariable
 * Summary:
 *         Kernel that averages across three dimensions within an 26-point window.
 */
package threedaveragevariable;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class ThreeDAverageVariableKernel extends Kernel {

    ThreeDAverageVariableKernel(KernelParameters parameters, int nxMax) {
        super(parameters);

        // Input
        DFEVar inStream = io.input("inStream", dfeFloat(8, 24));

        OffsetExpr nx = stream.makeOffsetParam("nx", 3, nxMax);
        OffsetExpr nxy = stream.makeOffsetParam("nxy", 3 * nx, nxMax * nx);

        // Extract 8 point window around current point
        DFEVar window[] = new DFEVar[27];
        int i = 0;
        for (int x=-1; x<=1; x++)
            for (int y=-1; y<=1; y++)
                for (int z=-1; z<=1; z++)
                    window[i++] = stream.offset(inStream, z*nxy+y*nx+x);

        // Sum points in window and divide by 27 to average
        DFEVar sum = constant.var(dfeFloat(8, 24), 0);
        for (DFEVar dfeVar : window) {
            sum = sum + dfeVar;
        }

        DFEVar result = sum / window.length;

        //Output
        io.output("outStream", result, dfeFloat(8, 24));
    }

}
```

Listing 28: Part of a simple Normal Move-Out application using dynamic offsets (NormalMoveOutKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 8      Example: 6      Name: Normal move-out
 * MaxFile name: NormalMoveOut
 * Summary:
 *    A kernel for a simple Normal Move-Out application using dynamic offsets
 */
package normalmoveout;

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

class NormalMoveOutKernel extends Kernel {

	NormalMoveOutKernel(KernelParameters parameters, int maxTraceSize) {
		super(parameters);

		// Inputs
		DFEVar inStream = io.input("inStream", dfeFloat(8,24));
		DFEVar moveByInTime = io.input("moveByInTime", dfeFloat(8, 24));
		DFEVar timeUnit = io.scalarInput("timeUnit", dfeFloat(8,24));

		// Calculate position of two points to extract from input stream
		DFEVar moveByInUnits = moveByInTime/timeUnit;

		// Convert to an integer, rounding down using a 'floor' function
		// which also converts the type from floating-point to a 16-bit integer

		DFEVar lowerPointPos = KernelMath.floor(moveByInUnits, dfeInt(16));
		DFEVar upperPointPos = lowerPointPos + 1;
		DFEVar interp = moveByInUnits - lowerPointPos.cast(dfeFloat(8,24));

		// Extract points from input stream
		DFEVar pointLower = stream.offset(inStream, lowerPointPos, -maxTraceSize, maxTraceSize);
		DFEVar pointUpper = stream.offset(inStream, upperPointPos, -maxTraceSize, maxTraceSize);

		// Interpolate between points to create output
		DFEVar result = interp * pointLower + (1-interp) * pointUpper;

		// Output
		io.output("outStream", result, dfeFloat(8, 24));
	}
}
```

Listing 29: Kernel demonstrating us of a stream hold (StreamHoldKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial)
 * Chapter: 8     Example: 7     Name: Stream Hold
 * MaxFile name: StreamHold
 * Summary:
 *     Kernel that uses a stream hold to keep the maximum
 *     value from a stream.
 */

package streamhold;

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

class StreamHoldKernel extends Kernel {

    StreamHoldKernel(KernelParameters parameters, int counterWidth) {
        super(parameters);

        // Input
        DFEVar inStream = io.input(”inStream”, dfeUInt(32));
        DFEVar holdCount = io.scalarInput(”holdCount”, dfeUInt(counterWidth));

        // Offsets and Calculation
        DFEVar count = control.count.simpleCounter(counterWidth);
        DFEVar result = Reductions.streamHold(inStream, count < holdCount);

        // Output
        io.output(”outStream”, result, dfeUInt(32));
    }
}
```

The following table shows example input and output over 11 Kernel ticks:

input	0	1	2	3	4	5	4	3	2	1	4
store	0	1	1	1	0	1	0	0	1	0	0
output	0	1	2	3	3	5	5	5	2	2	2

8.6.1 Stream hold example

Example 7 demonstrates simple use of a stream hold. The Kernel is shown in *Listing 29*.

The inputs to the Kernel are a stream of unsigned integer values and a scalar input:

```
        DFEVar inStream = io.input(”inStream”, dfeUInt(32));
        DFEVar holdCount = io.scalarInput(”holdCount”, dfeUInt(counterWidth));
```

A counter, continually loops around its range, and is compared with the scalar input to control the stream hold:

```
        DFEVar count = control.count.simpleCounter(counterWidth);
        DFEVar result = Reductions.streamHold(inStream, count < holdCount);
```

Exercises

Exercise 1: Static offsets

Create a Kernel design that uses stream offsets to implement the equivalent of the following code:

```
float inStream[simlen];
float outStream[simlen];
...
for ( int i = 3; i < simlen-3; i++) {
    outStream[i] = (inStream[i-3]-inStream[i+3])*(1.0/16)
        + (inStream[i-2]-inStream[i+2])*(1.0/8)
        + (inStream[i-1]-inStream[i+1])*(1.0/4)
        + inStream[i]*(1.0/2) ;
}
```

Tests for your implementation on DFEs and in simulation are provided.

Exercise 2: Variable offsets

Take the convolution Kernel you developed in Example 1 (or use the solution to Example 1 provided) and, assuming that the input stream is a two-dimensional array of size $n \times n$, transpose the operation so that it is executed in the y dimension. This can be expressed in C as follows:

```
float inStream[n*n];
float outStream[n*n];
...
for (int y = 3; y < n-3; y++) {
    for (int x = 3; x < n-3; x++) {
        outStream[y*n+x] = (inStream[(y-3)*n+x]-inStream[(y+3)*n+x])*(1.0/16)
                + (inStream[(y-2)*n+x]-inStream[(y+2)*n+x])*(1.0/8)
                + (inStream[(y-1)*n+x]-inStream[(y+1)*n+x])*(1.0/4)
                + inStream[y*n+x]*(1.0/2);
    }
}
```

Use a stream offset expression to allow n to be varied between 5 and 1024 without recompilation. You can ignore boundary cases. Make the four coefficient values scalar inputs, so that they can also be varied without recompilation.

Remember to edit the CPU code to set your runtime parameters and scalar inputs.

9

Control Flow in Dataflow Computing

The wheels on the bus go round and round.

– unknown

In this section we introduce counters, which are the dataflow equivalents of loops in sequential programs. Counters allow Kernel designs to keep track of where they are in the stream and keep track of various levels of streaming and iteration.

9.1 Simple counters

A simple counter is instantiated using the method `simpleCounter` from `control.count`, which takes the bit width for the counter as an argument:

*DFEVar **control.count**.simpleCounter(**int** bit_width)*

The counter generates a stream of values of unsigned integer type of the specified bit width, starting with the initial value 0 for the first incoming stream element and incrementing the value by one for each subsequent stream element. Upon reaching a value of $2^w - 1$ (where w is the bit width of the counter), the counter wraps around and starts again at 0.

There is also a second version of the `simpleCounter` method which takes the maximum value as its second parameter. This version of a counter wraps when it hits *one less than* this value.

*DFEVar **control.count**.simpleCounter(**int** bit_width, DFEVar wrap_point)*

Listing 30 shows a Kernel program using a simple counter to add a count to an incoming stream.

We create a simple counter to count from 0 through to the maximum value that can be held in an unsigned integer variable of width `width`:

```
        DFEVar count = control.count.simpleCounter(width);
```

Listing 30: Program for the simple counter Kernel (SimpleCounterKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 9      Example: 1      Name: Simple Counter
 * MaxFile name: SimpleCounter
 * Summary:
 *         Kernel that shows how to create a simple counter and add
 *    its count to the input stream.
 */

package simplecounter;

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

class SimpleCounterKernel extends Kernel {

    SimpleCounterKernel(KernelParameters parameters, int width) {
        super(parameters);

        // Input
        DFEVar x = io.input("x", dfeUInt(32));

        // Create a simple counter and add its count to the input
        DFEVar count = control.count.simpleCounter(width);

        DFEVar result = x + count;

        // Output
        io.output("y", result, dfeUInt(width));
    }

}
```

Figure 35 displays the corresponding Kernel graph. Visually, counters are represented by hexagons (⬡) in Kernel graphs.

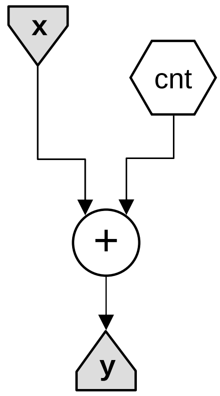


Figure 35: Graph for the simple counter Kernel, as constructed by the program in *Listing 30*

9.2 Nested loops

A common idiom in conventional programming languages is nested loops. For example, if we were working with a two-dimensional array, we might want to generate indices for each dimension. Consider the following Java code:

```
for (int i = 0; i < 6; i += 2) {
    for (int j = 0; j < 2; ++j) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

which generates the following output:

```
i = 0, j = 0
i = 0, j = 1
i = 2, j = 0
i = 2, j = 1
i = 4, j = 0
i = 4, j = 1
```

In the dataflow programming model, we use **chains of counters** to implement nested loops. A chained counter is created by calling the `control.count.makeCounterChain` method which returns a `CounterChain` object:

```
CounterChain control.count.makeCounterChain()
```

Calling the `addCounter(max, inc)` method on this new object creates a counter variable which produces output as if it were within the following `for` loop:

```
for (int n = 0; n < max; n += inc)
```

In *Listing 31*, we create a pair of counters `i` and `j` which count in the same way as the nested `for` loops above:

```
        CounterChain chain = control.count.makeCounterChain();
        DFEVar i = chain.addCounter(maxI, 2);
        DFEVar j = chain.addCounter(maxJ, 1);
```

Listing 31: A 2D counter Kernel using a counter chain (Simple2DCounterKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 9        Example: 2      Name: Simple two-dimensional counter
 * MaxFile Name: Simple2DCounter
 * Summary:
 *         Kernel that constructs a chained counter and outputs its
 *    values every cycle.
 */

package simple2dcounter;

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

class Simple2DCounterKernel extends Kernel {

    Simple2DCounterKernel(KernelParameters parameters, int maxI, int maxJ) {
        super(parameters);

        // Inputs
        DFEVar passThrough = io.input("input", dfeUInt(32));

        // Create Counters
        CounterChain chain = control.count.makeCounterChain();
        DFEVar i = chain.addCounter(maxI, 2);
        DFEVar j = chain.addCounter(maxJ, 1);

        i = i.cast(dfeUInt(32));
        j = j.cast(dfeUInt(32));

        // Outputs
        io.output("i", i, i.getType());
        io.output("j", j, j.getType());
        io.output("output", passThrough, passThrough.getType());
    }
}
```

9.3 Advanced counters

As a general control mechanism, a `simpleCounter` or `CounterChain` may not be flexible enough. It is possible to create more complex counting behavior by specifying several characteristics of the counter:

bit width Counters generate a `DFEUInt` with the specified bit width. The bit width may be set to any non-zero unsigned integer.

initial value By default a counter's initial value is 0. The `initial value` parameter sets an arbitrary starting point for counting.

increment The increment defaults to 1, and can be set to an arbitrary value via the `increment` parameter. The `increment` parameter is combined with the count based on the 'count mode' below.

count mode There are three modes of counting:

- `NUMERIC_INCREMENTING` — add an increment to the count on each enabled Kernel tick
- `SHIFT_LEFT` — logically shift left the count on each enabled Kernel tick

- `SHIFT_RIGHT` — logically shift right the count on each enabled Kernel tick

The remainder of this document refers only to `NUMERIC_INCREMENTING` counters, which is also the default counting mode.

maximum value By default, the maximum value of a counter is $2^w - 1$, where w is the bit width of the counter. The maximum may also be set explicitly, in which case it must be an integer greater than 0 which can be represented in w bits.

Additionally, it is possible to set the maximum value of the counter dynamically from another stream.

wrap mode What happens to a counter when it reaches its maximum is specified by the counter's wrap mode. There are three wrap modes:

- `COUNT_LT_MAX_THEN_WRAP` — The counter counts up to and including `max-1`, then restarts counting from 0. For example, if the maximum is 5 and the increment is 1, the counter's values are:

$$0, 1, 2, 3, 4, 0, 1, 2, \ldots$$

For a maximum value of 5 and an increment of 2 the values are:

$$0, 2, 4, 0, 2, 4, \ldots$$

- `STOP_AT_MAX` — The counter stops at the greatest multiple of the increment not exceeding maximum value until the Kernel is reset. This value is less than the maximum if the maximum is not a multiple of the increment. For example, if the maximum is 5 and the increment is 2 then the values are:

$$0, 2, 4, 4, 4, 4, \ldots$$

- `MODULO_MAX_OF_COUNT` — The counter's value is calculated *modulo* the maximum value. For example, if the maximum is 5 and the increment is 2 then the count is:

$$0, 2, 4, 1, 3, 0, 2, 4, \ldots$$

The default wrap mode is `COUNT_LT_MAX_THEN_WRAP`.

Whether or not a counter has wrapped can be found by getting the counter's **wrap signal**. This is a Boolean state variable which is 1 during the last tick before wrapping. The wrap signal is often used when combining several counters together and is also useful as an input for watch nodes.

wrap value When a counter wraps, its next value is specified by the wrap value. By default this is 0.

enable The enable signal for a counter is a Boolean (one bit wide) `DFEVar`. For every tick where the enable is equal to 1, the counter counts and otherwise stays at the current value.

9.3.1 Creating an advanced counter

The parameters that specify an advanced counter's behavior are encapsulated in a `Count.Params` object which is returned by the `count.makeParams` method:

*Count.Params **control.count**.makeParams(**int** bit_width)*

The counter's parameters can then be customized by calling the various `with` methods that are defined on `Count.Params` (e.g. `withInc`, `withMax`, etc.). These methods return a *new* `Count.Params` object that can again be customized by subsequent calls to `with` methods.

> Note that `Count.Params` objects are immutable, so calling a `with` method does not modify the existing object.

Once we have a suitable `Count.Params` object we can create a counter with the specified behavior by calling the `count.makeCounter` method, using the `Count.Params` object as the only argument. This returns a `Counter` object:

*Counter **control.count**.makeCounter(Count.Params params)*

Once we have a `Counter` object we can get the counter's value from the `getCount` method and the 'wrap' signal by calling `getWrap`.

DFEVar getCount()
DFEVar getWrap()

Listing 32 shows a Kernel design that creates an advanced counter. The corresponding Kernel graph is shown in *Figure 36*.

