Overview

- Advanced Static Interface
- Advanced Dynamic Interface
- Debugging
About SLiC

- **Simple Live CPU Interface**
- Allows CPU software to use DFEs

- CPU code must
  - Include `MaxSLiCInterface.h`
  - Include `MaxFile.max` or `MaxFile.h`
  - Link with `libslic.a` and the compiled maxfile
Convolution Kernel

- Simple example computation
  \[ z[i] = a \times (y[i-1]+y[i]+y[i+1]) + x[i] \]
- 2 input streams, 1 input scalar, 1 output stream

```java
public class ConvolveKernel extends Kernel {
    private static final DFEType type = dfeFloat(8,24);
    public ConvolveKernel(KernelParameters parameters) {
        super(parameters);
        DFEVar x = io.input("x", type);
        DFEVar y = io.input("y", type);
        DFEVar a = io.scalarInput("a", type);

        DFEVar conv = stream.offset(y,-1)
                      + y
                      + stream.offset(y,+1);
        DFEVar z = a*conv + x;

        io.output("z", z, type);
    }
}
```
Simple Manager + CPU code

**CPU code (.c)**

```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...
");
    // Allocate x,y,z of sizeBytes
    // Initialize x, y data
    printf("Convolving on DFE...
");
    Convolve(size, coeff1, x, y, z1);
    Convolve(size, coeff2, x, z1, z2);

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

**Manager (.maxj)**

```java
public class ConvolveManager {
    public static void main(String[] args) {
        // Create kernel and manager
        EngineParameters p = new EngineParameters(args);
        Manager m = new Manager(p);
        Kernel k = new ConvolveKernel(
            m.makeKernelParameters());

        // Set-up kernel I/O to/from CPU
        m.setKernel(k);
        m.setIO(
            link("x", IODestination.CPU),
            link("y", IODestination.CPU),
            link("z", IODestination.CPU));

        // Auto-generate simple SLiC interface
        m.createSLiCinterface();

        m.build();
    }
}
```

**SLiC function generated in MaxFile**

```c
void Convolve(int32_t param_N, double inscalar_ConvolveKernel_a,
              const float* instream_x, const float* instream_y,
              float* outstream_z);
```
SLiC basic static

• Use DFE with a single, simple function call
• Any suitable engine will be selected
• After first use, engine will be held until process terminates
• Multiple MaxFiles can be used by one process – each one will get a dedicated engine
• The `createSLiCinterface()` manager call automatically determines a good set of arguments for the SLiC function
  – We will see how to define more complex interfaces later
SLiC levels

• What if we want more control?
  – Exactly which DFE is used
  – Exactly how long the DFE is reserved for
  – If using multiple MaxFiles, should we use 2 engines or share 1?

• SLiC provides three levels of interaction:
  – **Basic Static**: single function calls
  – **Advanced Static**: allows you to *run* multiple *actions* on a single engine with a single maxfile, maintaining state on and control of the engine
  – **Dynamic**: Extension of the advanced static interface using dynamically generated objects to add flexibility at run-time, not limited to static compile-time changes, helps with debugging
# Advanced Static SLiC Level

## CPU code (.c)

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

int main(void)
{
    ...

    printf("Convolving on DFE...

    // Create actions
    Convolve_actions_t act1 = {size,coeff1,x,y,z1};
    Convolve_actions_t act2 = {size,coeff2,x,z1,z2};

    // Load DFE
    max_file_t* maxfile = Convolve_init();
    max_engine_t *eng = max_load(maxfile, "*");

    // Run actions
    Convolve_run(eng, &act1);
    Convolve_run(eng, &act2);

    // Unload DFE
    max_unload(eng);
    max_file_free(maxfile);

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

## Manager (.maxj)

```java
public class ConvolveManager {

    public static void main(String[] args) {

        // Create kernel and manager
        EngineParameters p = new EngineParameters(args);
        Manager m = new Manager(p);
        Kernel k = new ConvolveKernel(
            m.makeKernelParameters());

