CO405H

Computing in Space with OpenSPL

Topic 6: Programming DFEs (advanced)

Oskar Mencer **Georgi Gaydadjiev**

Department of Computing
Imperial College London

http://www.doc.ic.ac.uk/~oskar/
http://www.doc.ic.ac.uk/~georgig/

CO405H course page: http://cc.doc.ic.ac.uk/openspl14/
WebIDE: http://openspl.doc.ic.ac.uk
OpenSPL consortium page: http://www.openspl.org

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

Overview

- Advanced Static Interface
- Advanced Dynamic Interface
- Debugging

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)

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

int main(void)
{
  ...

  printf("Convolving on DFE...\n");

  // 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)

```
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();
```

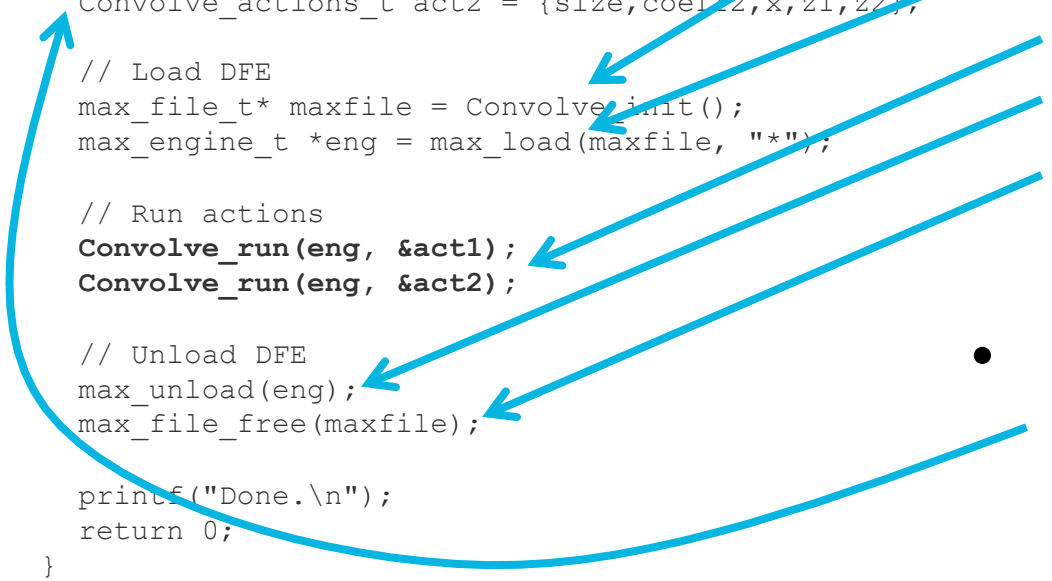
SLiC *actions* structs and run function generated in MaxFile

```
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

CPU code (.c)

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

int main(void)
{
  ...

  printf("Convolving on DFE...\n");

  // 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;
}
```



- 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)

```
#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

```
max_engine_t *eng = 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
  - <Node IP>:<Engine number>

Engine ID	Description
*	Any engine
:0	Engine 0 in the default_engine_resource
local:1	Engine 1 in the local node
mpcx001:*	Any engine in mpcx001
mpcx003:7	Engine 7 in mpcx003

- default_engine_resource is defined in SLIC_CONF and selects the default node to use DFEs from

Comparing SLiC Levels

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

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)

```
#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;
}
```

Full interface SLiC function assumes all values could vary independently

```
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)

```
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

```
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)

```
void Convolve(int64_t param_N,
              double inscalar_ConvolveKernel_a,
              const float* instream_x,
              const float* instream_y,
              float* outstream_z);
```

A more complex interface

CPU code (.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;
}
```

Standard Manager automatically generates extra interfaces to read/write memory

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

```
  ...
  // 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));

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

private static EngineInterface interfaceMul4() {
  EngineInterface i = new EngineInterface("mul4");
  CPUTypes   type = CPUTypes.FLOAT;
  int        size = type.sizeInBytes();
  InterfaceParam N = i.addParam("N", CPUTypes.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;
}
}
```