We first make a `Count.Params` object with the specified bit width, maximum and increment:

```
        Count.Params paramsOne = control.count.makeParams(width)
            .withMax(countOneMax)
            .withInc(countOneInc);
```

We use this `Count.Params` object to create our actual counter:

```
        Counter counterOne = control.count.makeCounter(paramsOne);
```

We then create a second counter with the wrap signal of the first counter as its enable signal:

```
        Count.Params paramsTwo = control.count.makeParams(width)
            .withEnable(counterOne.getWrap())
            .withMax(countTwoMax)
            .withWrapMode(WrapMode.STOP_AT_MAX);

        Counter counterTwo = control.count.makeCounter(paramsTwo);
```

If we set `count1Max` to 3 and `count2Max` to 2, the two counters count as follows:

```
ct1 = 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, ...
ct2 = 0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 2, 2, ...
```

Listing 32: A complex counter arrangement with a stopping counter (ComplexCounterKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf))>
 * Chapter: 9       Example: 3       Name: Complex Counter
 * MaxFile name: ComplexCounter
 * Summary:
 *        Kernel design that  creates advanced counters specifying
 *    maximum value, increment, wrap mode and enable streams.
 */

package complexcounter;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Counter;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.WrapMode;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class ComplexCounterKernel extends Kernel {

    private static  final  int  width = 32;

    ComplexCounterKernel(KernelParameters parameters,
        int  countOneMax, int countOneInc, int countTwoMax) {
        super(parameters);

        // Input
        DFEVar streamIn = io.input("input",  dfeUInt(width)) ;

        // Counters and calculation
        Count.Params paramsOne = control.count.makeParams(width)
            .withMax(countOneMax)
            .withInc(countOneInc);

        Counter counterOne = control.count.makeCounter(paramsOne);

        Count.Params paramsTwo = control.count.makeParams(width)
            .withEnable(counterOne.getWrap())
            .withMax(countTwoMax)
            .withWrapMode(WrapMode.STOP_AT_MAX);

        Counter counterTwo = control.count.makeCounter(paramsTwo);

        DFEVar countTwo = counterTwo.getCount();
        DFEVar countOne = counterOne.getCount();

        DFEVar result = streamIn + countTwo;

        // Output
        io.output("result",  result,  dfeUInt(width)) ;
        io.output("countOne", countOne, countOne.getType());
        io.output("countTwo", countTwo, countTwo.getType());
    }
}
```

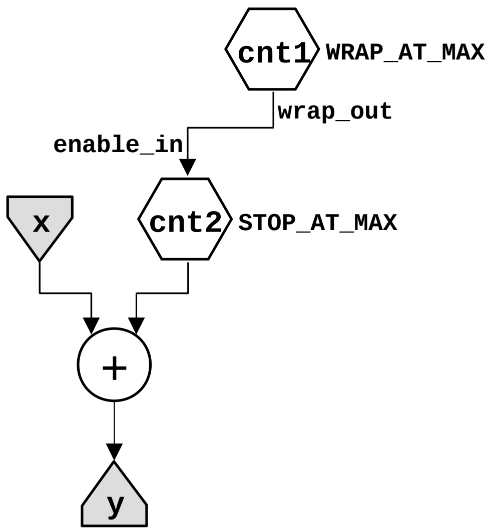


Figure 36: Advanced counter example Kernel graph

Exercises

Exercise 1: Simple counter

Create a Kernel design that instantiates a simple counter. Derive from it a count that goes from 15 down to zero, before wrapping from 15. Add this result to input stream `x` to produce output stream `y`. The CPU code is provided to test your implementation.

> Hint: you can use an algebraic expression of the output of a counter that goes from `0` up to `15`.

Exercise 2: 2D counter

Make a 2D counter with the maximum values taken from scalar inputs. Set the scalar input for the fast dimension to 4 and for the slow dimension to 3. Test that the output of both counters is as expected. MaxCompiler requires that all Kernels have at least one input and one output: as the core has no input, add a dummy input that is connected directly to a dummy output. The number of dummy data values sent to this core determines the number of values. Send in enough dummy data such that the counter in the slow dimension wraps.

Exercise 3: Advanced counter

This exercise revisits the 2D averaging filter with variable offsets from *section 8*. Adapt this example using a 2D counter to keep track of the edges of each 2D array as it is streamed through the core. Apply boundary conditions to the edges, such that points that lie outside the 2D array do not contribute to the average. The average should be calculated based on the number of valid points, not simply divided by 9 as before. CPU code is provided to stream in three consecutive 9x9 input arrays and verify that the output is as expected.

10

Advanced SLiC Interface

In *section 2*, we covered the Basic Static SLiC interface; in this section, we look at the Advanced Static and Advanced Dynamic SLiC interfaces. We then look in more detail at SLiC engine interfaces, looking this time at how they are defined in the MaxCompiler source. We also see how to run dataflow engines asynchronously using non-blocking functions, and cover the details of error handling, event monitoring and SLiC configuration. Finally we show how to use sliccompile to expose SLiC interfaces to scripting languages.

To recap, the SLiC functions are split into three levels of increasing complexity and flexibility:

Basic Static allows a single function call to run the design on a single DFE using only static actions defined via a given function call interface.

Advanced Static allows control of loading of DFEs, setting multiple complex actions, and optimization of CPU and DFE collaboration.

Advanced Dynamic allows for the full scope of dataflow optimizations and fine-grain control of allocation and de-allocation of all dataflow resources.

10.1 The lifetime of a `.max` file

The life-cycle of a `.max` file within a CPU application is as follows:

initialize - the `.max` file is initialized and functions become available.

load - the `.max` file is loaded onto a DFE. The DFE is now exclusively owned by the calling CPU process.

> Loading the `.max` file takes in the order of 100ms to 1s.

execute actions - the CPU calls SLiC functions to execute actions on the DFE.

> A loaded `.max` file has to be utilized for long enough to justify having waited up to a second to load the configuration.

unload - the DFE is released by the CPU process and returns to the pool of DFEs managed by MaxelerOS.

free - the `.max` file is deallocated.

The Basic Static SLiC interface loads the `.max` file onto the DFE when the first SLiC function is called, and releases the DFE when the CPU process terminates. The Advanced Static SLiC interface allows you to control exactly when the DFE is loaded and unloaded.

10.2 Advanced Static

With the Advanced Static SLiC interface, the `.max` file must first be initialized using a method specific to the `.max` file. For our moving average example, this is:

```
max_file_t* MovingAverage_init();
```

The `.max` file then gets loaded onto a DFE using `max_load`:

max_engine_t max_load(max_file_t *max_file, **const char** *engine_id_pattern);*

`engine_id_pattern` is a string that indicates which engines to use. This takes the form

```
hostname[:engine_id],
```

with `engine_id` either a number identifying a specific engine, or `*` for any engine. `hostname` can be one of:

- The host name of an MPC-X node for remote engines
- `"local"` - for using local engines
- `"*"` - the host name is taken from the configuration variable `"default_engine_resource"`

The return value from `max_load` is a handle for a DFE on which actions can be executed.

> The pattern format for arrays and groups of DFEs is different (see *subsection 10.4* and *subsection 10.5*).

10.2.1 Executing actions on DFEs

Actions are executed on a DFE using a structure containing the parameters for the action. This structure is specific to the `.max` file and engine interface. For example, for our moving average example, the default structure is:

```
typedef struct {
    int param_N;
    const float *instream_x;
    float *outstream_y;
} MovingAverage_actions_t;
```

The populated action structure can be executed on the loaded DFE using the `MovingAverage_run` function:

```
void MovingAverage_run(
    max_engine_t *engine,
    MovingAverage_actions_t *interface_actions);
```

This function returns when the action is complete and the output data is available to the CPU code. Finally, once it is no longer needed, the DFE can be unloaded:

```
void max_unload(max_engine_t *engine);
```

10.2.2 Holding the state of the DFE

The DFE can be loaded and unloaded multiple times during the execution of the CPU application to release the DFE for use by other applications or to load an alternative `.max` file.

The state of the DFE is maintained between load and unloading. Once the DFE has been unloaded, there is no guarantee that the state, including the contents of the LMem, is maintained.

Let us examine an Advanced Static example. The CPU code below shows the moving average CPU code required to run the `.max` file using the Advanced Static SLiC interface:

```
max_file_t *mavMaxFile = MovingAverage_init();
max_engine_t *mavDFE = max_load(mavMaxFile, "local:*");

MovingAverage_actions_t actions;
actions.param_N = size;
actions.instream_x = dataIn;
actions.outstream_y = dataOut;

MovingAverage_run(mavDFE, &actions);

max_unload(mavDFE);
```

10.3 Using multiple `.max` files

Using the Advanced Static SLiC interface, two `.max` files can be loaded onto multiple DFEs or loaded sequentially on one DFE. This is achieved by including the header files for each `.max` file and calling the appropriate functions for each file.

For example, imagine that we have our moving average `.max` file and another `.max` file called `Threshold.max` that thresholds its input stream. Using the Advanced Static SLiC interface, we can run the moving average, then reload the DFE with the thresholding `.max` file. passing the output of the moving average as the input:

```
#include "MovingAverage.h"
#include "Threshold.h"
#include <MaxSLiCInterface.h>

...
max_file_t *mavMaxFile = MovingAverage_init();
max_engine_t *myDFE = max_load(mavMaxFile, "*");

MovingAverage_actions_t mavAction;
mavAction.param_N = size;
mavAction.instream_x = dataIn;
mavAction.outstream_y = mavOut;

MovingAverage_run(myDFE, &mavAction);

max_file_t *threshMaxFile = Threshold_init() ;
max_unload(myDFE);

myDFE = max_load(threshMaxFile, "*");
Threshold_actions_t threshAction;
threshAction.param_N = size;
threshAction.instream_x = mavOut;
threshAction.outstream_y = dataOut;

Threshold_run(myDFE, &threshAction);

max_unload(myDFE);
```

10.4 Running `.max` files on multiple DFEs

A `.max` file can be run on multiple *connected, adjacent* DFEs with one command, using an **array** of actions in the Advanced Static and Advanced Dynamic SLiC interfaces. There are SLiC functions specific to the `.max` file with the `_array` suffix for running such an array, for example:

```
void MovingAverage_run_array(max_engarray_t *engarray, MovingAverage_actions_t *interface_actions[]);
```

There are also load and unload functions for arrays:

```
max_engarray_t* max_load_array(max_file_t *maxfile, int number_of_engines, const char *engine_id_pattern);

void max_unload_array(max_engarray_t *engarray);
```

The `engine_id_pattern` argument for arrays is one of:

- The host name of an MPC-X node when using remote engines
- `"local"` - use local engines;
- `"*"` - the host name is taken from the configuration variable `"default_engine_resource"`.

The array of actions is specified as a C array of pointers to action structures, as shown in the following example, which runs our moving average `.max` file on two DFEs:

```
const int numEngines = 2;
MovingAverage_actions_t *actions[numEngines];

for (int i = 0; i < numEngines; i++) {
    actions[i] = malloc(sizeof(MovingAverage_actions_t));
    actions[i]->param_N = size;
    actions[i]->instream_x = dataIn[i];
    actions[i]->outstream_y = dataOut[i];
}

max_file_t *maxfile = MovingAverage_init();
max_engarray_t *engines = max_load_array(maxfile, numEngines, "*");
Maxring_run_array(engines, actions);

max_unload_array(engines);
max_file_free(maxfile);
```

The DFEs in an array may communicate with each other via MaxRing connections. MaxRing is covered in more detail in *section 13*.

10.5 Sharing DFEs

Engine `groups` are multiple DFEs loaded with the same `.max` file, shared between threads, processes and users. Engine groups can optionally gain or lose real engines over time: this is managed by MaxelerOS at runtime.

An engine group is created using `max_load_group`:

```
max_group_t* max_load_group(
    max_file_t          *max_file,
    max_sharing_mode_t  sharing_mode,
    const char          *group_id,
    int                 group_size);
```

`sharing_mode` specifies how to share the DFEs and must be one of:

- `MAXOS_EXCLUSIVE` indicates that no other process can use an engine that belongs to the group: this is also the behavior when not using groups.
- `MAXOS_SHARED` is used for fine-grained sharing between processes, where no `.max` file loading takes place except on creating the group.
- `MAXOS_SHARED_DYNAMIC` allows the system to re-size the group and load/unload engines without explicit instruction from the user.

Table 5 summarizes the sharing and group size behavior for each of the sharing modes.

`group_id` is of the form `grouptag [@ hostname]`, where `hostname` is one of:

- The host name of an MPC-X node when using remote engines
- `"local"` - use local engines
- `"*"` - the host name is taken from the configuration variable `"default_engine_resource"`

In cases where `@` is absent, the entire string is used as the group tag. `group_size` is the required number of DFEs in the group, or the initial number for a DYNAMIC group.

Sharing mode	Share engines between processes[1]	Group size changed by user or system
MAXOS_EXCLUSIVE	no	no
MAXOS_SHARED	yes[2]	no
MAXOS_SHARED_DYNAMIC	yes[2]	yes

[1] Engines can always be shared between threads in the same process.
[2] Only processes with the same group ID can share engines in a group.

Table 5: Group properties.

10.5.1 Running actions on a DFE in a group

For individual engines and arrays, once they are loaded they are also locked for exclusive use, whereas with groups an additional lock step is required. An available DFE from the group is locked for use by calling `max_lock_any`:

```
max_engine_t* max_lock_any(max_group_t *group);
```

This function returns as soon as an engine becomes available. Once an engine has been locked, actions are executed on it using the SLiC functions specific to the `.max` file.

The `max_unlock` function releases the DFE back to the group:

```
void max_unlock(max_engine_t *engine);
```

Finally, the group can be unloaded when it is no longer required using `max_unload_group`:

```
void max_unload_group(max_group_t *group);
```

We can take our moving average as an example again. This time, we load the `.max` file to a group and run it on a DFE from the group:

```
max_file_t  *mavMaxFile = MovingAverage_init();

max_group_t *mavGroup = max_load_group(mavMaxFile, MAXOS_EXCLUSIVE, "mavGroup@local:*", 2);

MovingAverage_actions_t actions;
actions.param_N = size;
actions.instream_x = dataIn;
actions.outstream_y = dataOut;

max_engine_t *mavDFE = max_lock_any(mavGroup);

MovingAverage_run(mavDFE, &actions);
max_unlock(mavDFE);

max_unload_group(mavGroup);
```

If you only need to execute a single action on an engine, it is possible to use a high-performance *atomic* execution directly on the engine group. This will queue and execute the actions on any engine in the group then return. This type of interface is particularly useful when large numbers of CPU threads are submitting many small jobs on many engines. To use, use the auto-generated function:

```
MovingAverage_run_group(max_group_t *group, MovingAverage_actions_t *interface_actions);
```

This replaces the uses of `max_lock_any`, `MovingAverage_run` and `max_unlock` in the previous example.

<table>
<tr><th>Property</th><th>On max_load</th><th>At other times</th></tr>
<tr><td>MAXOS_EXCLUSIVE</td><td>Always, unless there is already a free engine with the same .max file loaded</td><td>Never</td></tr>
<tr><td>MAXOS_SHARED</td><td rowspan="2">Always, unless there is already a free engine with the same .max file loaded, which may have been loaded by another process [1]</td><td>Never</td></tr>
<tr><td>MAXOS_SHARED_DYNAMIC</td><td>If the user software requests to resize the group, or the system resizes the group based on demand</td></tr>
</table>

[1] Other processes can lock the engine after it has been loaded in one process.

Table 6: Engine loading behavior for different group properties.

10.5.2 Engine loads

MaxelerOS manages the loading of the `.max` file onto a DFE when required. When the `.max` file is loaded depends on the sharing mode and is detailed in *Table 6*.

10.6 Advanced Dynamic

The Advanced Dynamic SLiC interface offers all of functionality of the Advanced Static, but using strings to specify the parameters to functions to use a `.max` file, rather than static functions and structures defined in the `.max` file. This allows Advanced Dynamic CPU code to be decoupled from a particular `.max` file.

`.max` file initialization, engine loading and engine loading are performed in the same way as the Advanced Static (see *subsection 10.2*).

Actions are defined using a `max_actions_t` structure, which is initialized using `max_actions_init`:

```
max_actions_t *max_actions_init( max_file_t  *max_file,  const char *interface);
```

The `max_file` argument is a pointer to the initialized `.max` file handle and `interface` is the engine interface to use. Using `NULL` as the `interface` argument specifies that no engine interface is to be used.

A function API is used to configure the actions `max_actions_t` structure.

10.6.1 Setting engine interface parameters

Engine interface parameters can be set using `max_set_param_uint64t` for integer values, or `max_set_param_double` for floating-point values:

```
void max_set_param_uint64t(max_actions_t* actions, const char * const name, uint64_t value);
void max_set_param_double(max_actions_t* actions, const char * const name, double value);
```

The `name` argument is the string name for the engine interface parameter as defined in the Manager.

Likewise, there are functions for setting each element for arrays of engine interface parameters, `max_set_param_array_uint64t` and `max_set_param_array_double`:

```
void max_set_param_array_uint64t(max_actions_t *actions, const char * const name, uint64_t value, int idx);
void max_set_param_array_double(max_actions_t *actions, const char * const name, double value, int idx);
```

Calling the functions to set engine interface parameters when the `max_actions_t` has been initialized without an engine interface (i.e. set to `NULL`) is invalid and raises an error:

```
SLiC Error #517 @ actions_interfaces_internal.c:39 - Interface parameter "N"
     cannot be set for engine interface "(null)"
SLiC Error #518 @ actions.c:77 - Error reported from function "
    max_set_param_uint64t".
```

10.6.2 Streaming data

Data is added to input and output streams for an action set using `max_queue_input` and `max_queue_output`:

```
void max_queue_input(max_actions_t *actions, const char *stream_name, const void *data, size_t bytes);
void max_queue_output(max_actions_t *actions, const char *stream_name, const void *data, size_t bytes);
```

The `bytes` argument is the number of bytes of input data or the size of the memory allocated for the output data.

One advantage of the `max_queue` interface is that it can be called multiple times as part of a single actions object to queue multiple data transfers back-to-back. This can be used to "gather" data from memory into the DFE, "scatter" results into CPU memory, or to stream the same input data multiple times (by passing the same input pointer). Passing the same data pointer multiple times to an output stream leads to only the last values written to the memory being available.

10.6.3 Freeing the action set

When an action set is no longer needed, the memory can be released using `max_actions_free`:

```
void max_actions_free(max_actions_t *actions);
```

Failing to free an action set when it is no longer needed can lead to memory leaks.

10.6.4 Advanced Dynamic example

The code below shows the moving average example again, this time using the Advanced Dynamic SLiC interface:

```
max_file_t *mavMaxFile = MovingAverageSimple_init();
max_engine_t *mavDFE = max_load(mavMaxFile, "local:*");

max_actions_t *actions = max_actions_init(mavMaxFile, "default");
max_set_param_uint64t(actions, "N", size);
max_queue_input(actions, "x", dataIn, sizeBytes);
max_queue_output(actions, "y", dataOut, sizeBytes);
```

```
max_run(mavDFE, actions);

max_actions_free(actions);

max_unload(mavDFE);
```

10.6.5 Setting and retrieving Kernel settings

The number of Kernel ticks for a Kernel to run for is set using `max_set_ticks` with the name of the Kernel:

```
void max_set_ticks(max_actions_t *actions, const char *kernel_name, uint64_t nb_ticks);
```

Stream offsets are set using `max_set_offset` with the name of the Kernel and the name of the offset:

```
void max_set_offset(max_actions_t*actions, const char*kernel_name, const char*offset_var_name, int v);
```

Stream distance measurements and autoloop sizes (see Acceleration Tutorial - Loops and Pipelining) can be retrieved from a Kernel:

```
int max_get_stream_distance(max_actions_t *actions, const char *kernel_name, const char *offset_var_name);
int max_get_offset_auto_loop_size(max_actions_t *actions, const char *kernel_name, const char *offset_var_name);
```

10.6.6 Setting and reading mapped memories

Elements of mapped memories on a Manager block or Kernel are set individually using `max_set_mem_uint64t` or `max_set_mem_double`, depending on the type contained in the memory:

```
void max_set_mem_uint64t(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, uint64_t v);
void max_set_mem_double(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, double v);
```

Validation checks whether all of the elements in a mapped memory have been set.

Likewise, pointers for reading back each mapped memory element are set using `max_get_mem_uint64t` or `max_get_mem_double`:

```
void max_get_mem_uint64t(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, uint64_t *v);
void max_get_mem_double(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, double *v);
```

10.6.7 Action validation

The Advanced Dynamic SLiC interface provides automatic checking of actions before they are run on a DFE, ensuring that all the required parameters have been set correctly.

If we were to make a mistake in the CPU code for our moving average example, perhaps forgetting to set the `size` engine interface parameter for the default engine interface, the validation of the actions when they are run on the engine would raise an error:

```
Tue 17:10: SLiC Error #517 @ actions_interfaces_internal.c:182 - Interface
    parameter "N" not defined for engine interface "default"
Tue 17:10: SLiC Error #518 @ maxfile_setup.c:486 - Error reported from
    function "max_actions_get_param_uint64t".
Tue 17:10: Aborted
```

There are functions in the API to tell SLiC to ignore specific parameters for a set of actions when performing this checking:

void *max_ignore_route(max_actions_t ∗actions,* ***const char*** *∗block_name);*
void *max_ignore_offset(max_actions_t ∗actions,* ***const char*** *∗kernel_name,* ***const char*** *∗offset_var_name);*
void *max_ignore_kernel(max_actions_t ∗actions,* ***const char*** *∗kernel_name);*
void *max_ignore_block(max_actions_t ∗actions,* ***const char*** *∗block_name);*

For a mapped memory, the whole memory can be ignored, or just the input or output:

void *max_ignore_mem(max_actions_t ∗actions,* ***const char*** *∗block_name,* ***const char*** *∗mem_name);*
void *max_ignore_mem_input(max_actions_t ∗actions,* ***const char*** *∗block_name,* ***const char*** *∗mem_name);*
void *max_ignore_mem_output(max_actions_t ∗actions,* ***const char*** *∗block_name,* ***const char*** *∗mem_name);*

Validation for a set of actions can be disabled altogether:

void *max_disable_validation(max_actions_t ∗actions);*

Disabling action validation makes it possible to execute an incomplete set of actions, which can lead to bugs that are hard to track down.

10.6.8 Groups and arrays of engines

Groups and arrays of engines behave in the same way as when using the Advanced Static level functions (see *subsection 10.4* and *subsection 10.5*).

In the case of arrays, there is an Advanced Dynamic function for running the array, `max_run_array`:

void *max_run_array(max_engarray_t ∗engarray, max_actarray_t ∗actarray);*

For groups of engines, when no state needs to be maintained between operations on a DFE, `max_run_group` locks, runs and unlocks the DFE via a single function call:

void *max_run_group(max_group_t ∗group, max_actions_t ∗actions);*

10.7 Engine interfaces

Engine interfaces encapsulate different behavior or stages of execution for a `.max` file, simplifying the API for the CPU programmer. You can control which parameters and I/O are available in each engine interface, with inputs and outputs for the `.max` file either ignored or set automatically based on inputs from the CPU code.

Engine interfaces can be used to control the APIs for all inputs and outputs to the `.max` file.

- Number of ticks for Kernels to run
- Streams between the CPU and the DFE
- Streams to or from the LMem (see *section 13*)
- Scalar inputs and outputs
- Mapped memory inputs and outputs

- Offset parameters
- Routing blocks (fanout, multiplexers and demultiplexers) in Custom Managers (see Manager Compiler Tutorial)
- Autoloop offset values (see Acceleration Tutorial - Loops and Pipelining)
- Kernel distance measurements (see Acceleration Tutorial - Loops and Pipelining)

In the default engine interface, an argument for the number of ticks is added to the SLiC interface, and all streams are assumed to contain a number of elements equal to that number of ticks. All scalar inputs and outputs, mapped memories, stream offset parameters and Manager routing blocks are automatically added to the SLiC Interface, unless the default engine interface is overridden (see *subsubsection 10.7.2*). The only exceptions are AutoLoop offset values and Kernel distance measurements, which must be added explicitly (see *subsubsection 10.8.3*).

10.7.1 Adding an engine interface to a Manager

A standard simple engine interface is added to a `.max` file when `createSLiCinterface` is called without any arguments. This type of interface works for many simple programs but may not be flexible enough for all of your dataflow programs.

For complicated managers and kernels, custom engine interfaces can be added by creating an instance of the `EngineInterface` class, configuring the settings for the engine interface and then calling `createSLiCinterface` with the engine interface as a parameter, for example:

```
...
Manager manager = new Manager(params);
...
EngineInterface myInterface = new EngineInterface("myInterface");
...
manager.createSLiCinterface(myInterface);
manager.build();
...
```

Instances of `EngineInterface` are declared with a string for the name and an optional string argument `doc` for text that appears in the comments in the `.max` file and SLiC header file:

```
public EngineInterface(String name, String doc)
public EngineInterface(String name)
```

10.7.2 The default engine interface

Engine interfaces are given names by the MaxJ programmer, resulting in a interface to the Maxfile named `maxfilename_interfacename`. If the engine interface is given the special name “default” (or the shortcut constructor `EngineInterface()` is used), then the resulting function gets the special name `maxfilename` only. This is particularly suitable for DFEs which have one primary interface but may have other supporting interfaces to be used to perform specific operations.

If you do not create a “default” engine interface yourself, an interface will automatically be created which exposes all parameters for the kernels and manager.

For some DFEs, a “default” engine interface is inappropriate, so you can completely suppress the default engine interface from the SLiC API using the `suppressDefaultInterface` method on the `Manager` class.

10.7.3 Ignoring unset parameters

It is common for a `.max` file not to require a particular parameter to be set or visible for a given engine interface. All parameters that are not explicitly set in a engine interface can be ignored using the `ignoreAll` method:

void *ignoreAll(Direction flag)*

The `flag` argument selects whether to ignore only inputs, outputs or both:

```
public static enum Direction {
    IN,
    OUT,
    IN_OUT;
};
```

`Direction.IN` refers to all *settings* that are sent from the CPU to the DFE, and includes:

- Number of ticks for Kernels to run
- Stream inputs from the CPU to the DFE
- Stream outputs from the DFE to the CPU
- Streams to or from the LMem
- Scalar inputs
- Mapped memory inputs
- Offset parameters
- Routing blocks (fanout, multiplexers and demultiplexers) in Custom Managers (see Manager Compiler Tutorial)

`Direction.OUT` refers to data that is returned from the `.max` file or DFE to the CPU application:

- Scalar outputs
- Mapped memory outputs
- Autoloop offset values (see Acceleration Tutorial - Loops and Pipelining)
- Kernel distance measurements (see Acceleration Tutorial - Loops and Pipelining)

10.7.4 Ignoring specific parameters

Individual parameters can be suppressed in the API using one of a set of `ignore` methods on the engine interface:

```
void ignoreLMem(String streamName)
void ignoreStream(String streamName)
void ignoreRoute(String routingBlock)

void ignoreOffset(String blockName, String offsetName)
void ignoreScalar(String blockName, String scalarName)
```

In the case of mapped memories, whether to ignore inputs, outputs or both must also be specified, again using the `Direction` enum:

```
void ignoreMem(String blockName, String memName, Direction flag)
```

Similarly, parameters can be *un*ignored:

```
void unignoreScalar(String blockName, String scalarName)
void unignoreMem(String blockName, String memName)
void unignoreDistanceMeasurement(String kernelName, String name)
void unignoreAutoLoopOffset(String kernelName, String name)
```

Unignoring parameters is useful when used in conjunction with a call to `ignoreAll` to simplify the code where there is a large number of parameters.

10.7.5 Ignoring an entire Kernel

All of the inputs and outputs for a particular Kernel can be ignored with a single method call:

```
void ignoreKernel(String kernelName)
```

10.8 Engine interface parameters

An **engine interface parameter** is a user-defined parameter that can be used to express relationships between kernel entities such as tick count and stream length. Simple arithmetic operations may be performed with engine interface parameters to make more complex relationships.

Engine interface parameters can be created using the `addParam` method, which has two variants, the second of which takes a string argument `doc` to add documentation for the end-user of the `.max` file.

```
InterfaceParam addParam(String name, CPUTypes type)
InterfaceParam addParam(String name, CPUTypes type, String doc)
```

An engine interface parameter becomes an input argument in the SLiC API with the specified CPU type. `type` can be any of the following values:

```
public enum CPUTypes {
UINT8,      // -> uint8_t
INT8,       // -> int8_t
UINT16,     // -> uint16_t
INT16,      // -> int16_t
INT,        // -> int ( int64_t )
INT32,      // -> uint32_t
UINT32,     // -> uint32_t
UINT64,     // -> uint64_t
INT64,      // -> int64_t
FLOAT,
DOUBLE,
VOID;
}
```

For example, consider adding an engine interface parameter for the size of the incoming data stream:

```
...
EngineInterface myInterface = new EngineInterface();
InterfaceParam inputDataSize = myInterface.addParam("inputDataSize", UINT32, "The size of the input data stream in words");
...
```

This appears as a 32-bit, unsigned integer argument in the Basic Static SLiC interface:

```
/**
 * \brief Simple static function for the engine interface 'default'.
 *
 * ...
 * \param [in] param_inputDataSize Interface Parameter "inputDataSize": The size of the input data stream in words
 * ...
 */
void MyMaxFile(
     ...
     int32_t param_inputDataSize,
     ...);
```

10.8.1 Kernel settings

Various Kernel configuration options can be set in a SLiC engine interface:

- Number of ticks for Kernels to run

```
void setTicks(String blockName, InterfaceParam p)
void setTicks(String blockName, long p)
```

- Stream settings

```
void setStream(String streamName, CPUTypes type, InterfaceParam p)
void setStream(String streamName, CPUTypes type, long p)
```

- Mapped memory inputs

```
void setMem(String blockName, String memoryName, int index, double value)
void setMem(String blockName, String memoryName, int index, long value)
void setMem(String blockName, String memoryName, int index, InterfaceParam value)
```

- Offset expressions

```
void setOffset(String blockName, String offsetName, InterfaceParam p)
void setOffset(String blockName, String offsetName, long p)
```

- Scalar inputs