        // Set-up kernel I/O to/from CPU
        m.setKernel(k);
        m.setIO(
            link("x", IODestination.CPU),
            link("y", IODestination.CPU),
            link("z", IODestination.CPU));

        // Auto-generate simple SLiC interface
        m.createSLiCinterface();
        m.build();
    }
```

### SLiC actions structs and run function generated in MaxFile

```c
typedef struct {
    int32_t param_N;
    double inscalar_ConvolveKernel_a;
    const float* instream_x;
    const float* instream_y;
    float* outstream_z;
} Convolve_actions_t;

void Convolve_run(max_engine_t *engine,
    Convolve_actions_t *interface_actions);
```
Different Stages of using an Engine

- Any use of a DFE has the same basic stages

1. Initialize MaxFile data structure
2. Load MaxFile onto a DFE
3. Run one or more actions
4. Unload DFE
5. Free MaxFile

- Actions are structs that can be created separately from being run
Loading and Unloading DFEs

• Engines must be loaded with MaxFiles before use
• An engine can only be loaded with one maxfile at a time, but at different times can have different maxfiles
• It takes time to load the maxfile and ensures everything is initialized, including memory, etc (100ms-1s)
• The DFE is reserved for exclusive use, and state (DRAM contents, on-chip memories, etc) is kept between load and unload calls
Advanced Dynamic Level

**CPU code (.c)**

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

int main(void)
{
    ...
    printf("Convolving on DFE...\n");

    // Set-up action
    max_file_t* maxfile = Convolve_init();
    max_actions_t* act1 = max_actions_init(maxfile, "default");
    max_set_param_uint64t(act1, "N", size);
    max_set_double(act1, "ConvolveKernel", "a", coeff1);
    max_queue_input(act1, "x", x, sizeBytes);
    max_queue_input(act1, "y", y, sizeBytes);
    max_queue_output(act1, "z", z1, sizeBytes);

    // Load DFE
    max_engine_t *eng = max_load(maxfile, "*");

    // Run action
    max_run(eng, act1);

    // Unload DFE
    max_unload(eng);

    // Free action
    max_actions_free(act1);
    printf("Done.\n");
    return 0;
}
```

- Same semantics as Advanced Static, but

- **Actions** are now dynamically created objects
  1. **Init** action object
  2. **Set values** in action
  3. **Run** action
     (and reuse if desired)
  4. **Free** action object
Engine Identifiers

\[
\text{max_engine_t } *\text{eng} = \text{max_load(maxfile, engine_id);}
\]

- SLiC can run actions on any DFE that in the local node or in an MPC-X Series system on the network
- Engines can be identified by a string
  - \(<\text{Node IP}>:\text{<Engine number}>\)

<table>
<thead>
<tr>
<th>Engine ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Any engine</td>
</tr>
<tr>
<td>:0</td>
<td>Engine 0 in the default_engine_resource</td>
</tr>
<tr>
<td>local:1</td>
<td>Engine 1 in the local node</td>
</tr>
<tr>
<td>mpcx001:*</td>
<td>Any engine in mpcx001</td>
</tr>
<tr>
<td>mpcx003:7</td>
<td>Engine 7 in mpcx003</td>
</tr>
</tbody>
</table>

- \text{default_engine_resource} is defined in SLIC_CONF and selects the default node to use DFEs from
## Comparing SLiC Levels

<table>
<thead>
<tr>
<th></th>
<th>Basic Static</th>
<th>Advanced Static</th>
<th>Advanced Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operating model</strong></td>
<td>Simple function call</td>
<td>Construct actions object and run on engine</td>
<td></td>
</tr>
<tr>
<td><strong>Engine loads</strong></td>
<td>On first use, can’t control which DFE</td>
<td>Explicitly on <code>max_load</code></td>
<td></td>
</tr>
<tr>
<td><strong>Engine unloads</strong></td>
<td>On process exit</td>
<td>Explicitly on <code>max_unload</code></td>
<td></td>
</tr>
<tr>
<td><strong>Actions are</strong></td>
<td>Not needed</td>
<td>Struct</td>
<td>Object</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>Simplest, easiest</td>
<td>Moderate</td>
<td>Complex</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td><strong>Dependency on specific MaxFile</strong></td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Main uses</strong></td>
<td>Simple applications or self-contained functionality</td>
<td>Applications needing explicit control over which DFEs are used or what actions are run</td>
<td>Maximum control over actions, debugging and for meta-programming with maxfiles</td>
</tr>
</tbody>
</table>