SLiC interface function

```
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

Interfaces in the Advanced SLiC levels

Advanced Static CPU code (.c)

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

  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)

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

  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)

```
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)

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

  max_wait_t* w1 = max_run(eng, a1);
  max_wait_t* w2 = max_run(eng, a2);
  max_wait_t* w3 = max_run(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

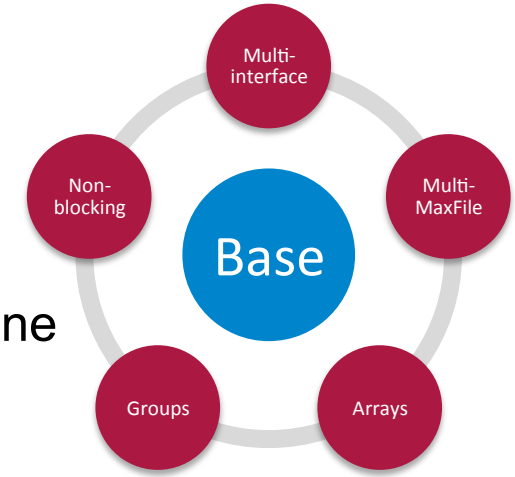
- Advanced static & dynamic interfaces only
  - Load with `max_load_array`
  - Run with `<MaxFile>_run_array` (static) or `max_run_array` (dynamic)

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

Base Interface



- The advanced static interface uses C structs to hold *actions* (parameters) which are sent to the dataflow engine by a generated *run* function.

```
#include "MaxSLiCInterface.h"
#include "AVG.max" // or #include "AVG.h"

void myfn(float *x, float *y, int n) {
    AVG_actions_t a1;
    a.instream_x = x;
    a.outstream_y = y;
    a.param_N = n;
    AVG_actions_t a2;
    ...
    max_file_t *maxfile = AVG_init();
    max_engine_t* eng1 = max_load(maxfile, "*"); // "*" => any engine
    printf("ENGINE: %s\n", eng1->id);
    AVG_run(eng1, &a1);
    AVG_run(eng1, &a2);
    max_unload(eng1);
}
```

Struct field-names and types are identical to those used in the basic interface.

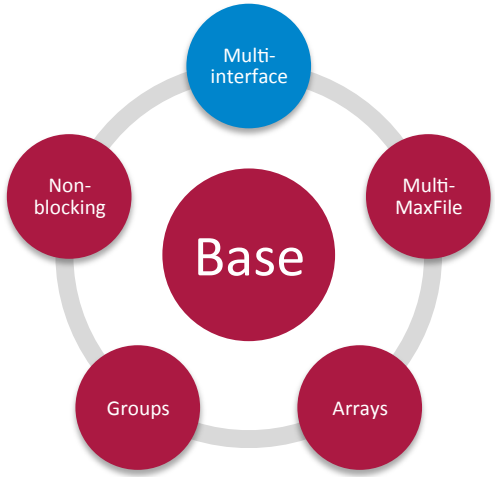
Auto-generated run functions.

Identifying engines

- “*” – Any engine.
- “local:*” – Any local device.
- “local:1” – Local engine 1
- “<hostname>:*” – Any device in MPC-X system <hostname>

- Engines will only be used if they support a given .max-file

Multi-interface

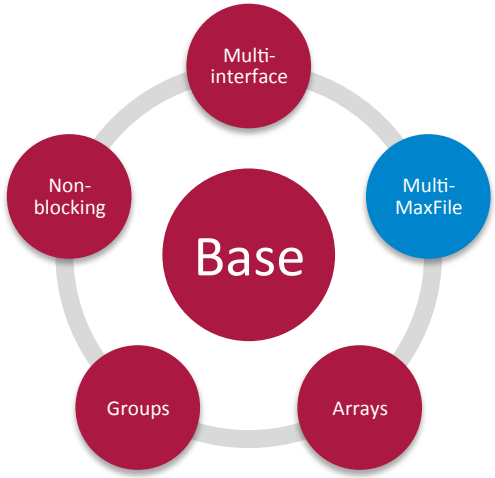



```
#include "MaxSLiCInterface.h"
#include "AVG.max" // or #include "AVG.h"


void myfn(float *x, float *y, int n) {
   AVG_actions_t a1; // actions for "default" interface
   a.instream_x = x;
   a.outstream_y = y;
   a.param_N = n;
   AVG_special_actions_t a2; // actions for "special" interface
   ...
   max_file_t *maxfile = AVG_init();
   max_engine_t* eng1 = max_load(maxfile, "*");
   printf("ENGINE: %s\n", eng1->id);
   AVG_run(eng1, &a1);
   AVG_special_run(eng1, &a2);
   max_unload(eng1);
}
```