```
void setScalar(String blockName, String scalarName, long value)
void setScalar(String blockName, String scalarName, double value)
void setScalar(String blockName, String scalarName, InterfaceParam p)
```

In each case, overloaded versions of the methods allow the values to be set either from a Java variable or from an engine interface parameter.

There are no explicit functions for retrieving the values of scalar outputs or mapped-memory outputs: these are automatically added by the engine interface, unless disabled using `ignoreScalar` or `ignoreMem`, as described in *subsubsection 10.7.4*.

10.8.2 LMem settings

LMem settings can be set either using all Java variables or all engine interface parameters:

__void__ setLMemLinear(__String__ streamName, __long__ address, __long__ size)
__void__ setLMemLinear(__String__ streamName, InterfaceParam address, InterfaceParam size) {

__void__ setLMemLinearWrapped(__String__ streamName, InterfaceParam address, InterfaceParam arrSize, InterfaceParam rwSize, InterfaceParam offset)
__void__ setLMemLinearWrapped(__String__ streamName, __long__ address, __long__ arrSize, __long__ rwSize, __long__ offset) {

__void__ setLMemStrided(__String__ streamName, __long__ address, __long__ sizeFast, __long__ sizeSlow, __long__ strideMode)
__void__ setLMemStrided(__String__ streamName, InterfaceParam address, InterfaceParam sizeFast, InterfaceParam sizeSlow, InterfaceParam strideMode)

__void__ setLMemBlocked(__String__ streamName, __long__ address,
__long__ arraySizeFast, __long__ arraySizeMed, __long__ arraySizeSlow,
__long__ rwSizeFast, __long__ rwSizeMed, __long__ rwSizeSlow,
__long__ offsetFast, __long__ offsetMed, __long__ offsetSlow)

__void__ setLMemBlocked(__String__ streamName, InterfaceParam address,
InterfaceParam arraySizeFast, InterfaceParam arraySizeMed, InterfaceParam arraySizeSlow,
InterfaceParam rwSizeFast, InterfaceParam rwSizeMed, InterfaceParam rwSizeSlow,
InterfaceParam offsetFast, InterfaceParam offsetMed, InterfaceParam offsetSlow)

__void__ setLMemInterruptOn(__String__ streamName)

Two methods are provided to create an engine interface parameter instance from a Java variable:

InterfaceParam addConstant(__double__ value)
InterfaceParam addConstant(__long__ value)

For details on LMem access patterns and interpretation of the settings, see *subsection 13.3*.

10.8.3 Autoloop offset parameters and distance measurements

Engine interface parameters to retrieve autoloop offset parameters and distance measurements (see the Loops and Pipelining Acceleration Tutorial) from a Kernel can be added to an engine interface:

InterfaceParam getAutoLoopOffset(__String__ kernelName, __String__ name)
InterfaceParam getDistanceMeasurement(__String__ kernelName, __String__ name)

These parameters are special cases that are added to the SLiC API as outputs to the CPU code, as well as behaving as standard `InterfaceParam` objects for the purposes of setting parameters on Kernels or Managers. For example, let us take adding a distance measurement in an engine interface:

```
EngineInterface myInterface = new EngineInterface();
InterfaceParam loopLength = myInterface.getDistanceMeasurement("myKernel", "loopLength");
```

This adds an argument `param_MyKernel_loopLength` to the SLiC interface, for example in the case of Basic Static:

```
void MyMaxFile(
    ...
    int32_t *param_MyKernel_loopLength,
    ...) ;
```

If the engine interface parameter for an autoloop offset parameter or distance measurement is only used in the Manager itself, for example in a calculation to set a Kernel or Manager parameter, it can be suppressed from the SLiC interface:

```
void ignoreAutoLoopOffset(String kernelName, String name)
void ignoreDistanceMeasurement(String kernelName, String name)
```

10.8.4 Engine interface parameter arrays

Engine interface parameter arrays allow an array of data to be passed from the CPU code to the `.max` file:

```
public InterfaceParamArray addParamArray( String name, CPUTypes type, String doc )
public InterfaceParamArray addParamArray( String name, CPUTypes type )
```

Within a Manager, the `InterfaceParam` elements of the array are accessed using the array access `[]` operator, using either a `InterfaceParam` or a Java integer as the index.

In many cases, the size of the engine interface parameter array can be inferred from the engine interface code, for example:

```
InterfaceParamArray coeff = myInterface.addParamArray("coeff", CPUTypes.FLOAT);
for ( int i = 0 ; i < 100 ; i++ ) {
    myInterface.setScalar("myKernel", " filter_ " + i, coeff[i]) ;
}
```

This may not be possible if the engine interface parameter array is indexed only using by an `InterfaceParam` instance, for example:

```
InterfaceParam idx = myInterface.addParam("idx", CPUTypes.UINT32);
InterfaceParamArray coeff = myInterface.addParamArray( "coeff", CPUTypes.FLOAT);
myInterface.setScalar("myKernel", "myScalar", coeff[idx] ) ;
```

In this case, the size of the engine interface parameter array must be set explicitly:

```
coeff.setMaxSize( 100 );
```

10.9 `.max` file constants

In many designs, a `.max` file is built with a number of compile-time parameters that define, for example, behavior, limits or dimensions. These can be passed to the CPU code as constants in the `.max` file.

Three methods on the `Manager` class are provided for defining integer, floating-point and string constants that appear as `#define`s in the SLiC header file:

```
void addMaxFileConstant(String name, int value)
void addMaxFileStringConstant(String name, String value)
void addMaxFileDoubleConstant(String name, double value)
```

The `name` argument is the string that is appended to the name of the `.max` file to give the name of the constant in the SLiC header file, for example, for a `.max` file with the name `MyMaxfile` and a floating-point constant `myconstant` with the value `0.5`, the resultant `#define` appears as:

```
#define MyMaxfile_myconstant (0.5)
```

It is also possible to retrieve maxfile constants using dynamic functions, named `max_get_constant_string`, `max_get_constant_double` and `max_get_constant_uint64t`, depending on the type of the constant.

10.10 Asynchronous execution

In many applications, it is more efficient for the CPU application to continue executing code while the DFE is running a set of actions. SLiC provides non-blocking versions of the functions we have seen so far for running actions on engines, groups and arrays. These functions return immediately once the actions have been committed, allowing the CPU application to continue execution.

Initialization, loading and unloading of `.max` files onto DFEs is performed in the same way as for the blocking API functions. When it comes to running individual DFEs, arrays or groups of DFEs, there are `_nonblock` versions of the Static SLiC Interface functions in the `.max` file header file. For example, our moving average has the following non-blocking functions:

```
max_run_t *MovingAverage_nonblock(int32_t param_N, const float *instream_x, float *outstream_y);
max_run_t *MovingAverage_run_nonblock(max_engine_t *engine, MovingAverage_actions_t *interface_actions);
max_run_t *MovingAverage_run_group_nonblock(max_group_t *group, MovingAverage_actions_t *interface_actions);
max_run_t *MovingAverage_run_array_nonblock(max_engarray_t *engarray, MovingAverage_actions_t *interface_actions[]);
```

Likewise, there are Advanced Dynamic non-blocking functions:

```
max_run_t* max_run_nonblock(max_engine_t *engine, max_actions_t *actions);
max_run_t* max_run_array_nonblock(max_engarray_t *engarray, max_actarray_t *actarray);
max_run_t* max_run_group_nonblock(max_group_t *group, max_actions_t *actions);
```

All of these non-blocking functions return a handle to the execution status (a pointer to a `max_run_t` structure), or `NULL` in the case of an error. To re-synchronize with the DFE, there is a function to wait for the actions to complete for an execution handle:

```
void max_wait(max_run_t *run);
```

In the case of arrays of DFEs, this waits for the set of actions to be completed on all the DFEs in the array.

Calling a non-blocking run function on the same DFE(s) queues up action sets: this helps minimize any idle time between actions on a DFE, especially when running remote DFEs on an MPC-X node.

To indicate to SLiC that the outcome of a set of actions being executed is to be ignored in the CPU code, `max_nowait` must be called:

```
void max_nowait(max_run_t *run);
```

This can be useful when queuing a number of action sets on a DFE (or array of DFEs), so that the CPU application only need wait for the last one to be completed.

> Either `max_wait` or `max_nowait` must be called for every execution status handle to ensure that SLiC can release all the associated memory.

> ✷ `max_nowait` cannot be run on an execution status handle for actions running on a group of DFEs, as actions may be executed out of sequence. Calling `max_nowait` with such a handle raises an error.

10.10.1 Asynchronous execution example

The moving average from *section 3* can be modified to run the CPU version of the moving average at the same time as the moving average is being executed on the DFE:

```
max_run_t *execStatus = MovingAverage_nonblock(size, dataIn, dataOut);

MovingAverageCPU(size, dataIn, expected);

/* Other CPU work can be done here. */

max_wait(execStatus);
```

And using the Advanced Static API:

```
max_file_t *mavMaxFile = MovingAverage_init();
max_engine_t *mavDFE = max_load(mavMaxFile, "local:*");

MovingAverage_actions_t actions;
actions.param_N = size;
actions.instream_x = dataIn;
actions.outstream_y = dataOut;

max_run_t *execStatus = MovingAverage_run_nonblock(mavDFE, &actions);

MovingAverageCPU(size, dataIn, expected);

/* Other CPU work can be done here. */

max_wait(execStatus);

max_unload(mavDFE);
```

10.11 Error handling

Errors are reported into *error contexts*, which are instances of the `max_errors_t` structure. There are a number of handles in SLiC that contain an error context:

`max_file_t` - `.max` file handles

`max_actions_t` - action sets

`max_engine_t` - engine handles

`max_engarray_t` - engine array handles

`max_actarray_t` - action set array handles

`max_group_t` - engine group handles

`max_run_t` - execution status handle (see *subsection 10.10*)

The error context is called `errors` in every case and thus can be accessed as, for example `mymaxfile->errors`.

By default, SLiC is configured to abort execution of the CPU program on an error. This can be disabled using `max_errors_mode`:

```
void max_errors_mode(max_errors_t *errors, int abort_on_error);
```

A value for `abort_on_error` of `0` instructs the application not to abort for the specified error context, `1` instructs it to abort.

Setting `max_errors_mode` on an error context for a handle sets the error contexts for all the handles created from it to have the same behavior. For example, setting the error context for a `.max` file to not abort on an error causes all arrays, groups, engines, actions and action arrays created from the `.max` file handle to also not abort on an error. Where a function takes multiple handles as arguments, the abort behavior of the returned handle is inherited from the engine, array or group handle. Individual error contexts that have inherited abort behavior can still be changed if required.

`max_ok` is used to check an error context on a handle once a function has returned:

```
int max_ok(max_errors_t *errors);
```

This returns `1` if there are no errors or `0` if there are errors. Alternatively specific errors can be checked using `max_errors_check`:

```
int max_errors_check(max_errors_t * errors, int error_code);
```

This takes an integer argument for the error code and returns `1` if the error has been raised or `0` if not.

> For error contexts that are not set to abort on error, the error context of *every handle* passed as an argument to a function must be checked once that function has returned.

> For error contexts that are not set to abort on error, handles returned from SLiC functions must *always* be tested for NULL.

A text trace of the error can be retrieved by passing the error context to `max_errors_trace`:

```
char* max_errors_trace(max_errors_t * errors);
```

The returned string is allocated by the function and must be deallocated as appropriate by your CPU application.

Option string	Type	Default	Description
default_pcie_timeout	integer	30	Default timeout for PCIE stream transfer (seconds)
default_wfi_timeout	integer	30	Default timeout for interrupt wait (seconds)
default_topology_timeout	integer	-1 (infinite)	Default timeout for topology allocation (seconds)
default_maxdebug_mode	enum	never	Default debug mode, see *Figure 5.3.5*
verbose	Boolean	false	Enable full debug output
eventlog_ignore_errors	Boolean	false	Ignore errors in the event logging module
eventlog_enable	Boolean	false	Enable event monitoring, even if it is not enabled in the CPU code
dfeprintf_enable	Boolean	true	Enable dfePrintf output
find_next_debug_dir	Boolean	true	Change the debug directory name if the current one already exists
printf_to_stdout	Boolean	true	Stream Debug.printf to standard output
default_engine_resource	string	NULL	Default location of the engines
use_simulation	string	NULL	Simulation server socket
default_eventlog_server	string	NULL	Name of event logging server
default_eventlog_process_name	string	NULL	Event logging process name
debug_dir	string	debug	Directory where debug output is written

Table 7: SLiC configuration options.

10.12 SLiC configuration

In addition to the configuration available through the C interface, aspects of SLiC can be configured through both configuration files and environment settings. The inputs are parsed in this order:

1. File defined in environment variable `$SLIC_DEFAULT_CONF_FILE`
2. File `~/.MaxCompiler_slic_user.conf`
3. File defined in environment variable `$SLIC_CONF_FILE`
4. Settings defined in environment variable `$SLIC_CONF` (used by MaxIDE and example Makefiles)

There are integer, Boolean and string settings, shown with their default values and description in *Table 7*. Settings are defined as key-value pairs of the form `option=value`. In a configuration file, one pair is defined per line, for example:

```
#set timeouts
default_pcie_timeout = 60
default_wfi_timeout = 60
```

In the environment variable, pairs are separated by the ; character, for example:

```
export SLIC_CONF="default_pcie_timeout = 60;default_wfi_timeout = 60"
```

Comments can be added to the file by beginning a line with the # character. Empty lines are ignored.

10.13 Debug directories

A SLiC application may produce a number of debug output files, as covered in *section 5*:

- when the application is run from within MaxIDE, these output files are generated in a time-stamped directory present under the `debug` in the corresponding RunRule folder, e.g. `RunRules/Simulation/debug/2013.07.03-14.16.30-380-BST`;
- when the application is run from the command line, these outputs are generated in a directory below the current directory. By default, the name of this debug directory is `debug` if no preexisting directory with that name is present, otherwise, a suffix is added to the `debug`, yielding `debug_1`, `debug_2`, etc. This behavior can be modified by way of the `debug_dir` and `find_next_debug_dir` configuration keywords presented in *Table 7*.

10.14 SLiC Installer

`sliccompile` supports generating installers generated from .max files that allow end users to install bindings for their chosen language.

To produce an installer using `sliccompile` specify the target as 'installer'. For example to create an installer for the moving average example run:

```
[user@machine]$ sliccompile -t installer -m MovingAverage.max
```

This creates an installer named `MovingAverage_installer`. This now accepts Python, MATLAB and R target for auto-generated bindings in these languages. To produce Python bindings, for instance, run

```
[user@machine]$ MovingAverage_installer -t python
```

This generates Python bindings for the `.max` file the installer was built with in the same way that sliccompile would if passed the `.max` file.

When creating an installer passing multiple `.max` files for multiple `.max` files is supported. Each must be specified after `-m` switch. E.g.

```
[user@machine]$ sliccompile -t installer -m VECTIS/MovingAverage.max -m
    CORIA/MovingAverage.max
```

The platform switch (`-p`) can then be passed to the installer to build the binding for the specified platform. E.g.