- **Operating model**: The Basic Static model is simple with function calls, while the Advanced Static and Dynamic models require actions to be constructed and run on the engine, respectively. The Advanced Dynamic model allows maximum control over actions, debugging, and meta-programming.
- **Engine loads**: In Basic Static, engine loads are on first use and cannot be controlled. Advanced models allow explicit control on `max_load`.
- **Engine unloads**: Basic Static has no explicit unloading. Advanced models can be explicitly unloaded on `max_unload`.
- **Actions are**: Not needed in Basic Static, Struct in Advanced Static, and Object in Advanced Dynamic.
- **Complexity**: Simplest in Basic Static, Moderate in Advanced Static, and Complex in Advanced Dynamic.
- **Flexibility**: Low in Basic Static, Medium in Advanced Static, and High in Advanced Dynamic.
- **Dependency on specific MaxFile**: High in all models, but in Advanced Static, it is high, while in Advanced Dynamic, it is low.
- **Main uses**: Simple applications in Basic Static, advanced applications in Advanced Static, and all advanced uses in Advanced Dynamic.
Engine Interfaces

• So far our MaxFile has exported one interface: Convolve
• Manager can declare multiple user-defined interfaces

• Why use user-defined interfaces?
  – Provide multiple functions in the same MaxFile
  – Set multiple complex on-chip values from a small number of meaningful user parameters

• Up to now we’ve used an auto-generated ‘good’ interface
  – Auto-generation only works for simple cases
  – User-defined interfaces allow us to create similar ‘good’ interfaces to arbitrarily complex DFE configurations

• All interfaces are based on the full interface
The Full Interface

**CPU code (.c)**

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

int main(void)
{
    ...

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

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

void Convolve(
    uint64_t ticks_ConvolveKernel,
    double inscalar_ConvolveKernel_a,
    const void* instream_x,
    size_t instream_size_x,
    const void* instream_y,
    size_t instream_size_y,
    void* outstream_z,
    size_t outstream_size_z);```

**Full interface SLiC function assumes all values could vary independently**

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

int main(void)
{
    ...

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

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

void Convolve(
    uint64_t ticks_ConvolveKernel,
    double inscalar_ConvolveKernel_a,
    const void* instream_x,
    size_t instream_size_x,
    const void* instream_y,
    size_t instream_size_y,
    void* outstream_z,
    size_t outstream_size_z);```

**Manager (.maxj)**

```java
public class ConvolveManager {

    public static void main(String[] args) {

        // Create kernel and manager
        EngineParameters p = new EngineParameters(args);
        Manager m = new Manager(p);
        Kernel k = new ConvolveKernel(
            m.makeKernelParameters());

        // Set-up kernel I/O to/from CPU
        m.setKernel(k);
        m.setIO(
            link("x", IODestination.CPU),
            link("y", IODestination.CPU),
            link("z", IODestination.CPU));

        // Assign a default, empty interface
        m.createSLiCinterface(interfaceDefault());

        m.build();
    }

    private static EngineInterface interfaceDefault() {
        EngineInterface i = new EngineInterface();
        return i;
    }
```
User defined interfaces

• How can we simplify the full interface for Convolve to get the better interface back?
  – Add an extra parameter (N), and pre-set some arguments that were present in the full interface

```java
private static EngineInterface interfaceDefault() {
    EngineInterface i = new EngineInterface();
    CPUTypes type = CPUTypes.FLOAT;
    int size = type.sizeInBytes();

    InterfaceParam N = i.addParam("N", CPUTypes.INT);
    i.setTicks("ConvolveKernel", N);
    i.setStream("x", type, N * size);
    i.setStream("y", type, N * size);
    i.setStream("z", type, N * size);
    return i;
}
```