Each interface has a different action struct-type with possibly different parameters

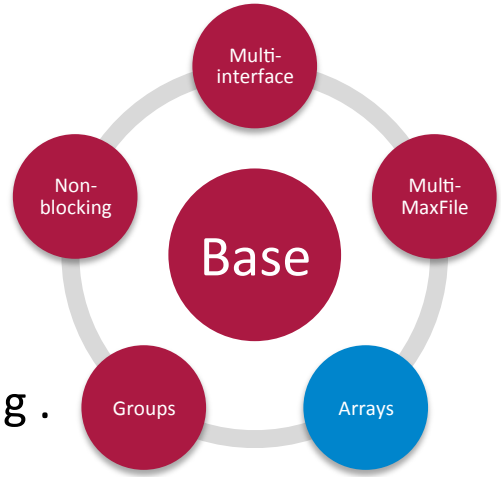
Multi-maxfile

```
#include "MaxSLiCInterface.h"
#include "AVG.max" // or #include "AVG.h"
#include "SUM.h" // or #include "SUM.max"
```




```
void myfn(...) {
    max_file_t *avg_maxfile = AVG_init();
    max_file_t *sum_maxfile = SUM_init();

    max_engine_t* eng = max_load(avg_maxfile, "*");
    ...
    max_unload(eng);

    eng = max_load(sum_maxfile, "*");
    ...
    max_unload(eng);

}
```

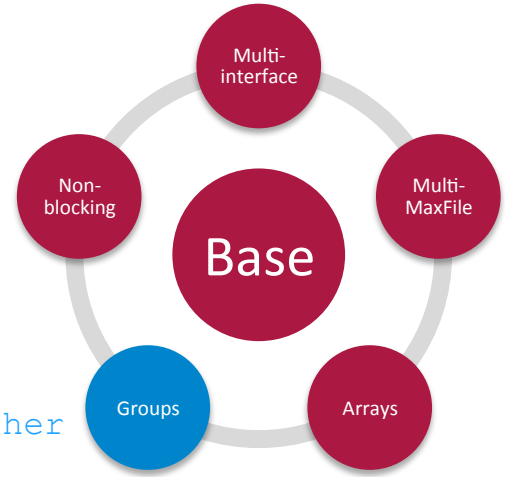
Array interface



- Functions ending in _array act on multiple MaxRing connected engines, atomically as a single unit.
- `max_load_array` loads N engines connected via MaxRing .