```
[user@machine]$ MovingAverage_installer -t python -p VECTIS
```

11

Controlled Inputs and Outputs

There are known knowns ... There are known unknowns ... But there are also unknown unknowns. There are things we do not know we don't know.

– Donald Rumsfeld

In the simplest case, all input and output streams have the same size, as for example in a simple dataflow program that multiplies every input value by a constant to produce its output. If we have a Kernel that adds every two input data items together to produce a single output, however, the output stream is half the size of the input stream. To deal with such input and output streams of non-uniform length, we use **controlled inputs** and **controlled outputs**.

11.1 Controlled inputs

We control dataflow inputs using streams of Boolean values, telling the gate for a dataflow input to be open or closed during each tick. A controlled input is declared using an extended version of the familiar method `io.input` with an additional argument for the control stream:

__io__.input(__String__ name, __KernelType__ type, DFEVar __control__)

Computing in a DFE is driven by data. The availability of data at the gate of an operation makes computation happen as the data flows through the computational units. Dataflow computation pauses when there is no valid data at any of the enabled inputs. For example, dataflow computation stalls if there is no data to compute on. If all enabled inputs for a particular kernel have valid data then the dataflow kernel is active. However, if all the inputs for a particular kernel are disabled, then the dataflow kernel does not wait for external data and simply keeps processing internal loops based on internal kernel state. So in essence, controlled inputs really also control execution of computation inside a dataflow kernel.

Assuming that valid data exists at all inputs and that outputs have room to write to their output buffers then:

- Each time a Boolean ‘1’ appears in the stream at the control input to a Kernel input, a new value from the input stream is passed into the Kernel.
- Each time a Boolean ‘0’ appears in the stream at the control input to a Kernel input, the previous value from the input stream is passed into the Kernel. If the designer of the Kernel prefers that a value other than the previous value, zero for example, should be streamed in, then they need to specify this explicitly in their design using a multiplexer.

11.2 Controlled outputs

As before, a controlled output is declared using an extended version of the familiar method `io.output` with an additional argument for the control stream:

__io__.output(__String__ name, __KernelObject__ output, __KernelType__ type, DFEVar __control__)

Assuming that valid data exists at all inputs and that outputs have room to write to their output buffers then:

- Each time a Boolean ‘0’ appears in the stream at the control of an output, the output stream’s value is discarded and is not passed out of the Kernel.
- Each time a Boolean ‘1’ appears in the stream at the control of an output, the output stream’s value is passed out of the Kernel.

11.3 Simple controlled input example

Listing 33 shows the source for an example using a controlled input. The corresponding Kernel graph is shown in *Figure 37*.

Input `a` and input `c` are continuous data streams that pass inputs to the core whenever there is data available:

```
        DFEVar a = io.input("a", dfeUInt(dataWidth));
        DFEVar c = io.input("c", dfeBool()) ;
```

Input `b` only passes inputs to the core when the current value of input stream `c` is 1:

```
        DFEVar b = io.input("b", dfeUInt(dataWidth), c) ;
```

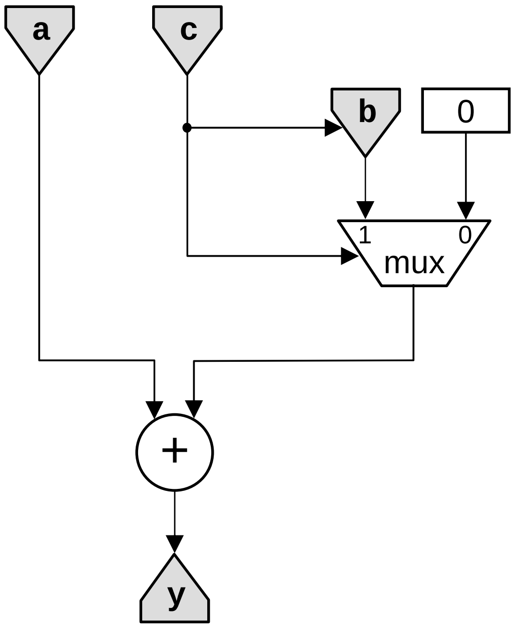


Figure 37: Kernel Graph featuring simple use of controlled input

When the Boolean value from input stream `c` is 0, the previous value of `b` passes into the core. A multiplexer uses the control stream `c` to select between the current value of `b` or 0, and thus if `c` is 0 then the output is `a + 0` instead of `a + b`:

```
DFEVar result = a + (c ? b : 0);
```

11.4 Example for an input controlled by a counter

The control for an input or output does not have to derive from an input: input and output controls can also be derived from internally generated streams such as counters. The output of a counter can be passed to a comparator to generate the necessary Boolean stream.

Listing 34 shows an implementation of a Kernel that uses a counter to control one of its inputs.

The first 10 values from the controlled input stream `b` are added to continuous stream `a` and output in stream `y`. After the first 10 elements of `a` and `b`, the unmodified data from stream `a` is output in stream `y`.

We use a counter to output only the first 10 elements of stream `b`:

```
DFEVar readLimit = io.scalarInput("readCount", dfeUInt(32));
DFEVar count = control.count.simpleCounter(32);
DFEVar read = count < readLimit;

// Inputs
DFEVar b = io.input("b", dfeUInt(32), read);
```

A multiplexer is used to select between adding either the first ten elements of `b` or zero (to give the unmodified value) to `a`:

```
DFEVar result = a + (read ? b : 0);
```

Listing 33: Class with a simple controlled input (SimpleControlledInputKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 11      Example: 1      Name: Simple controlled Input
 * MaxFile name: SimpleControlledInput
 * Summary:
 *     Kernel design using a controlled input. Inputs a and c are continuous
 *     data streams that will pass inputs to the core whenever there is data
 *     available. Input b will only pass inputs to the core when the current
 *     value of input stream c is 1.
 */

package simplecontrolledinput;

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

class SimpleControlledInputKernel extends Kernel {

    private static final int dataWidth = 32;

    SimpleControlledInputKernel(KernelParameters parameters) {
        super(parameters);

        // Inputs
        DFEVar a = io.input("a", dfeUInt(dataWidth));
        DFEVar c = io.input("c", dfeBool());

        DFEVar b = io.input("b", dfeUInt(dataWidth), c);

        // Logic
        DFEVar result = a + (c ? b : 0);

        debug.simPrintf("c: %d\n", c);
        // Output
        io.output("y", result, dfeUInt(dataWidth));
    }
}
```

The result is then written to `y`:

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

The example uses a simple 32-bit counter for clarity, but this wraps when it reaches 2^{32}, so the Kernel tries to read in another 10 elements from input stream `b`. A complex counter in `STOP_AT_MAX` mode provides a more robust implementation that only ever reads in the first 10 elements of `b`.

Listing 34: Class with a counter controlled input (CounterControlledInputKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 11      Example: 2      Name: Counter Controlled Input
 * MaxFile name: CounterControlledInput
 * Summary:
 *       Kernel design that uses a counter to control one of its inputs.
 */

package countercontrolledinput;

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

public class CounterControlledInputKernel extends Kernel {
    public CounterControlledInputKernel(KernelParameters parameters) {
        super(parameters);

        // Control Counter
        DFEVar readLimit = io.scalarInput(”readCount”, dfeUInt(32));
        DFEVar count = control.count.simpleCounter(32);
        DFEVar read = count < readLimit;

        // Inputs
        DFEVar b = io.input(”b”, dfeUInt(32), read);
        DFEVar a = io.input(”a”, dfeUInt(32));

        // Logic
        DFEVar result = a + (read ? b : 0);

        // Outputs
        io.output(”y”, result, dfeUInt(32));
    }
}
```

Exercises

Exercise 1: Counter-controlled input

Through the use of a counter-controlled input, create a Kernel design that merges two color images. The image blue.ppm is a 512x512 image that should serve as one input to the design. The other input should be the 256x256 image `lena256.ppm`. The output should be a 512 x 512 color image with the 256x256 Lena image centered on the blue background. The output should look like *Figure 38*.

Note that the input image stream comes into the Kernel as a sequence of three color components per pixel, Red, Green and Blue, one component per Kernel tick.

A copy of the image shown in *Figure 38* is provided in `lena_merged.ppm` for comparison purposes.

Exercise 2: Controlled output

Take the image result file from Exercise 1. Through the use of a controlled output, discard 3 out of every 4 pixels to produce an output that is the input image scaled to 256x256 size. For each group of four pixels {`(0,0), (1,0), (0,1), (1,1)`}, only pixel `(0,0)` should be taken. The output should look like *Figure 39*.

Figure 38: Blue-Framed Lena Image

Figure 39: Scaled Lena Image

12

On-chip FMem in Kernels

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.

– John von Neuman

DFEs provide two basic kinds of memory: FMem and LMem. **FMem** (Fast Memory) is on-chip Static RAM (SRAM) which can hold several MBs of data. Off-chip LMem (Large Memory) is implemented using DRAM technology and can hold many GBs of data. The key to efficient dataflow implementations is to choreograph the data movements to maximize the reuse of data while it is in the chip and minimize movement of data in and out of the chip. In this section we address the use of FMem directly within kernels.

All inputs and outputs to kernel memories are streams:

- Streams of addresses go in.
- Streams of data come out.
- Where data is being written, streams of data input go into the RAM.

Kernel memories have multiple streams in and out of the memory that can look at the same set of data in a single tick.

12.1 Allocating, reading and writing FMem

All kernel memory uses the same basic declaration, taking two arguments: the type of the data stored in the memory and the number of items to be stored. The method `alloc` returns a `Memory` object which can then be used to access the memory.

```
Memory<DFEVar> mem.alloc(DFEType type, int depth)
```

Memory is parameterized with a Java generic `DFEVar` indicating the type of stream that the memory will retain. Memories can also contain composite kernel types in which case the appropriate alternative string would be used (`DFEVector`, `DFEComplex`, etc).

Memory which is read and written is termed a RAM, while memory that is only read is termed a ROM. Example uses for RAMs include storing a window into an input or output stream, state to be reused within a Kernel and reordering of data. Data can be written into a RAM via the call:

```
Memory.write(DFEVar address, DFEVar data, DFEVar enable)
```

The write method has three input streams: addresses to write to, the data to write to that address and a 1-bit enable which indicates whether the write should be executed or not in that tick. The enable is vital to allow the programmer to selectively control what data is written into a RAM.

A second stream stream of outputs can be read from specified addresses in the RAM:

```
Memory.read(DFEVar address)
```

There are some restrictions on writing into memories. In particular:

- If you write to a memory address in a kernel tick you can not call `read` with the same address in the same tick. Attempting to do so will return undefined data.
- You are limited to either a maximum of 2 calls to `write` and no calls to `read` on a memory, or 1 call to `write` and any number of calls to `read`.

> The content of a memory is undefined when the dataflow engine is first loaded. Once the content is set by the user, the data in the memory persists when the Kernel is reset.

12.1.1 Memory example

Example 1 shows a memory used to reverse the order of data in an input stream. The Kernel source is shown in *Listing 35*.

The memory is declared to contain single-precision floating point numbers:

```
Memory<DFEVar> reverseRam = mem.alloc(dfeFloat(8,24), DATA_SIZE);
```

Data is written into the memory using a call to `write`:

```
reverseRam.write(inputAddress, inputData, readingInput);
```

The output is created with a `read` call:

```
DFEVar outputData = reverseRam.read(outputAddress);
```

The code in the example has two phases of operation:

Phase 1 : the input stream is written into the memory.

Phase 2 : the data is read out in reverse order from the memory.

A counter counts to twice the size of the data set stored in the memory. Dropping the most significant bit from the counter gives an incrementing address to each element in the memory to write the input data. Reversing this address gives a decrementing address to each element to read the data from the memory in reverse order. Data is written into the memory while the counter is less that the size of the memory. When the counter hits the size of the memory, the input stream is disabled and the contents of the memory are written out to the output stream in reverse order.

This example is not a general-purpose method for reversing an arbitrarily-sized input stream as it only reverses chunks of the stream that are the size of the memory.

A useful function when working with memories is `MathUtils.bitsToAddress(dataSize)`, which returns the number of bits required to address a memory of `dataSize`.

12.2 Using memories as read-only tables

One common use of kernel memories is to store tables of infrequently changing constants, for example coefficients, as Read-Only Memory (ROMs). This gives the equivalent behavior to using a large number of constants and multiplexing between them, but is more space and performance efficient when there are more than a few data items.

ROM tables can be initialized with contents specified either as an array of doubles or using `Bits`.

12.2.1 ROM example

Example 2 demonstrates the use of FMem as a ROM. The Kernel source for this example is shown in *Listing 36*. This simple example takes an input stream of addresses and outputs the contents of the ROM. The contents of the ROM are initialized with the first quarter of a sine curve using standard Java math routines, by calling the `setContents` function.

A single `read` call generates an output from the memory. Multiple read calls with different address inputs can return different values from the table in the same kernel tick.

12.2.2 Setting memory contents from the CPU

Memories can optionally be **mapped**, which allows them to be changed by the CPU at runtime, ideally when the kernel is inactive. Mapped memories are an alternative to setting memory contents at compile-time only, and are very useful for values that change slowly but sufficiently frequently that it would be undesirable to recompile the maxfile.

Memories that are mapped to the CPU are declared with the same syntax as normal memories, but with an additional call to declare a name which can be used by the CPU to set the memory contents.

*Memory.mapToCPU(**String** name)*

It is important to note that using a mapped memory is an alternative to setting the contents in the kernel, i.e. it is not valid to call both `setContents` and `mapToCPU` on the same memory. Mapped memories should be initialized with contents from CPU code.

12.2.3 Mapped ROM example

Example 3 demonstrates the use of a mapped memory as a ROM with multiple reads. The Kernel source for this example is shown in *Listing 37*. In this example, two input streams of addresses are passed to `read` calls for the memory and the corresponding outputs are connected directly to the kernel's output streams.

The following lines show the instantiation of the ROM and mapping it to the CPU with the name `‘‘mappedRom’’`:

```
        Memory<DFEVar> mappedRom = mem.alloc(dfeFloat(8,24), dataSize);
        mappedRom.mapToCPU("mappedRom");
```

Once the memory is initialized, two calls to `read` are made and the results are connected to the kernel output streams:

```
        DFEVar readA = mappedRom.read(addressA);
        DFEVar readB = mappedRom.read(addressB);

        io.output("outputA", readA, dfeFloat(8,24));
        io.output("outputB", readB, dfeFloat(8,24));
```

The contents of the ROM are set in the CPU code by passing a pointer to the contents as an argument to the SLiC function for running the DFE.

The CPU code for this example is shown in *Listing 38*. A block of memory is allocated and then set up with the desired values; in this case our sine function:

```
    double *romContents = malloc(sizeBytesDouble);
```

```
    generateInputData(
        size,
        inAddressA, inAddressB,
        romContents, romContentsReversed);
```

The contents are then passed to the SLiC function:

```
    DualPortMappedRom(
        size,
        inAddressA, inAddressB,
        outDataA, outDataB,
        romContents);
```

It is often useful to create custom SLiC engine interfaces to separate setting mapped memory values into a specific initialization SLiC interface function and then not set them during the compute function (using the `ignoreMem` function on the engine interface object). This saves uploading identical values for the mapped memory repeatedly every time the DFE runs a computation.

12.3 Creating a memory port which both reads and writes

As an optimization, it is also possible to create a single memory port which both reads and writes in the same tick using the same address:

DFEVar Memory.port(DFEVar address, DFEVar data_in, DFEVar enable, RamWriteMode portMode)

A read-write port has input address, data and write-enable streams and an output data stream; the input address stream is used for both the write and read locations. A read-write memory port always outputs data from the location specified in the input address stream, regardless of the status of the write-enable stream.

Read/write memory ports must be set with a `RamWriteMode`, which can either be `READ_FIRST` or `WRITE_FIRST`. This determines the behavior when data is read from and written in the same tick. In `READ_FIRST` mode, the existing contents of the memory location is read before being written over. In `WRITE_FIRST` mode, the new value propagates directly to the output in the same tick as it is being written. Not all DFE architectures support both modes.

> `RamWriteMode` applies only to determining whether the read or write should be performed first for this port, accessing the same address from another port will return undefined data regardless of what mode is used.

12.4 Understanding memory resources

Kernel memories are implemented using a special on-chip memory resource. The amount of on-chip memory used by an application can often be a determining factor in its performance, so it is worth understanding a little of how memories are built and what the costs of different operations are.

The number of on-chip **BRAM** resources required for a particular kernel memory depends on:

- The number of items in the memory (i.e. its depth).
- The type of the data held in the memory (i.e. its width).
- The number and type of ports.

In general, the larger the memory the more silicon area it will require. Memories with many ports will use also use more on-chip area because the basic storage element must be replicated to provide parallel access. A physical on-chip BRAM resource supports two ports, so if more ports are requested (e.g. many calls to the `read` function) then several instances of the resource will be automatically allocated to provide the correct level of access parallelism. The fact that BRAMs have 2 ports is also the source of the restriction that kernel memories with more than two ports must have at most one write port, since the write data must be copied to all parallel memory instances.

In general, the amount of memory resource required in silicon for memories with only read ports scales with:

$$depth \times width \times (numreadports \div 2)$$

For memories with 1 write port and many read ports, the silicon resource requirements are proportional to:

$$depth \times width \times numreadports$$

(a) Original image *(b)* Processed image

Figure 40: Original image and image after the coefficients have been applied.

Exercises

Exercise 1: Simple ROM

In this exercise, a simple application is supplied that streams in a 256x256 pixel image in the same format as for the exercises in *section 11*: *Controlled Inputs and Outputs*. Modify this example to apply a set of coefficients to each line of the image. There should be one coefficient for each pixel in a line of the image. The coefficients should be floating-point numbers between 0.0 and 1.0. A suitable set of coefficients can be calculated using the equation:

$$c_i = \begin{cases} 1.0 - (1.0/(X/2)) \times ((X/2) - i) & \text{if } i < X/2 \\ 1.0 - (1.0/(X/2)) \times (i - (X/2)) & \text{if } i \geq X/2 \end{cases}$$

Where c_i is the coefficient for the ith pixel in a row and X is the width of a row in the image. *Figure 40* shows the image before and after these coefficients have been applied.

Remember that the input image stream comes into the Kernel as a sequence of three color components per pixel, Red, Green and Blue, one component per tick.

Exercise 2: Dual-port RAM

Modify the previous exercise to read in the coefficients from an input stream and store them in a RAM. Once the coefficients have been read into the RAM, start processing the input data stream using these coefficients as before. Use a separate input stream for the coefficients.

It is important to control the input and output streams to ensure only correct data is read from the input streams and written to the output stream correctly.

Listing 35: A memory used to reverse the data in an input stream (DualPortRamKernel.maxj).