Could pass a String argument to create a specific named interface instead of the default

1. Add a single dataset size param N
2. Set the kernel to run for N ticks
3. Set the streams to be of type float and size N*sizeof(float)
A more complex interface

CPU code (.c)

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

int main(void) {
    ...
    printf("Uploading x data to DFE.\n")
    Convolve_writeLMem(0, sizeBytes, x);
    printf("Convolving y on DFE...\n")
    Convolve_mul4(size, y, z1);
    printf("Done.\n")
    return 0;
}
```

SLiC interface function

```c
void Convolve_mul4(
    int64_t param_N,
    const float* instream_y,
    float* outstream_z);
```

- Input stream x will be read from LMem
- Scalar coefficient a already set to 4.0

Standard Manager automatically generates extra interfaces to read/write memory

```c
void Convolve_writeLMem(
    int64_t param_address, int64_t param_nbytes, 
    const void* instream_cpu_to_lmem);
```

```c
// Set-up kernel I/O to/from CPU
m.setKernel(k);
m.setIO(
    link("x", IODestination.LMEM_LINEAR_1D),
    link("y", IODestination.CPU),
    link("z", IODestination.CPU));
```

```c
// Interface to manage reading x from LMem
m.createSLiCinterface(interfaceMul4());
m.build();
```

```c
private static EngineInterface interfaceMul4() {
    EngineInterface i = new EngineInterface("mul4");
    CPUPtypes type = CPUPtypes.FLOAT;
    int size = type.sizeInBytes();
    InterfaceParam N = i.addParam("N", CPUPtypes.INT);
    i.setTicks("ConvolveKernel", N);
    i.setScalar("ConvolveKernel", "a", 4.0);
    i.setLMemLinear("x", i.addConstant(0l), N*size);
    i.setStream("y", type, N * size);
    i.setStream("z", type, N * size);
    i.ignoreAll(Direction.IN_OUT);
    return i;
}
```
Interfaces in the Advanced SLiC levels

### Advanced Static CPU code (.c)

```c
int main(void)
{
    ...
    // Load engine
    max_file_t* maxfile = Convolve_init();
    max_engine_t* eng = max_load(maxfile, ".c");

    printf("Writing x data to DFE LMem.\n");
    Convolve_writeLMem_actions_t act_load =
        { 0, sizeBytes, x };
    Convolve_writeLMem_run(eng, &act_load);

    printf("Convolving on DFE...\n");
    Convolve_mul4_actions_t act_compute =
        { size, y, z1 };
    Convolve_mul4_run(eng, &act_compute);

    // Unload engine
    max_unload(eng);

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

### Advanced Dynamic CPU Code (.c)

```c
int main(void)
{
    ...
    // Load engine
    max_file_t* maxfile = Convolve_init();
    max_engine_t* eng = max_load(maxfile, ".c");

    printf("Writing x data to DFE LMem.\n");
    max_actions_t* act_load =
        max_actions_init(maxfile, "writeLMem");
    max_set_param_uint64t(act_load, "address", 0);
    max_set_param_uint64t(act_load, "nbytes", sizeBytes);
    max_queue_input(act_load, "cpu_to_lmem", x, sizeBytes);
    max_run(eng, act_load);

    printf("Convolving on DFE...\n");
    max_actions_t* act_compute =
        max_actions_init(maxfile, "mul4");
    max_set_param_uint64t(act_compute, "N", size);
    max_queue_input(act_compute, "y", y, sizeBytes);
    max_queue_output(act_compute, "z", z1, sizeBytes);
    max_run(eng, act_compute);

    // Unload engine, can also free actions
    max_unload(eng);

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

- Non-blocking run functions return immediately, allowing CPU execution to continue
- Functions return a *run handle*
  - At some point later must *wait* or *nowait* this handle

Advanced Static CPU code (.c)