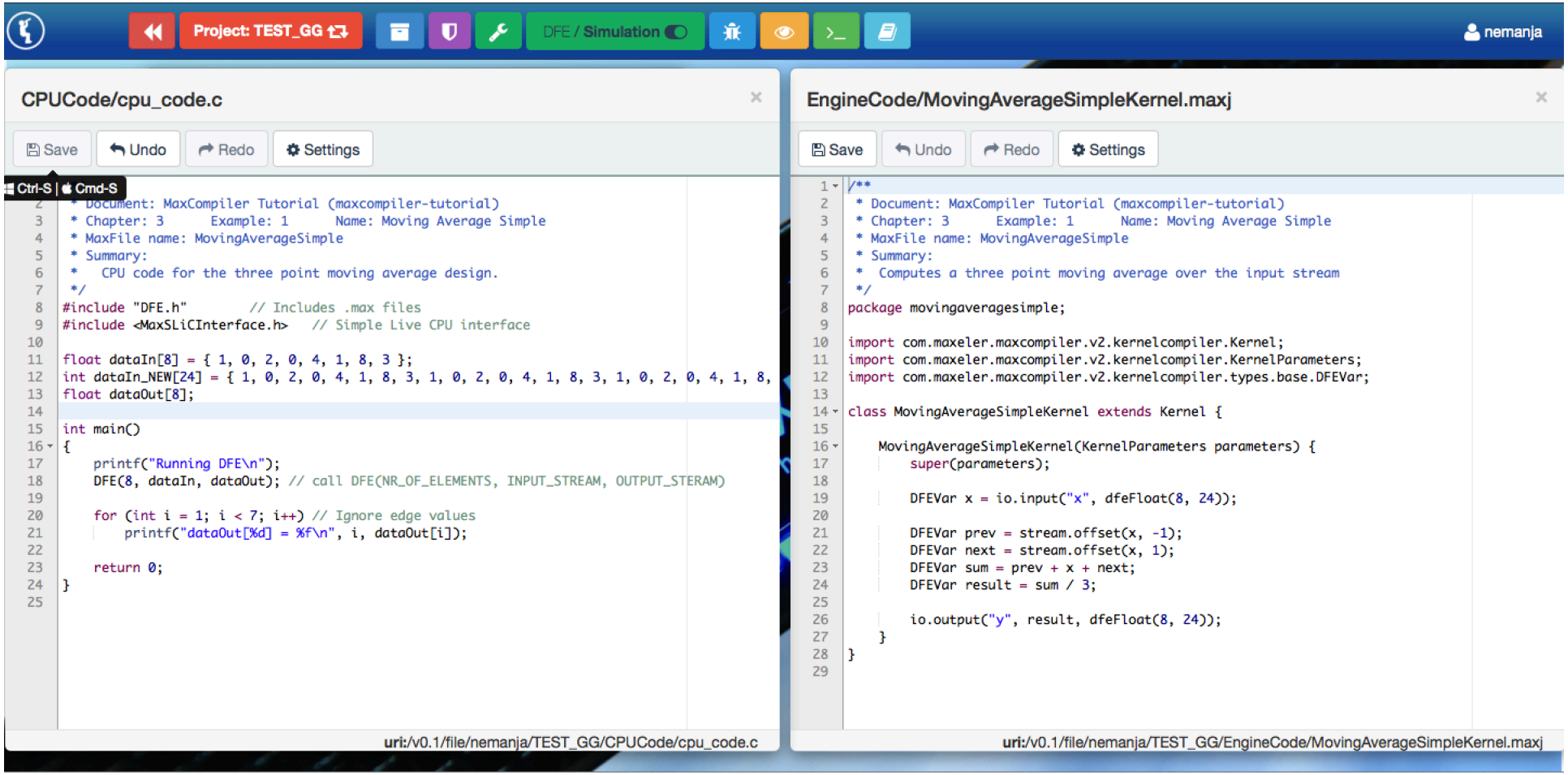
```
void myfn(...) {
    max_file_t* maxfile = AVG_init();
    AVG_actions_t act1, act2, act3;

    // Fill in AVG_action_t objects as normal

    max_engarray_t* engs = max_load_array(maxfile, 3, "*");
    printf("%s %s %s", engs->ids[0], engs->ids[1], engs->ids[2]);
    // Run array of actions on the array of engines.
    AVG_actions_t *acts[] = {&act1, &act2, &act3};
    AVG_run_array(engs, &acts);
    max_unload_array(engs);
}
```