```
package dualportram;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Counter;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Params;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.utils.MathUtils;

class DualPortRamKernel extends Kernel {
    DualPortRamKernel(KernelParameters parameters) {
        super(parameters);

        int DATA_SIZE = 16;

        /*Create a counter to generate the addresses to the RAM. This counts to twice the size of
         * the data set stored in the RAM. */
        int addrBits = MathUtils.bitsToAddress(DATA_SIZE);
        Params addressCounterParams = control.count.makeParams(addrBits+1);
        Counter addressCounter = control.count.makeCounter(addressCounterParams);

        /* Dropping the most significant bit from the counter gives us an incrementing address to
         * each element in the RAM to write the input data. Reversing this address gives us a
         * decrementing address to each element to read the data from the RAM in reverse order. */
        DFEVar inputAddress = addressCounter.getCount();
        DFEVar outputAddress = DATA_SIZE - 1 - addressCounter.getCount();

        inputAddress = inputAddress.cast(dfeUInt(addrBits));
        outputAddress = outputAddress.cast(dfeUInt(addrBits));

        // If the counter is less that the size of the RAM, then we are reading input data
        DFEVar readingInput = addressCounter.getCount() < DATA_SIZE;

        // Read input data during the first half of the counter
        DFEVar inputData = io.input("inputData", dfeFloat(8,24), readingInput);

        /* The input port takes the input address and data input stream. The write-enable is set
         * to readingInput, which is true for the first half of the counter. */
        // The output port takes the decrementing output address
        Memory<DFEVar> reverseRam = mem.alloc(dfeFloat(8,24), DATA_SIZE);
        reverseRam.write(inputAddress, inputData, readingInput);
        DFEVar outputData = reverseRam.read(outputAddress);
        /* When the counter is in its second half, the contents of the RAM will be read out in
         * reverse order. */
        io.output("outputData", outputData, dfeFloat(8,24), ~readingInput);
    }
}
```

Listing 36: A memory used as a ROM, initialized with doubles (RomKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 12      Example: 2      Name: Rom Kernel
 * MaxFile name: Rom
 * Summary:
 *         Kernel design that demonstrates the use of a single port ROM.
 */

package rom;

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

class RomKernel extends Kernel {
    RomKernel(KernelParameters parameters) {
        super(parameters);

        final int addrBits = 8;
        final int dataSize = (int)Math.pow(2, addrBits);

        // Input
        DFEVar address = io.input("address", dfeUInt(addrBits));

        double contents[] = new double[dataSize];
        for (int i = 0; i < dataSize; i++)
            contents[i] = Math.sin(((Math.PI/2.0)/dataSize)*i);

        Memory<DFEVar> table = mem.alloc(dfeFloat(8,24), dataSize);
        table.setContents(contents);

        DFEVar result = table.read( address );

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

Listing 37: A mapped memory used as ROM, with two simultaneous reads (DualPortMappedRomKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 12      Example: 3      Name: Dualport mapped ROM
 * MaxFile name: DualPortMappedRom
 * Summary:
 *       Kernel design that demonstrates the use of a dual port mapped ROM.
 */

package dualportmappedrom;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.utils.MathUtils;

class DualPortMappedRomKernel extends Kernel {
    DualPortMappedRomKernel(KernelParameters parameters, int dataSize) {
        super(parameters);

        int addrBits = MathUtils.bitsToAddress(dataSize);

        // Input
        DFEVar addressA = io.input("addressA", dfeUInt(addrBits));
        DFEVar addressB = io.input("addressB", dfeUInt(addrBits));

        Memory<DFEVar> mappedRom = mem.alloc(dfeFloat(8,24), dataSize);
        mappedRom.mapToCPU("mappedRom");

        // Output
        DFEVar readA = mappedRom.read(addressA);
        DFEVar readB = mappedRom.read(addressB);

        io.output("outputA", readA, dfeFloat(8,24));
        io.output("outputB", readB, dfeFloat(8,24));
    }
}
```

Listing 38: Main function for CPU code demonstrating setting of a mapped ROM (DualPortMappedRomCpuCode.c).

```
int main()
{
    const int size = 256;
    int sizeBytesFloat = size * sizeof(float);
    int sizeBytesDouble = size * sizeof(double);
    int sizeBytesInt = size * sizeof(uint8_t);
    uint8_t *inAddressA = malloc(sizeBytesInt);
    uint8_t *inAddressB = malloc(sizeBytesInt);
    double *romContents = malloc(sizeBytesDouble);
    double *romContentsReversed = malloc(sizeBytesDouble);
    float *outDataA = malloc(sizeBytesFloat);
    float *outDataB = malloc(sizeBytesFloat);

    generateInputData(
        size,
        inAddressA, inAddressB,
        romContents, romContentsReversed);

    printf("Running DFE.\n");
    DualPortMappedRom(
        size,
        inAddressA, inAddressB,
        outDataA, outDataB,
        romContents);

    int status = check(
        size,
        outDataA, outDataB,
        romContents, romContentsReversed);

    if (status)
        printf("Test failed.\n");
    else
        printf("Test passed OK!\n");

    return status;
}
```

13

Talking to CPUs, Large Memory (LMem), and other DFEs

A Dataflow Engine needs to communicate with its LMem (Large Memory, GBs of off-chip memory), CPUs and other DFEs. The Manager in a dataflow program describes the choreography of data movement between DFEs, connecting CPUs, and also the GBs of data in LMem.

Figure 41 illustrates the architecture of a Maxeler acceleration system which comprises dataflow engines (DFEs) attached directly to local memories and to a CPU. In a Maxeler solution, there may be multiple dataflow engines connected together via high-bandwidth **MaxRing** interconnect. A dataflow engine is made up of one or more Kernels and a **Manager**. Within a Kernel, streams provide a predictable environment for the designer to concentrate on data flow and arithmetic. Managers provide a predictable input and output streams interface to the Kernel.

MaxCompiler provides pre-configured Managers, including the **Standard Manager**, and the **Manager Compiler** for creating complex Managers of your own.

The Manager Compiler allows you to create complex Manager designs, with multiple Kernels and complex interaction between Kernels and with IO resources. The **Manager Compiler Tutorial** document, also provided with MaxCompiler, covers the Manager Compiler in detail.

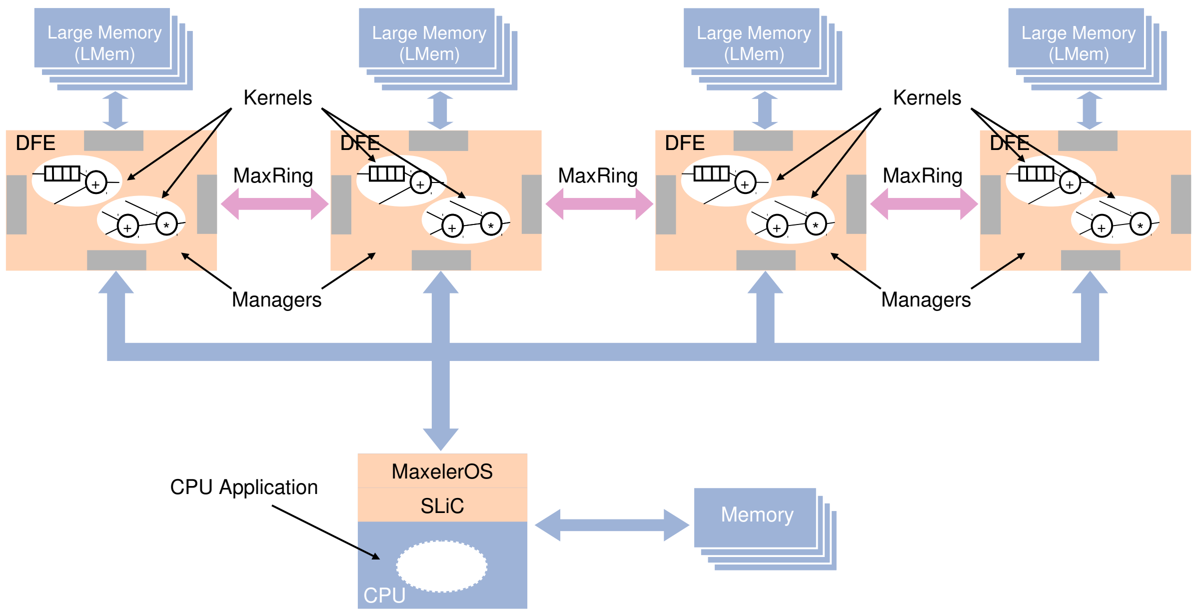


Figure 41: Overview of a Maxeler acceleration system

13.1 The Standard Manager

The Standard Manager provided with MaxCompiler is a general-purpose Manager which supports:

- a single Kernel
- external LMem interfaces
- linking between multiple dataflow engines
- links to the CPU

Figure 42 shows a two-DFE Maxeler acceleration system that can be targeted with the Standard Manager.

All the Manager classes in the examples and exercises so far have used the Standard Manager with all inputs and outputs directly connected to the CPU.

The Standard Manager is constructed with an `EngineParameters` object:

Manager(EngineParameters configuration)

The `EngineParameters` object contains information passed by MaxIDE; you can also add extra parameters to this object.

By default, a clock frequency of 75 MHz is used. You can also set a different clock rate:

***public void** setClockFrequency(**int** clock_frequency)*

The default can be accessed explicitly using `DEFAULT_CLOCK_FREQUENCY`.

A Standard Manager can encapsulate a single Kernel, which is set using this `setKernel`:

***void** setKernel(Kernel k)*

The version of `makeKernelParameters` in the Standard Manager class does not take a string for the name of the Kernel as only one Kernel is allowed in the Standard Manager:

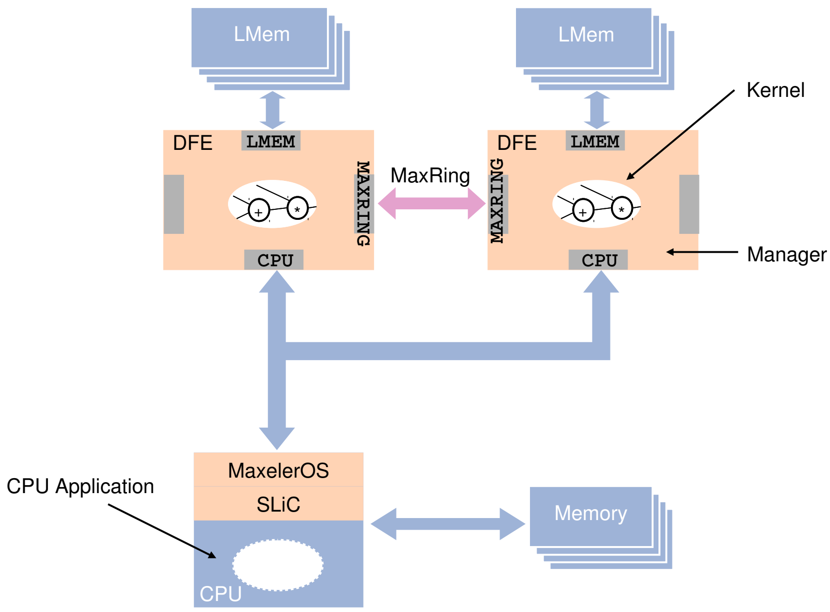


Figure 42: Two-chip acceleration system using the Standard Manager

```
public KernelParameters makeKernelParameters()
```

setIO allows you to link the input and output streams in the Kernel to I/O resources enabled by the Manager:

```
void setIO(Manager.IOType io_type)
void setIO(IOLink... links )
```

The first version of the function allows all inputs and outputs to be set together. All the links have been to the CPU in the examples so far, for example:

```
Manager m = new Manager(new EngineParameters(args));
m.setIO(IOType.ALL_CPU);
```

Figure 43 illustrates how a Manager, Kernel and CPU interact with one input and one output both set to CPU.

Another option for setting all of the I/Os together is NOIO:

```
Manager m = new Manager(new EngineParameters(args));
m.setIO(IOType.NOIO);
```

This builds the Kernel as a block of logic with no connections to the outside world. This is useful for determining the performance of a Kernel in isolation from the rest of the logic and optimization. A BuildConfig object (see *subsubsection 13.4.1*) with the build level set to FULL_BUILD cannot be used in this mode.

