Overview

• Real OpenSPL SCS instantiation – Maxeler’s DFEs
• Architecture and Programming Model
• SLIC: Using DFEs from MatLab, Python and more
Maxeler Multiscale Dataflow Computing

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

**Dataflow Engine [DFE]**

\[ P_{\text{avg}} = C_{\text{load}} \cdot V_{\text{DD}}^2 \cdot f \]

PCI Express or Infiniband
Mapping SCS to DFE

- SCS → DFE for Multiscale Computing Platforms
- Spatial Configurable Technology → (largest conf. dev.)
- High-level language used Java
- OpenSPL Compiler → MaxCompiler
- OpenSPL OS → MaxOS
- OpenSPL data movement managed by “Manager.maxj”
- Domain specific support (too big to mention here), e.g., Finite Differences
The Platforms

*decoupling the data plane and control plane*

Multiscale Computing Platforms

- Risk Analytics Software Platform
- Seismic Imaging Software Platform
- Trading Transactions Software Platform

- MaxIDE: MaxCompiler
- MaxSkins: MatLab, Python, R, Excel, C/C++, Fortran

Linux based MaxelerOS: Realtime communication management

Controlflow Box: conventional CPUs

- Fast Interconnect

Dataflow Box: Custom Intelligent Memory System

Three Software Layers

HW layer
Maxeler Accelerator Architecture

Customized DataFlow Engine - DFE

- DFEs run for very long times
- DFE accelerates specific large tasks

*DFK – DataFlow Kernel

DFE==MAX card

DFEs run for very long times,
DFE accelerates specific large tasks.
MaxCompiler

• Complete development environment for Maxeler DFE accelerator platforms
• Write *MaxJ* code to describe the dataflow accelerator
  – *MaxJ* is an extension of Java for MaxCompiler
  – *Execute* the Java → *generate* the accelerator
• C software on CPUs *uses* the accelerator

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

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

        io.output("z", result, dfeFloat(8, 24));
    }
}
```
MaxCompiler Architecture
Application Components

- Host application
- CPU
  - MaxCompilerRT
  - MaxelerOS
- Memory
- PCI Express or Infiniband
- DFE
  - Kernels
- Memory
- Manager

Diagram showing the integration of application components with MaxCompilerRT, MaxelerOS, and memory systems connected through PCI Express or Infiniband interfaces.
DFE based systems

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

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

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

MaxWorkstation
Desktop dataflow development system

Dataflow Engines
48GB DDR3, high-speed connectivity and dense configurable logic
Using DFEs

C/ Python/ MATLAB/etc program

DFE configuration (.max file)

CPU

Interconnect

DFE

CPU Memory

DFE Memory
Development Process

Host Code (.c)

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

Manager m = new Manager();
Kernel k =
    new MyKernel();
m.setKernel(k);
m.setIO(
    link("x", CPU),
    link("y", DRAM_LINEAR1D));
m.build();
Application Execution

1. Application done!
   I’m ready to run it...

2. MaxelerOS allocates a DFE

3. DFE is Configured using .max-file

4. DFE is Ready to use

Application

CPU

MaxelerOS

DFE

.PCIe, InfiniBand

.DFE
MaxCompiler – Create exportable DFE Configurations

C / Python / MATLAB / etc. program

MaxCompiler

.max-file collection

DFEs

CPU

Interconnect

DFE Memory

C

Python

MATLAB

etc.

program

CPU

Memory

Python.

MATLAB.

etc.

program

MaxCompiler

.collecEon

15
.max-files

- DFE configuration data
- OR
- DFE simulation model

- DFE Interface Info
e.g. List of CPU settable values, number and names of I/O streams, etc.

- CPU function code providing ‘Engine Interface’ APIs specific to .max-file
SLiC Interface: CPU <-> DFE API

• **Simple Live CPU Interface:**
  Combination of fixed software function calls and MaxCompiler-generated code for interacting with DFEs

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

• By default all functions in all layers can be used from C
Example SLiC Application

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

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

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

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

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

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

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

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

```
SLiC-Compile tool

MATLAB: .mex + .m
C/C++: .o
Python: .py + .so
R: .tar.gz
```

!.max file
import DFEImageLib

from DFEImageLib import *

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

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

execute the kernel on the DFE
(mask, sharp_img) = DFEImageLib_sharpen(h, w, amount, sigma, img)
% Initialize DFE
dfe = DFEImageLib();

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

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

# Execute the kernel on the DFE
[mask, sharp_img] = dfe.sharpen(amount, sigma, img);
library(jpeg)
# Installed with “R CMD INSTALL DFEImageLib”
library(DFEImageLib)

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

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

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

# Extract output from list
mask    <- outputs$outstream_mask;
Sharp_img <- outputs$outstream_usm;
Types of Interface

• Basic Static:
  – single-function-call interface for using Maxeler engines

• Advanced Static:
  – fine-grain control over which DFEs to use
  – enables use of groups + arrays
  – asynchronous execution

• Dynamic:
  – all functionality of advanced static interface
  – doesn’t require use of statically generated API
  – kernels, scalar I/Os, etc. identified with strings
Summary

• DFEs are the first SCS instantiation
• MaxCompiler and MaxOS together with the Manager and the Chip vendor specific tools (Xilinx or Altera) provide a complete OpenSPL implementation.
• Interfaces to Python, MatLab, C, C++, R and even Fortran...