Group Interface

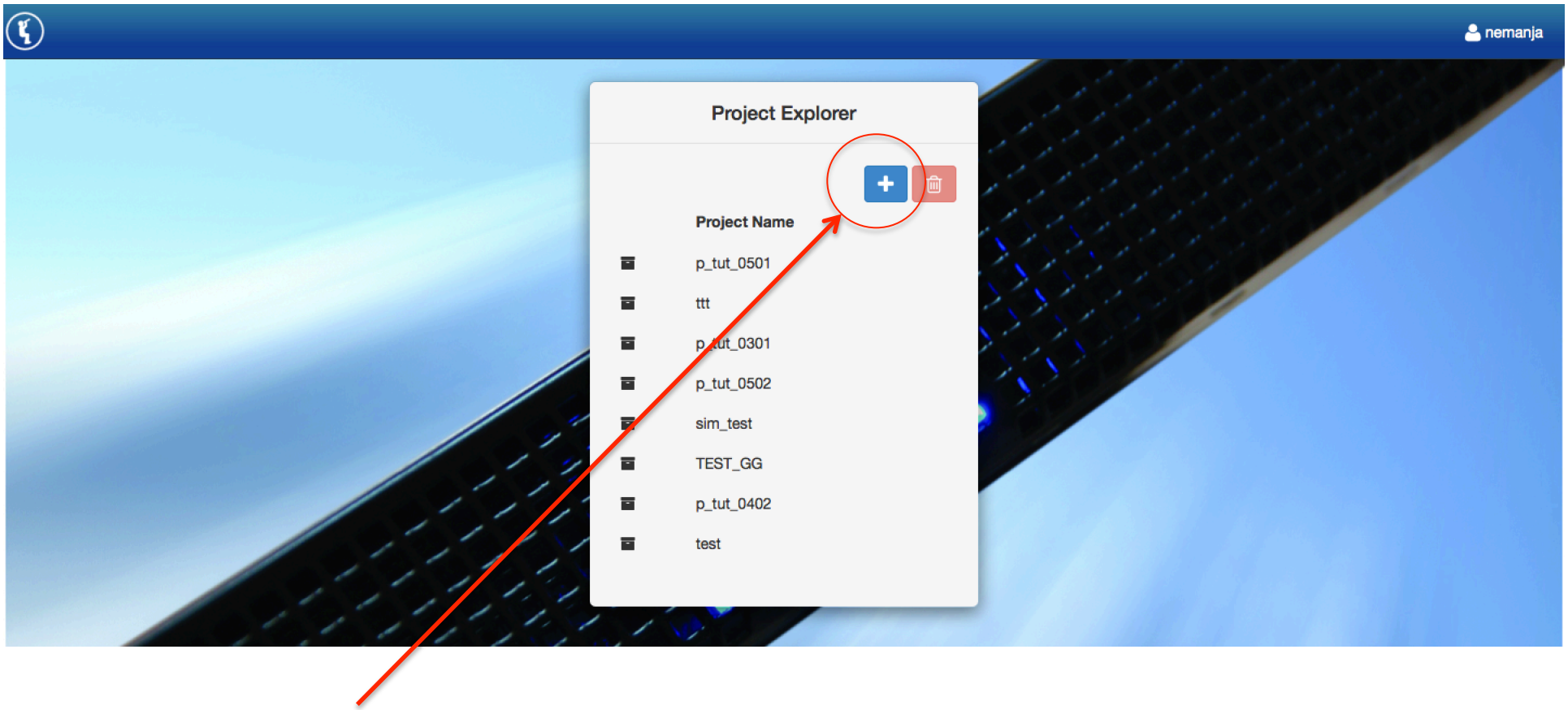



```
void myfn(...) {
    max_file_t *maxfile = AVG_init();
    AVG_actions_t act1, act2;

    // Load a named group of engines that may be shared with other
    // processes. MAXOS_SHARED is group-sharing mode.
    max_group_t *grp = max_load_group(
          maxfile, MAXOS_SHARED, "grouptag@hostname", num);

    ... // sometime later

    // Set-up actions as normal

    // We can lock and run on the engines as normal
    max_engine_t *eng = max_lock_any(grp);
    AVG_run(eng, &act1);
    AVG_run(eng, &act2);
    max_unlock(eng);

    // Or, run on an engine atomically (fast)
    AVG_run_group(grp, &act1);
    AVG_run_group(grp, &act2); // no state preserved between these runs
}
```

Project: Using DFEs, WebIDE



Project: Using DFEs, WebIDE (cont)



Create New Project

Give it a name you like and select the “Using DFE” template

Project Using DFEs (moving average)

```
#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_STERAM
        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