The link for each input and output stream in the Kernel can be specified individually using a list of IOLinks. An IOLink is declared using a stream name and the corresponding link type:

```
IOLink link (String io_name, IOLink.IODestination iotype)
```

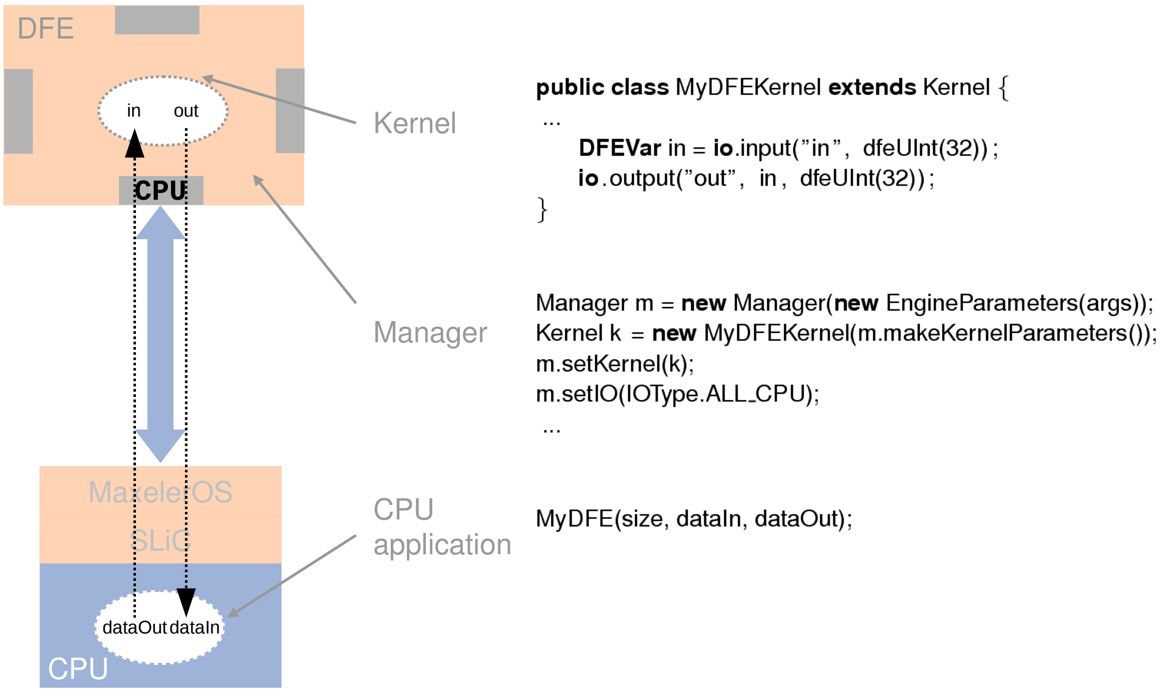


Figure 43: All CPU links on a Standard Manager

`iotype` can be one of:

- `CPU` connects the stream to the CPU
- `MAXRING_A` or `MAXRING_B` connects the stream to one of two, bi-directional MaxRing links on a MAX3
- `LMEM_LINEAR_1D` connects the stream to LMem with a linear address generator
- `LMEM_BLOCKED_3D` connects the stream to LMem with a 3D address generator

13.2 MaxRing communication

The MaxRing interconnect allows data to be transferred at high speed directly between dataflow engines. Each dataflow engine in the system has a direct bidirectional connection to up to two other DFEs, as shown in *Figure 41*.

There are two MaxRing connections on a MAX3: `MAXRING_A` and `MAXRING_B`.

Listing 39: Inter-chip loopback Kernel (MaxringKernel.maxj).

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
 * Chapter: 13      Example: 1      Name: Maxring
 * MaxFile name: Maxring
 * Summary:
 *       Kernel design that takes a scalar input to select whether
 *    it should behave as the left or right DFE.
 */

package maxring;

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

class MaxringKernel extends Kernel {

    MaxringKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar isLeft = io.scalarInput("isLeft", dfeBool());

        DFEVar inA = io.input("inA", dfeUInt(32), isLeft);
        DFEVar inB = io.input("inB", dfeUInt(32), ~isLeft);

        io.output("outA", inA, dfeUInt(32), isLeft);
        io.output("outB", inB, dfeUInt(32), ~isLeft);
    }
}
```

13.2.1 Example with loop-back across two chips

Example 1 shows a simple application that reads a stream of data from the CPU into one dataflow engine, passes the data over a MaxRing link to the second DFE and then writes the data back from the second DFE to the CPU. *Listing 39* shows the source code for the Kernel.

As both dataflow engines are identically configured, the Kernel takes a scalar input to select whether it should behave as the left or right DFE in *Figure 44*:

```
        DFEVar isLeft = io.scalarInput("isLeft", dfeBool());
```

The inputs and outputs are controlled by `isLeft` to either read from `inA` and write to `outA`, or read from `inB` and write to `outB`:

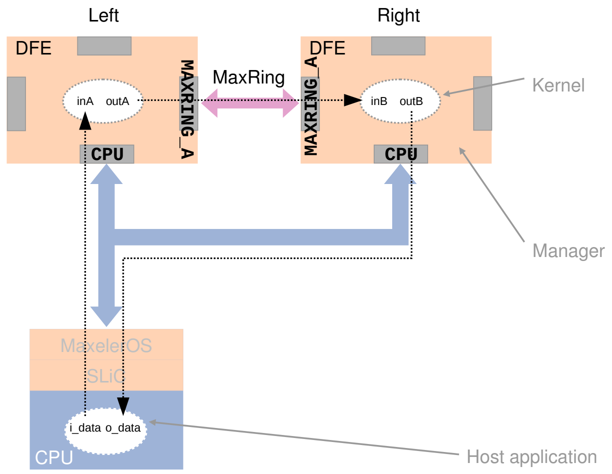
```
        DFEVar inA = io.input("inA", dfeUInt(32), isLeft);
        DFEVar inB = io.input("inB", dfeUInt(32), ~isLeft);

        io.output("outA", inA, dfeUInt(32), isLeft);
        io.output("outB", inB, dfeUInt(32), ~isLeft);
```

Figure 45 shows the resultant Kernel graph for the example.

The Manager connects the inputs and outputs to the CPU and MaxRing links:

```
        m.setIO(link("inA", CPU), link("inB", MAXRING_A),
                link("outA", MAXRING_A), link("outB", CPU));
```




```
public class MaxRingKernel extends Kernel {
  ...
    DFEVar inA = io.input("inA", dfeUInt(32), isLeft);
    DFEVar inB = io.input("inB", dfeUInt(32), ~isLeft);

    io.output("outA", inA, dfeUInt(32), isLeft);
    io.output("outB", inB, dfeUInt(32), ~isLeft);
  ...

Manager m = new Manager(new EngineParameters(args));
Kernel k = new IfpgaKernel( m.makeKernelParameters() );

m.setKernel(k);
m.setIO(link("inA", CPU), link("inB", MAXRING_A),
    link ("outA", MAXRING_A), link("outB", CPU));


Maxring_actions_t **actions = malloc(numEngines * sizeof(Maxring_actions_t*));

for (int i = 0; i < numEngines; i++) {
    int isLeft = (i == 0);
    actions[i] = malloc(sizeof(Maxring_actions_t));
    actions[i]->param_N = size;
    actions[i]->param_isLeft = isLeft ? 1 : 0;
    actions[i]->instream_inA = isLeft ? inA : NULL;
    actions[i]->outstream_outB = isLeft ? NULL : outB;
}

printf ("Running DFEs.\n");
max_file_t *maxfile = Maxring_init ();
max_engarray_t *engines = max_load_array(maxfile, numEngines, "*");
Maxring_run_array(engines, actions);
```

Figure 44: MaxRing links on a Standard Manager

13.3 Large Memory (LMem)

The Standard Manager allows you to connect any number of streams to the LMem on the dataflow engine. This allows large amounts of data to be kept local to the dataflow engine and iterated over. The LMem appears as one contiguous piece of memory. There are different access patterns available for the memory:

- `LMEM_LINEAR_1D` connects the stream to LMem with a simple linear address generator.
- `LMEM_BLOCKED_3D` connects the stream to LMem with a 3D address generator.

The Manager Tutorial covers more advanced `LMEM_LINEAR_1D` and `LMEM_BLOCKED_3D` usage, as well as other memory access patterns such as `LMEM_STRIDED_2D`.

In the Standard Manager, each stream has its own address generator. The parameters for the behavior of the memory address generators for each stream can be set up either in the CPU code or in a SLiC engine interface to simplify the CPU interface to the DFE.

The Standard Manager provides a CPU input stream called `"write_lmem"` and an output stream to the CPU called `"read_lmem"` for accessing the LMem linearly in the CPU software. MaxCompiler automatically creates two SLiC engine interfaces, `<.max file name>_writeLMem` and `<.max file name>_readLMem`, to write to and read from the LMem from the CPU using these streams.

The memory controller and its address generators work in *bursts*. The burst length can be retrieved in CPU code through SLiC:

```
int max_get_burst_size(
        max_file_t * const maxfile,
        const char * const name);
```

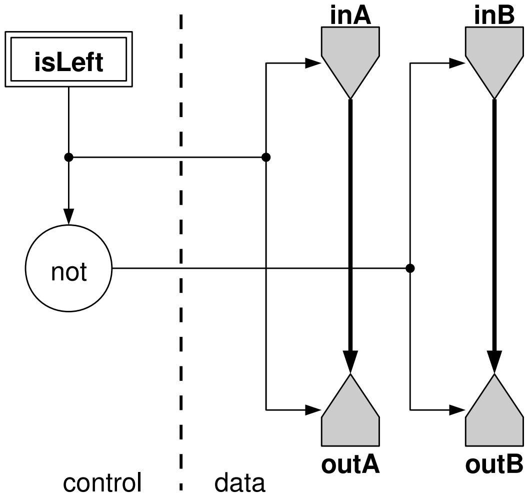


Figure 45: Kernel for the simple MaxRing loopback

The burst length is also available in the Manager through the `getBurstLength` method.

All dimensions provided as arguments to address generator functions are in *bytes* and must be a multiple of the burst length.

13.3.1 Linear address generators

A linear address generator is set up using two arguments `address` and `size` to address a block of LMem, for example in a SLiC engine interface:

```
public void setLMemLinear(String streamName,
                          InterfaceParam address,
                          InterfaceParam size)
```

`setLMemLinear` reads `size` bytes from `address`, then returns to `address` to start reading again.

13.3.2 3D blocking address generators

A 3D Blocking address generator operates in a coordinate system where the unit of size in each dimension is in *bytes*. A block of size *(rwSizeFast, rwSizeMed, rwSizeSlow)*, with its origin at *(offsetFast,offsetMed,offsetSlow)* is read from a larger block of size *(arraySizeFast, arraySizeMed, arraySizeSlow)*:

```
public  void setLMemBlocked(String streamName,
                            long address,
                            long arraySizeFast,
                            long arraySizeMed,
                            long arraySizeSlow,
                            long rwSizeFast,
                            long rwSizeMed,
                            long rwSizeSlow,
```

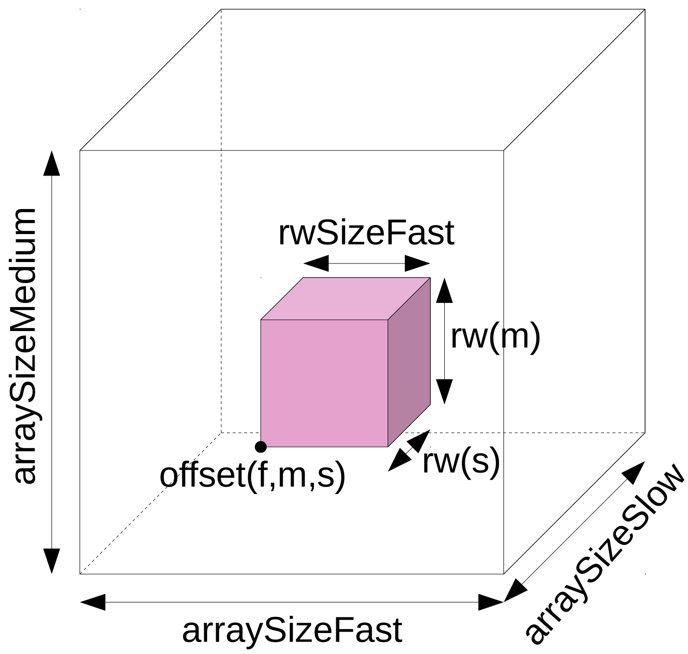


Figure 46: 3D Blocking Address Generator, where offset(f,m,s) is the point (offsetFast, offsetMed, offsetSlow), rw(m) is rwSizeMed and rw(s) is rwSizeSlow

```
long offsetFast,
long offsetMed,
long offsetSlow)
```

The terms fast, medium and slow refer to the speed of indexing the LMem in that dimension: the most efficient access to the LMem indexes in the fast dimension first, then the medium, then the slow.

Figure 46 shows the meaning of the arguments in 3D space.

13.3.3 Large Memory (LMem) example

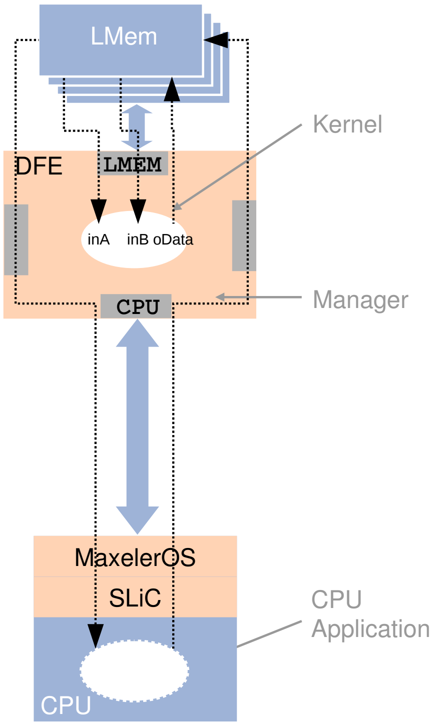
This example shows a Kernel with two inputs connected to LMem and an output to LMem. There are no inputs or outputs to the CPU from the Kernel.

In this example, the two input streams are read from different locations in memory, added together and written back to a third location. The input data is written directly from the CPU code via the "write_lmem"stream before the Kernel runs and the output data is read back via the "read_lmem" stream once the Kernel has completed. *Figure 47* shows the interaction of the Kernel, Manager and CPU code.

The body of the Kernel simply connects the output stream to the sum of the two input streams:

```
        DFEVar inA = io.input("inA", dfeUInt(32));
        DFEVar inB = io.input("inB", dfeUInt(32));

        io.output("oData", inA+inB, dfeUInt(32));
```




```
public class LMemLoopbackKernel extends Kernel {
 ...
     DFEVar inA = io.input("inA", dfeUInt(32));
     DFEVar inB = io.input("inB", dfeUInt(32));

     io.output("oData", inA+inB, dfeUInt(32));
 ...