```c
int main(void)
{
    ...
    Convolve_actions_t a1, a2, a3;
    // Load DFE and prepare actions to run

    max_wait_t* w1 = Convolve_run_nonblock(eng, &a1);
    max_wait_t* w2 = Convolve_run_nonblock(eng, &a2);
    max_wait_t* w3 = Convolve_run_nonblock(eng, &a3);

    // Run other computation on CPU in parallel
    ...
    // Synchronize when last action has completed
    max_nowait(w1);
    max_nowait(w2);
    max_wait(w3);
    ...
}
```

Advanced Dynamic CPU code (.c)

```c
int main(void)
{
    ...
    max_actions_t* a1, a2, a3;
    // Load DFE and prepare actions to run

    max_wait_t* w1 = max_run_nonblock(eng, &a1);
    max_wait_t* w2 = max_run_nonblock(eng, &a2);
    max_wait_t* w3 = max_run_nonblock(eng, &a3);

    // Run other computation on CPU in parallel
    ...
    // Synchronize when last action has completed
    max_nowait(w1);
    max_nowait(w2);
    max_wait(w3);
    ...
}
```
Arrays of DFEs

- **Fixed size set of engines** loaded with the same MaxFile
- **MaxRing connections** between engines in the array
- The whole array can be run with a single command

- Advanced static & dynamic interfaces only
  - Load with `max_load_array`
  - Run with `<MaxFile>_run_array` (static) or `max_run_array` (dynamic)
Groups

• Groups are pools of engines with the same MaxFile that can be shared
  – Multiple processes on multiple nodes can share multiple DFEs

• Can have a fixed size, or can change size dynamically depending on demand at run-time

• A single process can lock an engine from a group
  – Provides exclusive use for the duration of the lock

• Useful for optimizing execution of short actions where DFEs all have the same state e.g. searching
More Information

• *Multiscale Dataflow Programming* tutorial contains an chapter on advanced SLiC usage

• Full API documentation in `$MAXCOMPILERDIR/docs/SLiC-Interface-API`
  – (Or access via Help->Welcome in MaxIDE)

• MaxIDE will auto-complete SLiC names in your C code

• MaxIDE new project wizard auto-generates template SLiC code
Project: Using DFEs, WebIDE
Create New Project

Give it a name you like and select the “Using DFE” template
#include "DFE.h"  // Includes .max files
#include <MaxSLiCInterface.h>  // Simple Live CPU interface

float dataIn[8] = { 1, 0, 2, 0, 4, 1, 8, 3 };  
int dataIn_NEW[24] = { 1, 0, 2, 0, 4, 1, 8, 3, 1, 0, 2, 0, 4, 1, 8, 3, 1, 0, 2, 0, 4, 1, 8, 3 };  
float dataOut[8]; 

int main()
{
    printf("Running DFE\n");
    
    DFE(8, dataIn, dataOut);  // call DFE(NR_OF_ELEMENTS, INPUT_STREAM, OUTPUT_STREAM)
    for (int i = 1; i < 7; i++)  // Ignore edge values
        printf("dataOut[%d] = %f\n", i, dataOut[i]);

    return 0;
}
Using DFE (via WebIDE)

• Run the DFE and try to understand the console output (compiler and runtime)
• Check correctness of the produced results
• Modify the CPU code to use `dataIn_NEW`
• Make sure that everything in the CPU code is modified accordingly
• Make sure that the output is correct*

* - pay special attention to compiler warnings and errors
Using DFE (continue)

• There are more examples to explore
• You can try to understand more of those
• Please pay attention that WebIDE is on a virtual machine
Summary

• Advanced interfaces provide more control over DFEs
• More control comes with additional complexity
• SLIC allows you to create your own convenient versions of your interface
• Debugging DFEs is complex and requires special approach (more later)
• Get familiar with the WebIDE since you will use it for your final project