Manager m = new Manager(new EngineParameters(args));
Kernel k = new LMemLoopbackKernel(m.makeKernelParameters());
m.setKernel(k);

m.setIO(link("inA", IODestination.LMEM_LINEAR_1D),
                     link ("inB", IODestination.LMEM_LINEAR_1D),
                     link ("oData", IODestination.LMEM_LINEAR_1D));


 printf ("Loading DFE memory.\n");
LMemLoopback_writeLMem(size, 0, inA);
LMemLoopback_writeLMem(size, size, inB);
 printf ("Running DFE.\n");
LMemLoopback(size);
 printf ("Reading DFE memory.\n");
 int32_t *outData = malloc(sizeBytes);
LMemLoopback_readLMem(size, 2 * size, outData);
```

Figure 47: LMem links on a Standard Manager

The Manager attaches the three streams to linear memory address generators:

```
m.setIO(link("inA", IODestination.LMEM_LINEAR_1D),
          link ("inB", IODestination.LMEM_LINEAR_1D),
          link ("oData", IODestination.LMEM_LINEAR_1D));
```

The CPU code first creates two buffers of data:

```
int32_t *inA = malloc(sizeBytes);
int32_t *inB = malloc(sizeBytes);

for (int i = 0; i < size; i++) {
    inA[i] = i;
    inB[i] = size - i;
}
```

The two buffers are written to two separate locations in the LMem:

```
LMemLoopback_writeLMem(0, sizeBytes, inA);
LMemLoopback_writeLMem(sizeBytes, sizeBytes, inB);
```

To run the Kernel, the default SLiC engine interface can be run:

```
 printf ("Running DFE.\n");
LMemLoopback(size);
```

The SLiC function returns once the Kernel has completed writing its output to the LMem. Now the contents of the LMem at the output stream location can be read back to the CPU:

```
int32_t *outData = malloc(sizeBytes);
LMemLoopback_readLMem(2 * sizeBytes, sizeBytes, outData);
```

13.4 Building DFE configurations

All Managers extend the `maxcompiler.v1.managers.DFEManager` class which provides several methods. Specific Managers have additional methods.

abstract void *build()*

`build` launches the build process. It can be called only once.

void *logMsg(**String** msg, Object... args)*
void *logWarning(**String** msg, Object... args)*

`logMsg` and `logWarning` allow you to log messages that are output in the `_build.log` file in the build directory. This is preferable to printing directly to the console as these messages are saved for reference. Messages are formatted using a `printf`-like format.

`makeKernelParameters` is required for constructing a Kernel and supplies the name:

*KernelParameters makeKernelParameters(**String** kernel_name)*

`BuildConfig` objects are useful for controlling the configuration of the build process:

BuildConfig getBuildConfig()
void *setBuildConfig(BuildConfig build_config)*

13.4.1 `BuildConfig` objects

A `BuildConfig` object can be used to set and retrieve build settings. Methods available include `setBuildEffort`:

void *setBuildEffort (BuildConfig. Effort effort)*

Build effort tells the third party tools how much effort to put into trying to find an implementation of the circuit to meet the design requirements (clock speed and area). The options available are `HIGH`, `LOW`, `MEDIUM`, `VERY_HIGH`. Though it may be tempting to always run with high effort levels, builds can take a long time when constraints are tight, so lower effort levels are useful for iterative test and optimization.

void *setBuildLevel(BuildConfig.Level level)*

Build level tells MaxCompiler up to which stage to run the build process. By default, MaxCompiler runs the complete build process and produces a `.max` file. Options available are:

- `FULL_BUILD` (default) runs the complete process
- `COMPILE_ONLY` stops after producing the VHDL output from MaxCompiler
- `SYNTHESIS` compiles the VHDL output
- `MAP` maps the synthesized design to components in the silicon device
- `PAR` places and routes the design for the chip, producing a DFE configuration

Levels other than `FULL_BUILD` are typically only useful when debugging a build-related problem.

Multi-pass Place and Route (MPPR) is the term used for automatically running the place and route process on the same input design multiple times with different starting conditions in order to see which run gives the best results. `setMPPRCostTableSearchRange` instructs the silicon vendor's map and place and route tools to use a range of "cost tables" to initialize a multi-pass run:

void *setMPPRCostTableSearchRange(**int** min, **int** max)*

MaxCompiler can produce a timing report based on the output of the place and route tools. This can be used to identify timing issues which may help when optimizing the design. Timing reports are enabled by default but can be enabled and disabled explicitly:

void *setEnableTimingAnalysis(**boolean** v)*

Exercises

Exercise 1: LMem and MaxRing loop-back

Using the two examples in this section for help, write an application using two devices where:

1. The first device reads data from CPU and passes it to the other via an inter-chip link.
2. The second device reads data from the MaxRing link and writes it to LMem.
3. The CPU code reads the data from LMem and checks it against the input.

Figure 48 shows the required flow of data through the system.

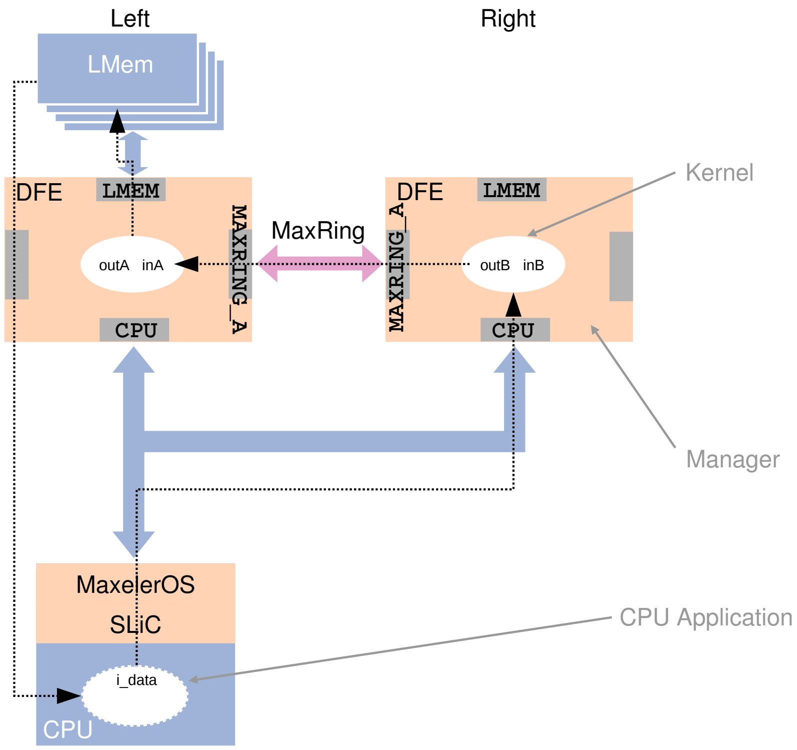


Figure 48: Exercise required data flow from CPU, via MaxRing links, to LMem

A

Java References

For further information on the Java language we recommend the following resources:

- http://docs.oracle.com/javase/tutorial/java/index.html
  An overview of Java and an introduction to its major syntactical features.
- http://docs.oracle.com/javase/tutorial/collections/index.html
  An overview of the Java “Collections” API which is used often in MaxCompiler interfaces.
- http://docs.oracle.com/javase/6/docs/api/
  API documentation for the standard Java libraries.
- http://www.java-tips.org/java-se-tips/java.lang/using-the-varargs-language-feature.html
  Introduction to using variable-argument methods in Java which are also common in MaxCompiler interfaces.

B

On Multiscale Dataflow Research

– Oskar Mencer and Maxeler Advisors, December 2012

Once upon a time: http://www.cs.berkeley.edu/~kubitron/asplos98/abstracts/oscar_mencer.ps

Today, Multiscale Dataflow combines static dataflow computing on Dataflow Engines (DFEs) with optimization on multiple levels of abstraction and scale: from mathematics and algorithms all the way to arithmetic and logic gates. Such vertical optimization is needed for mission-critical computations where every second counts. Being part of MAX-UP opens up a wide range of opportunities to investigate theory and practice of computing at the physical limits of a given generation of technology. This document is intended for the seasoned MAX-UP researcher, writing papers based on MPC systems and looking for interdisciplinary research and new funding opportunities to advance multiscale dataflow computing. For Maxeler publications see: http://www.maxeler.com/publications/

1 Multiscale Dataflow Computing

Multiscale Dataflow Computing addresses the requirements of very large datasets and computationally intensive problems on these datasets. We get a lot of requests to clarify which applications dataflow computing is most suitable for and how an FPGA chip compares to some multi-core chip running a small loop-nest with a small dataset. However, instead of comparing chips, or suitability of algorithms, the key to meaningful investigation is to ask which problem sizes best balance the dataflow computation given a full system configuration, including multiple nodes with storage, networking and compute units. The hard part for researchers is to get a meaningfully large dataset. In the example referenced below, we show 3D finite difference running on a Maxeler 1U compute node with 4-8 dataflow engines (DFEs), and find that speedup with MPC systems grows with problem size, especially for large problems beyond a mesh with 1000^3 points.

[**FD modeling beyond 70Hz with FPGA acceleration**. D. Oriato, O. Pell (Maxeler), C. Andreoletti and N. Bienati (ENI). Society of Exploration Geophysicists 2010, Denver, USA, Oct. 2010]

2 Algorithm Transformation

So what else could one do to publish (not perish)? There are many opportunities to explore algorithm transformations. A question rich in research potential is: *How could this algorithm be transformed to run optimally on the particular system configuration.* In essence, a multiscale dataflow machine brings with it a vast space for developing novel versions of many existing algorithms. Even algorithms that so far have not been popular are getting a new chance to shine. Following common folklore, one can always trade off FFTs with convolutions in the time domain. And dataflow machines really excel at the convolution, but there are a few notable exceptions depending on the size of the FFT and the amount of computation involved. Back to our example with 3D finite difference, there is flexibility in designing stencils of different shapes (locations of coefficients for the convolution). A significant project at Stanford showed how a cube stencil brings a 5x advantage over the star stencil, even though the star stencil is optimal on a CPU.

[**Accelerating 3D Convolution using Streaming Architectures on FPGAs**. H. Fu, R. G. Clapp, O. Mencer and O. Pell. (Stanford University, Imperial College London, Maxeler Technologies), 79th Society of Exploration Geophysicists (SEG), Houston, October 2009.]

3 From Bits to Numbers

Of course all this is only the tip of the iceberg. Considering the representation of data (how do you use zeros and ones to describe all the other numbers), it may well be possible to invent new ways to design the data structures and arrange the layout and access of numbers in memory. By finding innovative ways to streamline dataflow memory accesses, DFE technology can really flex its muscle. And data layout is just the beginning. The encoding of arrays of numbers can be investigated by encoding each array differently: Are there ways to compress the data or expand the data in order to achieve further acceleration? The famous examples here are gigantic sparse matrix computations in the Finite Elements (FE) method, where the same shaped sparse matrix has to be read and written over and over and over again. Imagine if one had a memory controller that specializes in reading and writing this particular sparse matrix in compressed form.

[**Surviving the End of Scaling of Traditional Microprocessors in HPC**. O. Lindtjrn, R. G. Clapp, O. Pell, O. Mencer and M. J. Flynn. (Schlumberger,Stanford University,Maxeler Technologies) IEEE HOT CHIPS 22, Stanford, USA, August 2010.]

4 Memory o'Memory

Making computation happen quicker is all about joint layout of data and compute modules. The research challenge starts when the data structures do not support controlling memory accesses, making memory access "random". But memory controllers cannot help with random memory access, or can they? For irregular memory access issues such as applications with graph data structures, a top research question is: How can we expand the data by, for example, looking at the adjacency matrix representation or a sorted array representation of nodes and put all that together with metadata about location in the graph. We could do all that without pointer-based linked-nodes which create all the "random" memory accesses. Since a dataflow machine is all about data and memory, using more memory (redundancy) to regularize (or regularise) memory access is a counter intuitive transformation with significant potential. All such new memory layouts offer wide design spaces with wide opportunities for advancing our understanding of algorithms and computation in general.

[**Accelerating Unstructured Mesh Computations using Custom Streaming Architectures**. Kyrylo Tkachov, Supervisor: Prof. Paul H J Kelly (Imperial College London)]

5 Adapting Models to Dataflow

Let us also consider the reason to compute in the first place. Looking at the objective of the computation rather than a particular implementation, how can we attack a much more ambitious objective given the capabilities of the dataflow systems? Is it possible to add more "Physics", a more compute-intensive approximation method, or try to achieve a leap in capability? Just assuming that our 3D finite difference is solving an acoustic wave equation clearly limits the potential research of solving the underlying partial differential equation. Looking at the overarching objective of creating 3D images of the earth, scientists contemplate the use of the elastic wave equation which models the world a lot more accurately, or even add viscosity to arrive at the visco-acoustic/elastic equations. It turns out that dataflow machines become more and more attractive the more complexity becomes available, offering the possibility to accelerate the development of next generation models and science in general. Writing a paper about Physics models requires collaboration between scientists developing models and computer engineering researchers. Such collaboration offers substantial gain but it is no small political challenge in any (academic) environment. One of the goals of MAX-UP is to support and facilitate such interdisciplinary collaboration.

[**Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications**. O. Lindtjrn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn and H. Fu. (Schlumberger, Stanford,Maxeler,Tsinghua University), IEEE Micro, vol. 31, no. 2, March/April 2011.]

6 Numerics

The most discrete field in computational research lies in numerics. Contributing to the understanding of interaction between number representation and convergence of algorithms, number of iterations needed, and accuracy of inputs and final results, may not be an entirely un-useful endeavor, helping scientists to better understand their problems but also making a statement about the appropriateness and stability of their results. What does this really mean? On the simple end, it is possible to explore various number representations such as variable bitwidths for floating point or fixed point (challenging many proponents of the rigid IEEE floating point standards by using sub-single precision and super-quad precision). To further shrink the representation, we have block floating point with one exponent for a block of mantissa values, logarithmic numbers, all the way to mixed approaches minimizing bitwidth combined with statistical methods, such as in for example:

[**A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems**. Chow, Tse, Jin, Luk, Thomas, Leong, (Imperial, Sydney), FPGA 2012]

7 Arithmetic

Tightly coupled with representation we have arithmetic, and in that case, the space for research exploration contains yet many more alternatives and options. If we add elementary function evaluations, we can ask the question which precision and range is needed for input and output respectively, and given a function, there is some optimal architecture that provides the desired result while minimizing latency, area or arithmetic units. Of course, the same brute force design space exploration also applies to higher-level functions and whole implementations of algorithms.

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

8 Precision and convergence

For some datasets there are issues of convergence of results when iterating millions of times in a numerical solver, which occasionally arise in some number representations and certain meshing strategies and not in others. On the other hand, some convergence issues might only arise after thousands of iterations, which may take too long on conventional computers but can be reached on an MPC system. Investigating convergence goes hand-in-hand with minimizing precision and optimizing rounding. The third dimension is discretization in time and space: length of time-steps of simulations and the shape of the grid that we are computing the simulation on. The opportunity here is to study the interaction and correlations between precision, discretization in time and space, and convergence of the algorithm. In particular, there is scope to study application specific precision and rounding methods (in space, time and value) based on domain specific criteria for the quality of results, to maximize dataflow computation and provide further levers to better understand the implications of computing digitally in an analog world.

[**Stanford Seminar EE380: Flexible Number Representations for Computing with FPGAs** http://www.stanford.edu/class/ee380/ay0304.html, April 2004.]

9 Domain Specific Languages

Programming DFEs is intellectually stimulating. But how about making it even more exciting by adding Domain Specific Languages (in the spirit of our Finite Difference Compiler)? Or investigating direct dataflow compilation from MATLAB or OpenCL or even Excel? Or translating and mapping established open source software to a hyperfast platform? Of course, the ultimate challenge is to devise an automatic or semi-automatic translation of sequential programs to dataflow, or even translation dynamic object files of applications as they are running.

[**Building Deep, Hazard-free Hardware Pipelines from OpenCL Programs** Stanislaw Czerniawski, MEng Thesis, Department of Computing Imperial College London (Supervised by Paul H J Kelly and Wayne Luk).]

10 Comparisons

Finally, if there is no way to avoid comparing technologies, we believe that the fair approach is to fix the size of the machine (lets say to 1U), for example a 1U MPC-X dataflow appliance, and compare performance and power consumption to a 1U machine from other vendors. Of course, the problem size needs to be significant enough to justify dataflow computing. Alternatively, one can look at larger scale systems with multiple nodes and normalize to the 1U space unit. Furthermore, one can normalize to power consumption, and compare large problem performance per Watt, as long as the power measurement is done at the power socket and not based on some artificial measure of power consumption inside the chip. Finally, another perspective can be obtained by looking at what performance $1M can buy, and then compare the performance and electricity and real-estate costs of the resulting machines over a typical 3 year lifetime. *Buying multi-core processors for $1M could bring another $1M in electricity costs, while buying a dataflow machine for $1M reduces electricity costs to under $100K AND the dataflow machine is faster. How can that be?*

SLiC API Index

MaxJ API Index

Index